QDE.

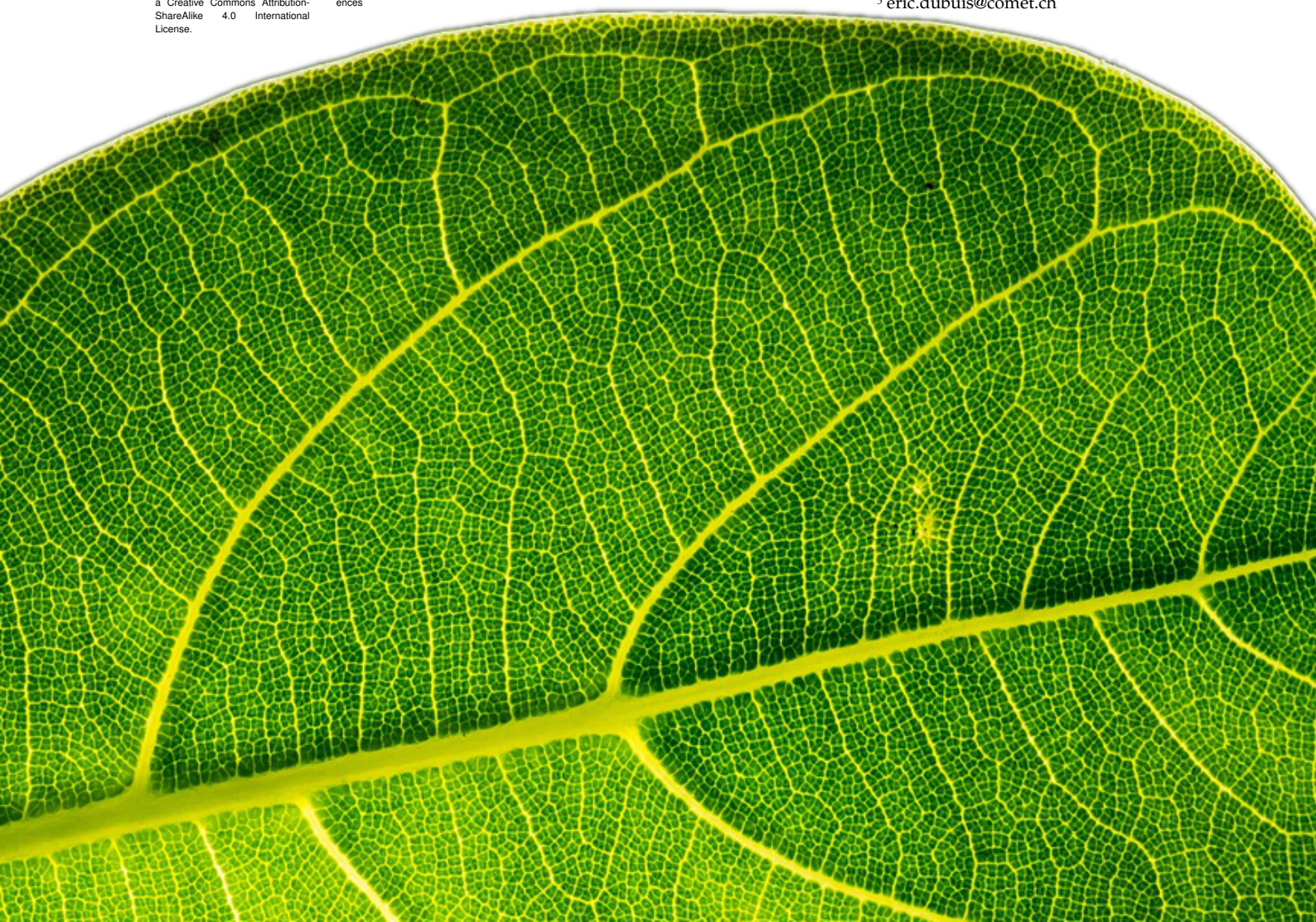
A SYSTEM FOR COMPOSING REAL TIME COMPUTER GRAPHICS.

MTE7103 — MASTER THESIS

Major:	Computer science
Author:	Sven Osterwalder[1]
Advisor:	Prof. Claude Fuhrer[2]
Expert:	Dr. Eric Dubuis[3]
Date:	2017-07-30 22:29:46
Version:	3cbeodc

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

a1e709d	2017-06-27 12:43:16	SO	Update meeting minutes, restructure appendix, add results
52bb736	2017-07-11 17:16:21	SO	Remove obsolete parts, first corrections
2a95fb7	2017-07-20 16:19:39	SO	Finish corrections and implementation
9f4da06	2017-07-28 23:56:18	SO	Final version of the main part
3cbeodc	2017-07-30 22:29:46	SO	Addendum to original hand-in: fix scraps and references

Abstract

Computer science has always strived to create representations of scenes and models as near to human reality as possible. One such representation is *ray tracing*, based on the physics of light as well as the properties of surface materials. In contrast to ray tracing, *sphere tracing* allows the rendering of ray traced images in real-time.

The de facto way of representing objects however, using triangle based meshes, cannot be used directly with this method. Instead, an alternative basis uses distance fields defined by implicit functions. At present there are no solutions known to the author that provide a convenient way to use implicit functions for modeling and sphere tracing for rendering.

This thesis presents the design and development of a program which provides both a modular, node based approach for modeling and animating objects using implicit functions, and allows rendering in real-time using sphere tracing.

To reach the intended goal, the approach was to develop a software architecture and to use literate programming and the agile methodology of extreme programming for development. *Literate programming* considers programs as works of literature. A literate program includes documentation which explains in human terms what the computer must do.

The results of this thesis are an architecture for a software program, and the program itself, written using the literate programming paradigm.

Three aspects define the software architecture: (1) the model-view-view model software design pattern using controllers in addition, (2) the layered software architectural pattern and (3) the observer software design pattern, allowing communication between components of the software.

The software itself is an editor which allows modeling objects, composing objects to scenes and rendering scenes in real-time. Scenes are stored in a scene graph structure and are represented by a tree view. Each scene can contain one or more objects, defined by exter-

nal files and represented as nodes in the graph structure. The parameters of objects are used for interconnections between nodes. For rendering, the sphere tracing algorithm established in the preceding project work was used.

LITERATE PROGRAMMING TAKES SOME ACCUSTOMIZATION as it requires a different way of thinking to that of conventional software development. Despite this, the basic goals could be reached. Sphere tracing is a very interesting and promising approach to rendering which is coming into use in the industry, for example for calculating ambient occlusion or soft shadows. Time will tell if the method will establish itself further and become practicable for rendering conventional meshes.

Contents

List of Figures

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

Editor → Scene model 93

62 The constructor of the scene model.

Editor → Scene model → Constructor 94

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

Editor → Scene graph view model 94

64 The constructor of the scene graph view model.

Editor → Scene graph view model → Constructor 95

65 The scene graph controller, inheriting from `QAbstractItemModel`.

Editor → Scene graph controller 96

66 Constructor of the scene graph controller.

Editor → Scene graph controller → Constructor 96

67 A method to add the root node from within the scene graph controller.

Editor → Scene graph controller → Methods 97

68 The root node of the scene graph being added by the main application.

Editor → Main application → Constructor 97

69 Initialization of the header data and the root node of the scene graph.

Editor → Scene graph controller → Constructor 98

70 Implementation of QAbstractItemModel's index method for the scene graph controller.

Editor → Scene graph controller → Methods 99

71 Implementation of QAbstractItemModel's parent method for the scene graph controller.

Editor → Scene graph controller → Methods 100

72 Implementation of QAbstractItemModel's columnCount method for the scene graph controller.

Editor → Scene graph controller → Methods 101

73 Implementation of QAbstractItemModel's rowCount method for the scene graph controller.

Editor → Scene graph controller → Methods 102

74 Implementation of QAbstractItemModel's data method for the scene graph controller.

Editor → Scene graph controller → Methods 103

75 Implementation of QAbstractItemModel's headerData method for the scene graph controller.

Editor → Scene graph controller → Methods 104

76 Scene graph view models hold references to the nodes they contain.

Editor → Scene graph view model → Constructor 104

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

Editor → Scene graph view model → Methods 105

78 The scene graph controller gets initialized within the main application.

Editor → Main application → Constructor 105

79 Scene graph view, based on Qt's QTreeView.

Editor → Scene graph view 105

80 Constructor of the scene graph view.

Editor → Scene graph view → Constructor 106

81 The scene graph controller is being set as the scene graph view's model.

Editor → Main application → Constructor 106

82 Slot which is called when the selection within the scene graph view is changed.

Editor → Scene graph view → Slots 107

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

Editor → Scene graph view → Methods 108

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

Editor → Scene graph view → Signals 108

85 Signals that get emitted whenever a scene is added or removed.

Editor → Scene graph view → Signals 109

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

Editor → Scene graph view → Constructor 109

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

Editor → Scene graph view → Constructor 110

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

Editor → Scene graph view → Slots 111

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

Editor → Scene graph controller → Slots 112

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

Editor → Scene graph controller → Methods 113

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

Editor → Scene graph controller → Methods 114

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

Editor → Main application → Constructor 114

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

Editor → Scene graph controller → Signals 115

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

Editor → Main application → Methods 117

95 Set up of the logging from within the main application class.

Editor → Main application → Constructor 117

96 An example of how to use the logging decorator in a class. 119

97 The scene graph view logs a corresponding message whenever an item is added from the scene graph. Note, that this logging only happens in *debug* mode.

Editor → Scene graph view → Methods 119

98 The scene graph view logs a corresponding message whenever an item is removed from the scene graph. Note, that this logging only happens in *debug* mode.

Editor → Scene graph view → Methods 119

99 Connections between nodes in EBNF notation. 123

100 Types of a node wrapped in a class, implemented as an enumerator.

Editor → Types → Node type 124

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

Editor → Parameters → Atomic type 125

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

Editor → Parameters → Atomic types 126

103 Definition of the node (domain) model.

Editor → Node model 127

104 Constructor of the node (domain) model.

Editor → Node model → Constructor 127

105 Definition of the node view model.

Editor → Node view model 128

106 Constructor of the node view model.

Editor → Node view model → Constructor 128

107 The type attribute of the node view model as property.

Editor → Node view model → Methods 129

108 The name attribute of the node view model as property.

Editor → Node view model → Methods 129

109 The type attributes of the node domain model as property.

Editor → Node (domain) model → Methods 130

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

Editor → Node view model → Methods 130

111 The cache key is being initialized within a node's constructor.

Editor → Node view model → Constructor 131

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

Editor → Node view model → Methods 131

113 The status of the node is being initialized within the node's constructor.

Editor → Node domain model → Constructor 131

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

Editor → Node domain model → Methods 132

115 A cache key is being created when no cache key for the given attributes is found.

Editor → Node view model → Methods → Paint 132

116 Based on the created or retrieved cache key a pixmap is being searched for.

Editor → Node view model → Methods → Paint 132

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

Editor → Node view model → Methods → Paint 133

118 Definition of the scene view component, derived from the QGraphicsView component.

Editor → Scene view 134

119 Constructor of the scene view component.

Editor → Scene view → Constructor 134

120 The scene view component is being set up by the main window.

Editor → Main window → Methods → Setup UI 135

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

Editor → Main window → Methods → Setup UI 136

122 Whenever a scene is selected in the scene graph, the scene graph controller informs the scene controller about that selection.

Editor → Main application → Constructor 136

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

Editor → Main application → Constructor 136

124 Definition of the scene controller.

Editor → Scene controller 137

125 The scene controller being set up by the main application.

Editor → Main application → Constructor 137

126 Definition of the scene view model.

Editor → Scene view model 138

127 Constructor of the scene view model component.

Editor → Scene view model → Constructor 138

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

Editor → Scene view model → Methods 139

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

Editor → Scene controller → Constructor 140

130 The slot which gets triggered whenever a new scene is added via the scene graph.

Editor → Scene controller → Slots 141

131 The slot which gets triggered whenever a scene is removed via the scene graph.

Editor → Scene controller → Slots 142

132

Editor → Scene controller → Slots 142

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

Editor → Scene controller → Signals 143

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

Editor → Scene view → Slots 143

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

Editor → Main application → Constructor 143

136 Definition of a node for an implicitly defined sphere.

Implicit sphere node 144

137 Radius of the implicit sphere node as input.

Implicit sphere node → Inputs 145

138 The output of the implicit sphere node, which is of the atomic type implicit.

Implicit sphere node → Outputs 145

139 Implementation of the sphere in the OpenGL Shading Language (GLSL).

Implicit sphere node → Definitions 146

140 The position of the implicit sphere node as invocation.

Implicit sphere node → Invocations 146

141 The "body" of the implicit sphere node as node part.

Implicit sphere node → Parts 147

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

Implicit sphere node → Connections 148

143 Definition of the node controller.

Editor → Node controller 149

144 Constructor of the node controller.

Editor → Node controller → Constructor 149

145 A method that loads node definitions from external files from within the node controller.

Editor → Node controller → Methods 150

146 Definition of a part of a node definition.

Editor → Node definition part 150

147 Constructor of the node definition part.

Editor → Node definition part 151
148 The node controller keeps track of node definition parts.

Editor → Node controller → Constructor 151
149 Helper function which creates a value function from the given value.

Editor → Node domain model → Module methods 152
150 Definition of the value function class which is used within nodes.

Editor → Value function 153
151 Definition of the default value function class, which is derived from the value function class.

Editor → Default value function 154
152 The node part class.

Editor → Node part 155
153 Constructor of the node part class.

Editor → Node part 155
154 Definition of the function class which is used in parts of nodes.

Editor → Function 156
155 A class which holds the possible values of a state change of a node part.

Editor → State change 156
156 The node controller provides the atomic types which build the basis of the part of a node.

Editor → Node controller → Methods → Load nodes 157
157 Definition of the node definition class, which represents the definition of a node.

Editor → Node definition 157
158 Constructor of the node definition class.

Editor → Node definition → Constructor 158
159 The node controller holds a dictionary containing node definitions.

Editor → Node controller → Constructor 158
160 The node controller loads and parses node definition files from the file system.

Editor → Node controller → Methods → Load nodes 159

161 A method which tries to load a node definition from the file system using the provided file name.

Editor → Node controller → Methods → Load node definition from file name 161

162 Whenever a new node definition gets loaded, the node controller emits a corresponding signal containing the node view model for the loaded node definition.

Editor → Node controller → Methods → Load node definitions 162

163 A class method of the JSON module, which loads a node definition from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Load node definition 163

164 The inputs of a node definition are parsed and then added to the list of known inputs.

Editor → JSON → Methods → Load node definition 164

165 The outputs of a node definition are parsed and then added to the list of known outputs.

Editor → JSON → Methods → Load node definition 164

166 The child nodes of a node definition are parsed and then added to the list of known child nodes.

Editor → JSON → Methods → Load node definition 164

167 The connections of a node definition are parsed and then added to the list of known connections.

Editor → JSON → Methods → Load node definition 164

168 The definitions of a node definition are parsed and then added to the list of known definitions.

Editor → JSON → Methods → Load node definition 165

169 The invocations of a node definition are parsed and then added to the list of known invocations.

Editor → JSON → Methods → Load node definition 165

170 The parts of a node definition are parsed and then added to the list of known parts.

Editor → JSON → Methods → Load node definition 165

171 A class method of the JSON module, which builds the input of a node definition from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition input 166

172 A method of the node controller, which returns a node definition part by a provided identifier. If no node definition part is found for the given identifier, a new node definition part is created.

Editor → Node controller → Methods → Get node definition part 167

173 Method of the parameter module, which creates an object of a specific value instance based on the provided type of the value.

Editor → Parameter → Create value 168

174 Interface as basis for the value specific instances.

Editor → Parameter → Value interface 169

175 Class which provides an interface to the value of the value specific instances.

Editor → Parameter → Value 169

176 Implementation of the float value type.

Editor → Parameter → FloatValue 170

177 Implementation of the scene value type.

Editor → Parameter → SceneValue 170

178 Implementation of the input of the definition of a node.

Editor → Node definition input 171

179 Constructor of the input of the definition of a node.

Editor → Node definition input → Constructor 171

180 Function that creates a default value function based on a provided value.

Editor → Node → Methods → Create default value function 172

181 A class method of the JSON module, which builds the output of a node definition from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition output 173

182 Implementation of the output of the definition of a node.

Editor → Node definition output 174

183 Constructor of the output of the definition of a node.

Editor → Node definition input → Constructor 174

184 A class method of the JSON module, which builds the definition of a node from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition 175

185 A method of the node controller, which returns a node definition by a provided identifier. If no node definition is found for the given identifier, a new node definition is created by loading the definition from the file system.

Editor → Node controller → Methods → Get node definition 176

186 The root node of the system is manually created by the node controller and is also a node definition.

Editor → Node controller → Constructor 177

187 Methods which add a given output of a node definition to a node definition. The first method adds the output at the end of the list of outputs, the second adds the output at the given index.

Editor → Node definition → Methods 178

188 Type property of a node definition. If the node definition uses outputs, the type is derived by its primary output. Otherwise a generic type is assumed.

Editor → Node definition → Methods 179

189 A class method of the JSON module, which builds a part of the definition of a node from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition part 180

190 Instantiation of the node controller from within the main application.

Editor → Main application → Constructor 181

191 Loading of nodes is triggered by the main application right after instantiating the node controller.

Editor → Main application → Constructor 181

192 Definition of a dialog to add nodes to the currently active scene. The nodes are ordered in columns according to their type.

Editor → Add node dialog 182

193 Class representing column within the dialog to create new node instances.

Editor → Add node dialog → Column 183

194 The dialog for adding new node instances is initialized by the scene view.

Editor → Scene view → Constructor 183

195 The event method of the scene view is overwritten for being able to show the dialog for adding new instances of nodes when the tabulator key is pressed.

Editor → Scene view → Methods 184

196 The signal of the node controller that is emitted whenever a node definition was read.

Editor → Node controller → Signals 184

197 The slot of the dialog to add a new node that is called whenever a new node definition is added.

Editor → Add node dialog → Slots 185

198 Implementation of the slot of the dialog to add a new node that is called whenever a new node definition is added.

Editor → Add node dialog → Slots 185

199 It is checked that the given node definition is not already known.

Editor → Add node dialog → Slots 185

200 The column which the node definition belongs to is depending on the node definition's type. If a column for the type of the node definition already exists, that column is used. Otherwise a new column is created for the type.

Editor → Add node dialog → Slots 185

201 The method for getting or creating a column by type name.

Editor → Add node dialog → Methods 186

202 If a column by the given name exists that column is returned.

Editor → Add node dialog → Methods 186

203 When no column for a specific name exists the column is created.

Editor → Add node dialog → Methods 187

204 A sub frame is created for each new column.

Editor → Add node dialog → Methods 187

205 For the type of the given node definition a button is created and added to the previously created sub frame.

Editor → Add node dialog → Methods 188

206 A callback function is created which gets triggered whenever a button is clicked. The callback function stores the selected node definition (type) and closes the dialog.

Editor → Add node dialog → Methods 188

207 The created sub frame is added to the layout of the column and appended to the list of sub frames of the column.

Editor → Add node dialog → Methods 188

208 The node definition is saved to the list of known node definitions.

Editor → Add node dialog → Slots 189

209 The main application connects the node controller's signal that a new node definition was added with the corresponding slot of the dialog to add nodes.

Editor → Main application → Constructor 189

List of Tables

Introduction

The application area of computer graphics exists since the beginning of modern computing. Computer scientists have always strived to create realistic depictions of the observable reality.

Over time various approaches for creating artificial images (the so called rendering) have evolved. One of these approaches is ray tracing. This was introduced in 1968 by A. Appel in the article "Some Techniques for Shading Machine Renderings of Solids" [1]. In 1980 this methods were improved by T. 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 of light. For a long time although, the approach was not efficient enough to deliver images in real time, which requires creating at least 25 rendered images (frames) per second. Otherwise, due to the human anatomy, the output is perceived as either still and jerky images or as a too slow animation.

Sphere tracing is a ray tracing approach introduced in 1994 by J. C. 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 its method of finding intersections between rays and objects.

Graphics processing units (GPUs) have evolved over time and have become exponentially 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, however, a clear disadvantage. The de facto way of representing the surface of objects using triangle based meshes cannot be used directly. Instead, distance functions are used for modeling the surfaces as seen from any view point.

Purpose and conditions

Motivation

UP 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 program which provides both, a node based approach for modeling and animating objects using signed distance functions and allowing the rendering of scenes using sphere tracing, efficiently enough to be executed in real time on the GPU.

Objectives and limitations

THE OBJECTIVE OF THIS THESIS is the design and development of a program for *modeling*, *composing* and *rendering* real time computer graphics by providing a graphical toolbox.

MODELING is done by composing simple objects into complex objects and scenes using a node based graph structure of "nodes".

COMPOSING includes two aspects: the combination of objects into scenes, and the creation of an animation which is defined by multiple scenes which follow a chronological order. The first aspect is realized by a scene graph structure. The second aspect is realized by a time line.

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 rendering algorithm used all objects are modeled using distance functions.

REQUIRED OBJECTIVES are the following:

- Development of an editor for creating and editing real time rendered scenes, containing the following features.
  - A scene tree, allowing management (creation and deletion) of scenes.

- One node-based graph structure for each scene in the scene tree. This allows the composition of scenes using nodes and connections between the nodes.
- Nodes
  * Simple objects defined by signed distance functions: In this thesis the objects are limited to cube and sphere (other solids could be modeled in the same way).
  * Simple operations: Union, Intersection, Subtraction.
  * Transformations: Rotate, Translate and Scale.
  * Camera.
  * Renderer (ray traced rendering using sphere tracing)
  * Point light sources.

Related works

PRELIMINARY to this thesis two project works were completed. The first was "Volume ray casting — basics & principles" [4], which describes the concepts of sphere tracing The second was "QDE — a visual animation system, architecture" [5], which established the concepts of an editor and a player component, as well as the basis for a possible software architecture for these components.

FOR CLARITY IN THE EXPLANATION of this thesis first the architecture is presented, see section "Software architecture". Then the rendering is presented in section "Rendering".

Document structure

THIS DOCUMENT IS DIVIDED INTO SIX CHAPTERS, the first being this chapter, "Introduction". The second chapter "Administrative aspects" shows the planning of the project, including the persons involved, the deliverables, and phases and milestones.

THE ADMINISTRATIVE ASPECTS ARE FOLLOWED BY the chapter "Fundamentals". The purpose of this chapter is to present the principles, based on the previous project works mentioned above that this thesis is built upon.

THE NEXT CHAPTER on "Methodologies" introduces a concept called literate programming and elaborates some details of the implementation using this. Additionally it introduces standards and principles for implementation of the developed software.

THE FOLLOWING CHAPTER on "Implementation" describes the editor component.

THE LAST CHAPTER "Discussion and conclusion" considers 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 follow appendices, see "Appendix", detailing the requirements for building the above-mentioned components, and the actual source code in form of a literal program.

Administrative aspects

The previous chapter provided an introduction to this thesis by outlining the purpose and conditions, the previous works and structure of this work.

This chapter covers administrative aspects of this project, although they are not required for an understanding of the results.

The first section defines the persons involved and their roles during this thesis. Afterwards the deliverable items are defined and described. The last section elaborates on the organization of the project work involved, including meetings, phases and milestones as well as the work schedule.

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 the system for composing real time computer graphics, developed during this project.
Presentation	The presentation of the defense of this thesis.

Table 3: List of deliverables.

Organization of work

Meetings

VARIOUS MEETINGS with the supervisor and the expert helped to reach the defined goals and to prevent erroneous directions during the project. The supervisor and the expert supported the author by providing suggestions throughout the work period.

Phases and milestones

Phase	Week / 2017
Start of the project	8
Definition of objectives and constraints	8-9
Documentation and development	8-30
Final review	30-31
Preparation for the defense of the thesis	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

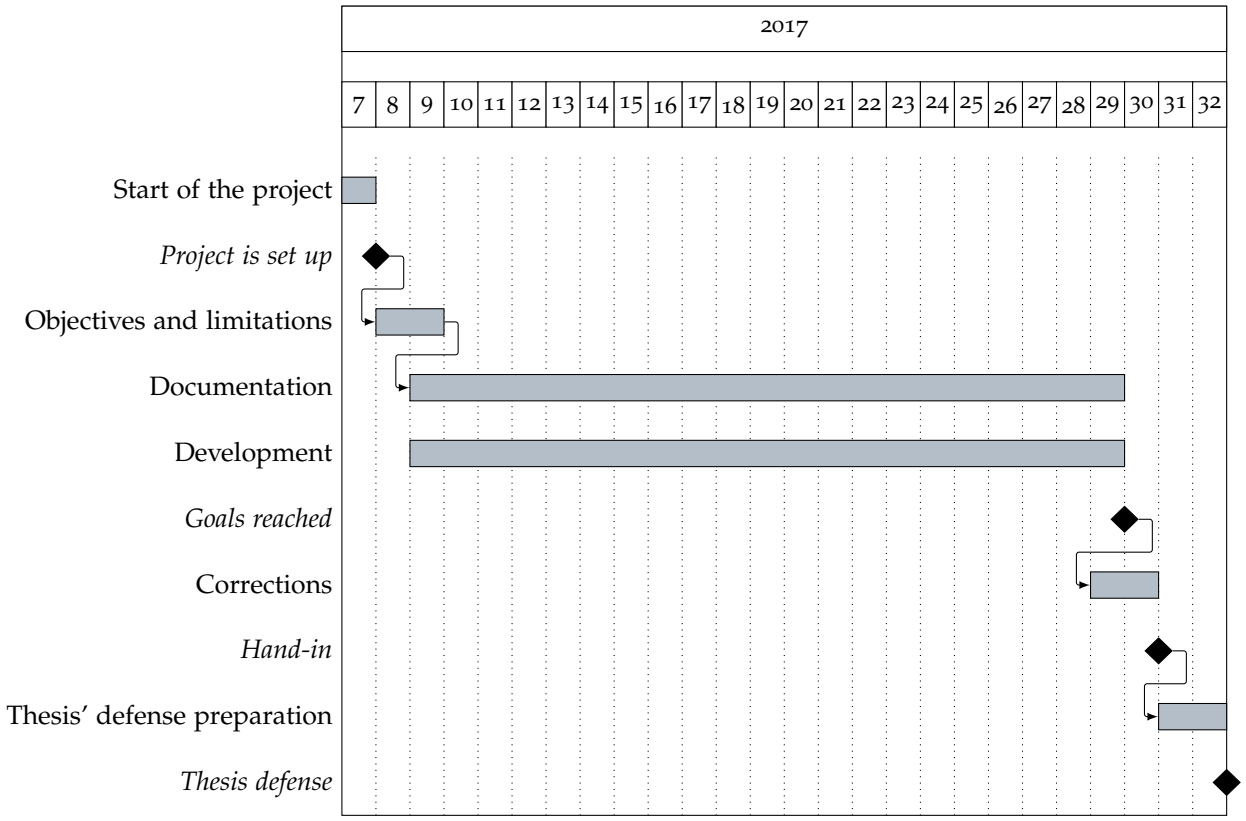


Figure 1: Schedule of the project. The subtitle displays calendar weeks.

Fundamentals

The previous chapter covered administrative aspects including the persons involved, phases and the schedule, and milestones of the project work.

This chapter presents the fundamentals which are required for an understanding of this thesis.

The first section of this chapter defines the software architecture that would be used for the implementation of the program implemented. 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 program implemented for this thesis.

Software architecture is inherent to software engineering and software development. It may be implicit, for example when developing a smaller program where the concepts are intuitively clear and the design decisions are self-explanatory. Unfortunately, sometimes momentary "self-explanatory" decisions are in retrospect deceptive, so that some documentation may be necessary. But the architecture may also be developed as an initial conceptual process, for instance when developing large and complex programs.

But what is software architecture? P. Kruchten [6] 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."

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

THE IDEA ENVISAGED FOR THIS THESIS of using a node based graph for modeling objects and scenes and rendering them using sphere tracing was developed in advance of this thesis. To ensure that this technical implementation was really feasible, a prototype was developed during the previous project work "Volume ray casting — basics & principles" [4].

THE ARCHITECTURE OF THIS PROTOTYPE had however evolved implicitly, and showed itself as hard to maintain and extend by providing no clear segregation between the data model and its representation.

WITH THE NEXT PROJECT WORK, "QDE — a visual animation system, architecture" [5], a software architecture was developed to prevent the occurrence of such problems. The software architecture is based on the rational unified process (RUP) [6] what leads to an iterative approach.

BASED ON THE VISION of this thesis and using the methodologies of RUP, actors are defined. The actors in turn are participants in use cases that define functional requirements for the behavior of a system. The definition of use cases shows the limitations of the program and define its functionality and therefore the requirements.

THE COMPONENTS, shown in Table 6 and Figure 2, are established based on these requirements.

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.

Table 6: Description of the components of the program implemented.

IDENTIFYING THE COMPONENTS helps finding the important conceptual items. Decomposing a domain into noteworthy concepts

Sub component	Description
Scene tree	Holds scenes in a tree like structure. A scene is a directed graph holding nodes.
Node graph	Contains all nodes which define a single scene.
Parameter	Holds the parameters of a node of the node graph.
Rendering	Renders a node, presenting it for a viewer.
Time line	Depicts temporal events in terms of scenes which follow a chronological order.

Table 7: Description of the sub components of the editor component.

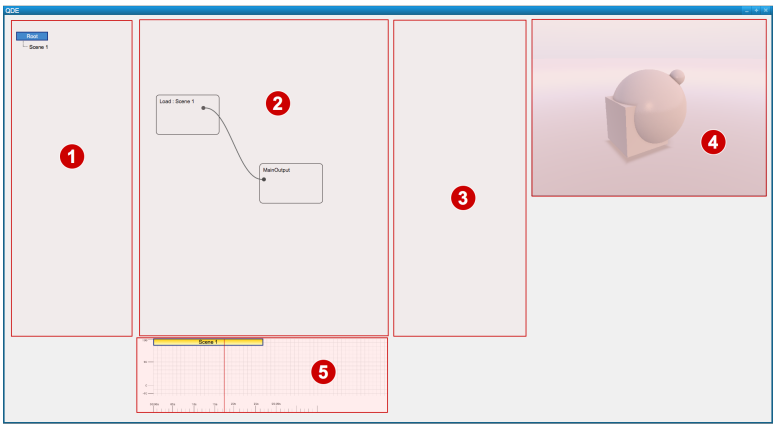


Figure 2: A mock up of the editor application showing its components.
1: Scene tree.
2: Node graph.
3: Parameter view.
4: Rendering view.
5: Time line.

is "the quintessential object-oriented analysis step" [8]. "The domain model is a visual representation of conceptual classes or real-situation objects in a domain." [8]

THE EDITOR AND PLAYER COMPONENTS are shown in Figure 3 and in Figure 4 respectively in the form of a domain model. Each domain model is composed of components which build the sub components of the parent component, in this case the editor and player. These sub components represent the objects of the respective domain.

IDENTIFYING THE IMPORTANT CONCEPTS allows the definition of the logical architecture and shows the overall image of (software) classes in form of packets, subsystems and layers. For a detailed definition of these items the reader is referred to the previous project work [4, pp. 37 ff.].

TO REDUCE DEPENDENCIES AND THE COUPLING of components a "relaxed layered" architecture is used. In contrast to a strict layered architecture, which allows any layer to call services or interfaces only from the layer below, a relaxed layered architecture allows higher layers to communicate with any lower layer. The architecture defines five layers, as shown in Table 8.

TO ENSURE LOW COUPLING AND DEPENDENCIES also for the graphical user interface, the models and their views are segregated using

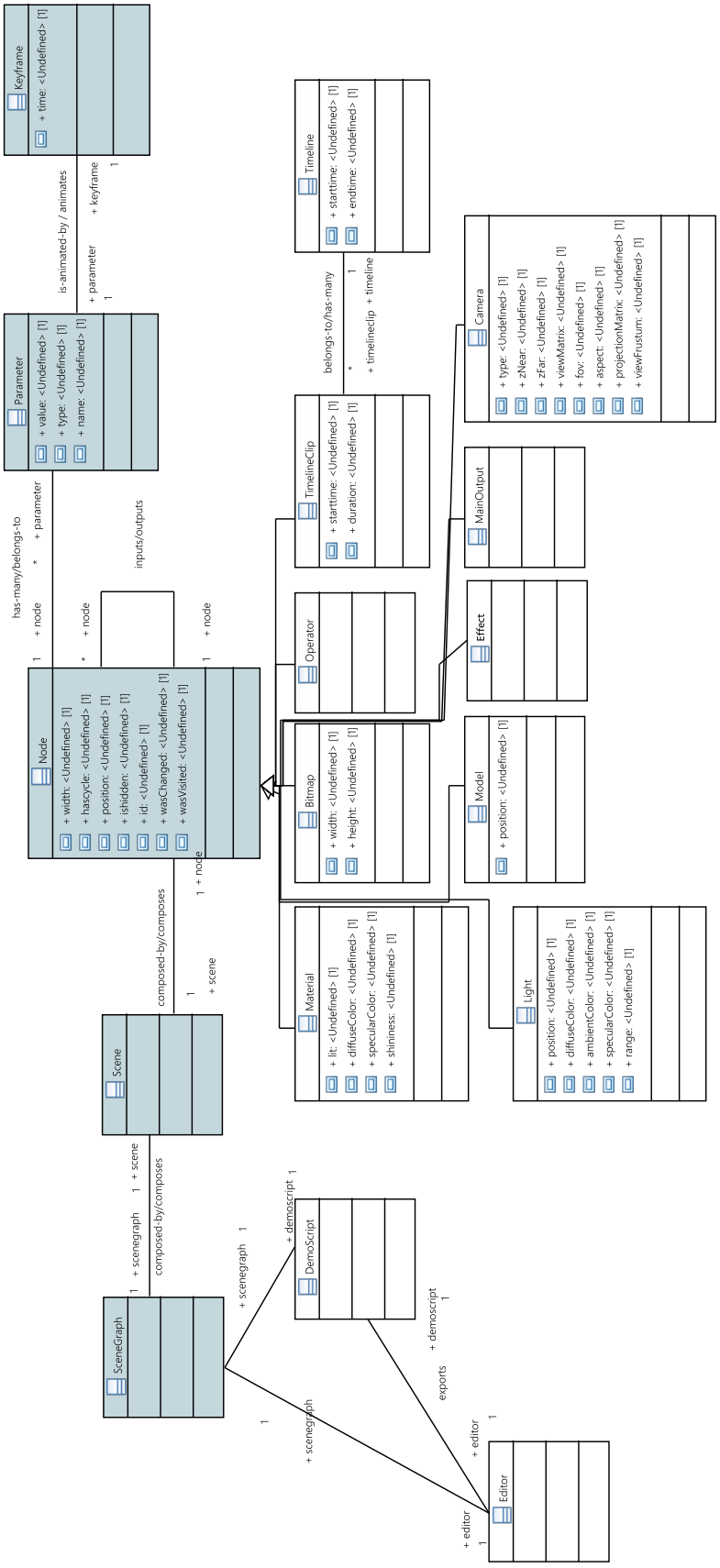


Figure 3: Domain model of the editor component.

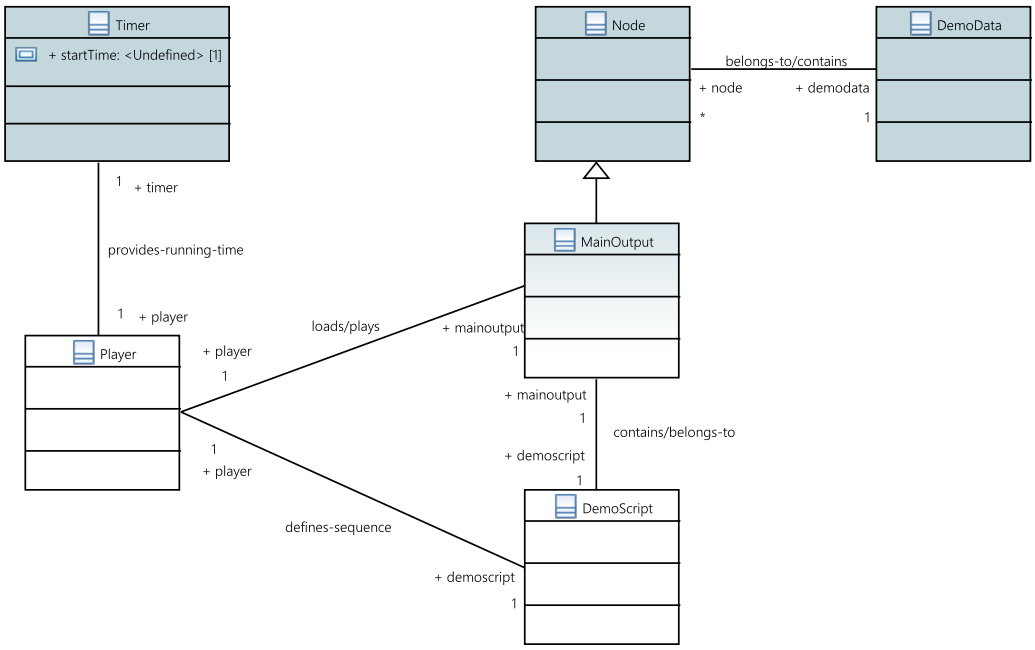


Figure 4: Domain model of the player component.

the principle of model-view separation which states that domain objects, which are data models, should have no direct knowledge about their corresponding objects of the graphical user interface.

WORKFLOW OBJECTS control the user interaction with the visual objects by keeping track of the data models. Such *controllers* support the model-view separation principle and exist in the application layer.

Layer	**Description**
UI	All elements of the graphical user interface.
Application	Controllers (workflow objects).
Domain	Data models according to the 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 timer, arrays or other data classes.

Table 8: Layers as envisaged during the conceptual phase and used for the program implemented.

CLASS DIAGRAMS ARE USED TO PROVIDE A SOFTWARE POINT OF VIEW, whereas domain models provide rather a conceptual point of view. A class diagram [1] shows classes, interfaces and their relationships. Figure 5 shows the class diagram of the editor component and Figure 6 shows the class diagram for the player component.

[1] The concepts of RUP and OOP are as used in the previous project (QDE) and therefore will not be detailed here.

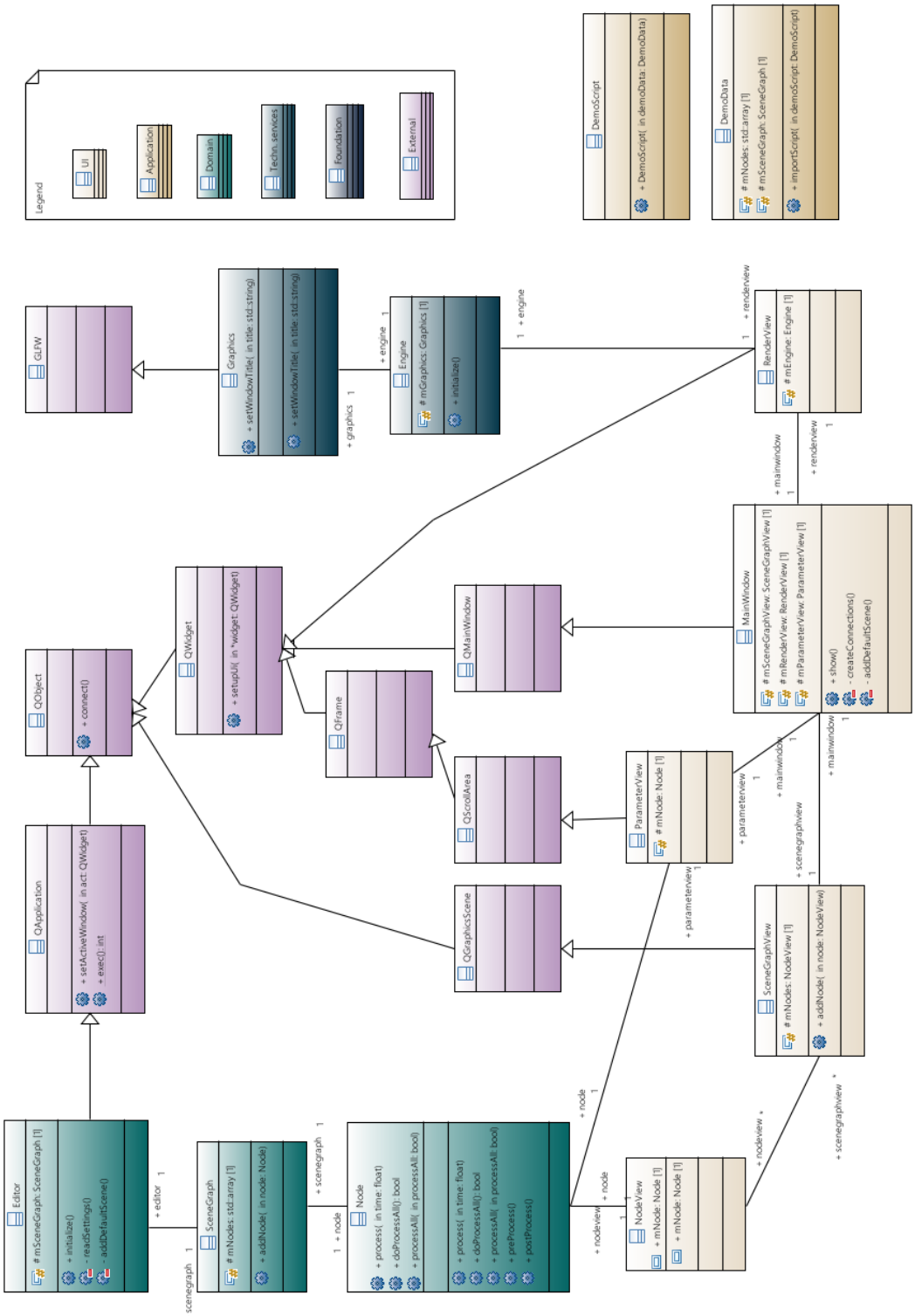


Figure 5: Class diagram of the editor component.

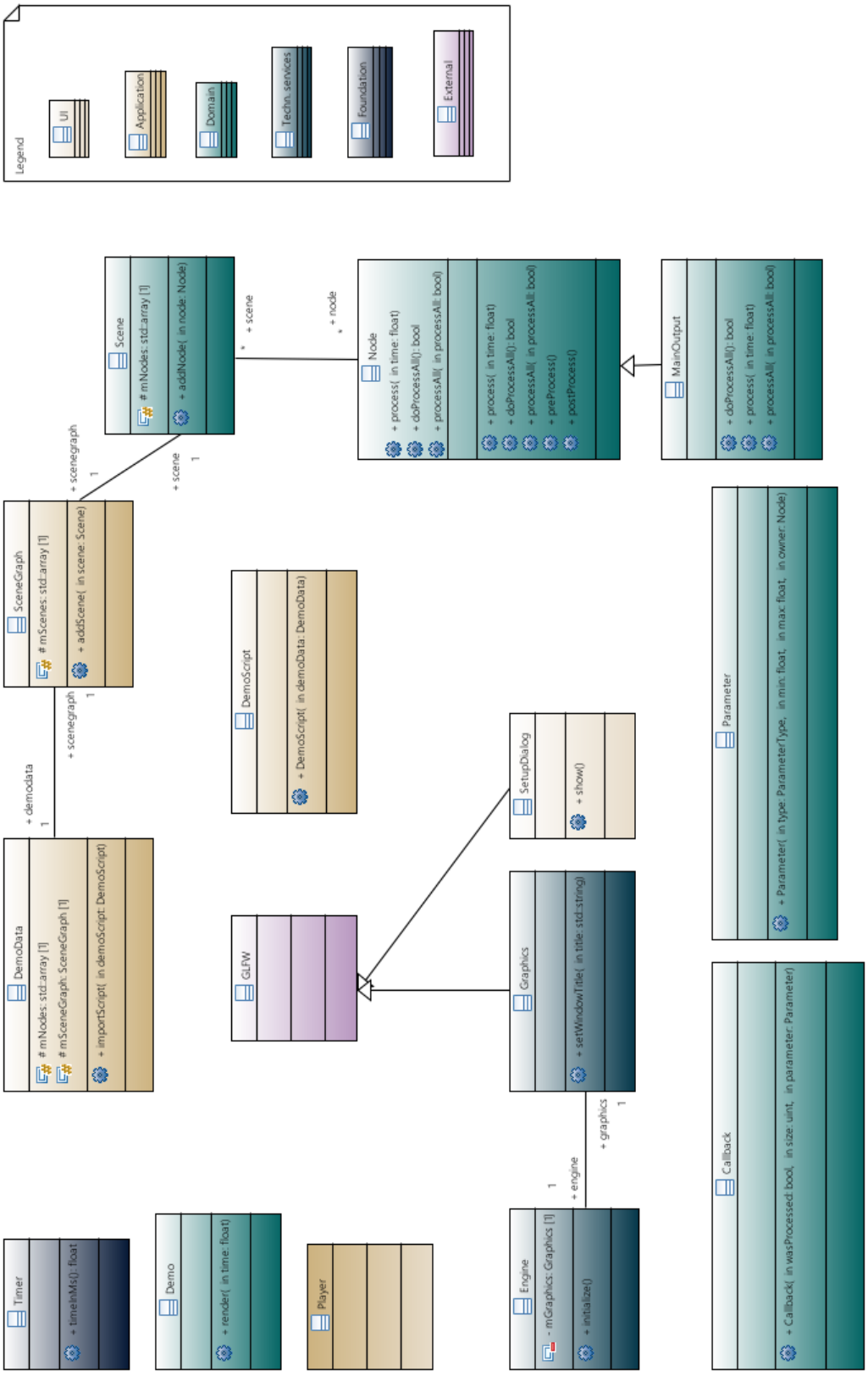


Figure 6: Class diagram of the player component.

Rendering

THIS SECTION is a summary of the 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 program implemented in this thesis.

RENDERING is the second main aspect of this thesis, as the objective is the design and development of a program for modeling, composing and *rendering* real time computer graphics by providing a graphical toolbox.

J. FOLEY DESCRIBES RENDERING as a "process of creating images from models" [9]. The basic idea of rendering is to determine the color of a surface at each point. For this task two concepts have evolved: *illumination models* and *shading models*.

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 from light sources, global illumination models consider also indirect light from reflections and refractions. The algorithm used for rendering in the implemented program uses a *global illumination model*.

SHADING MODELS define when and how to use which illumination model.

GLOBAL ILLUMINATION MODELS "express the light being transferred from one point to another in terms of the intensity of the light emitted" [9, pp. 775 and 776]. Additionally to this direct intensity, the indirect intensity is considered, meaning "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.

In 1986 James Kajiya set up the so called rendering equation, which expresses this behavior. [10, 9, p. 776] The rendering equation is shown in Figure 7.

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

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 scaling factor:

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

$\varepsilon(x, x')$ 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 near 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. Where S_i is the surface of object i, and so 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 (computational feasible in a workflow for instance for the production of animated films) or even real time is not really feasible, even on the fastest modern hardware (as of 2017). The procedure is computationally complex and very time demanding.

A simplified approach to implement global illumination models (and 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 viable for producing still images. For producing images in real time however, the algorithm 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 J. C. 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 its method of finding intersections between rays and objects. In contrast to the classical ray tracing approaches, the "marching distance" of rays is not defined by an absolute or a relative distance, but 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.

Bounding volumes are defined as the enclosure of a solid. On the other hand, unbounding volumes are the space outside of a bounding volume, including the surface of the solid itself. For calculating a unbounding volume, the distance between an object and any defined origin point is evaluated. If this distance is known, it can be taken as the radius of a sphere centered at the origin point.

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 can be calculated. These distances build a so called distance field.

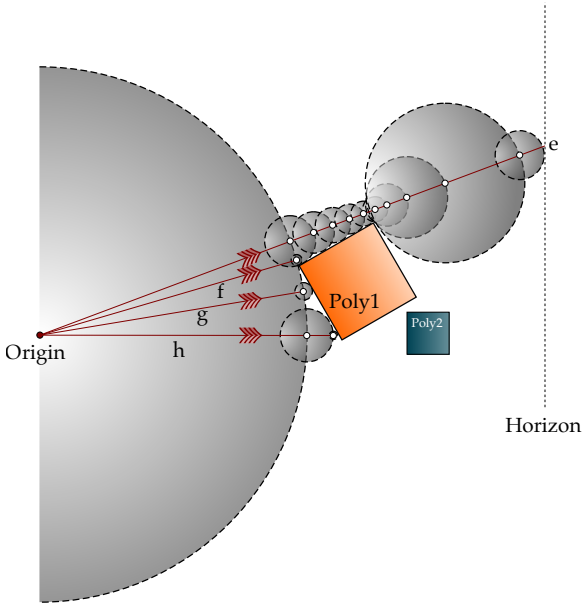


Figure 8: Illustration of the sphere tracing algorithm. Ray *e* hits no objects until reaching the horizon. Rays *f*, *g* and *h* hit the solid *poly1*.

THE SPHERE TRACING ALGORITHM is as follows. A ray is shot from an origin point (for example, the viewer such as an eye or a pinhole camera) into a scene. The radius of an unbounding volume in form of a sphere is calculated from the origin, as described above. The ray intersects with the sphere, which gives the distance that the ray will travel in a first step. From this intersection the next unbounding volume (sphere) is expanded and its radius is calculated, which gives the next intersection of the ray. This procedure continues until an object is hit or until a predefined maximum distance of the ray is being reached, defined as the "horizon". An object is considered as "hit" whenever the returned radius of the distance function is below a predefined constant ϵ, the "convergence precision".

A POSSIBLE IMPLEMENTATION of the sphere tracing algorithm is shown in Figure 9, although this shows only the distance estimation. Shading is done in another implementation, for example in a render method which calls the sphere trace method. Shading means in this context the determination of the color of a surface or pixel.

SHADING is done as proposed by T. Whitted in "An Improved Illumination Model for Shaded Display" [2]. This means, that the sphere tracing algorithm must return which object was hit and its material. Depending on the material, four cases can occur. The material may be: (1) reflective and refractive, (2) reflective only, (3) diffuse or (4) emissive. For simplicity only the third case is being taken into account. For the actual shading a local illumination method is used, called *Phong shading*.

```
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]

THE PHONG ILLUMINATION MODEL [9, p. 123] describes (reflected) light intensity I as a composition of the ambient, the diffuse and the perfect specular reflection of a surface. Figure 10 shows the Phong illumination model.

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

Figure 10: The Phong illumination model as defined by Phong Bui-Tuong. [9, p. 123] Note that the emissive term was left out intentionally as it is mainly used to achieve special effects.

$I(x, x')$ Intensity of the light at point $\vec{V}$.

k_a A constant defining the ratio of reflection of the ambient term of all points in the scene.

L_a Intensity of the ambient light.

k_d A constant defining the ratio of the diffuse term of incoming light.

$\sum_{i=0}^{n-1}$ Sum over all light sources in the scene.

L_i Intensity of the i-th light source.

$\vec{S_i}$ Direction vector from the point on the surface toward light source i.

$\vec{N}$ Normal vector at the point of the surface.

$\vec{R_i}$ Direction that a perfectly reflected ray of light would take from this point on the surface.

$\vec{V}$ Direction pointing toward the viewer.

k_e Shininess constant for the material.

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

Documenting software is critical. Whenever software is written, decisions are made. In the moment a decision is made, it may seem intuitively clear, as it has evolved through creative thought processes. The seeming clarity of the decision is most of the time deceptive. Is a decision still clear when some time has passed since making that decision? What were the considerations that led to it? Is the decision also clear for other, may be less-involved persons? All these concerns show that documenting software is critical. No documentation at all, or outdated or irrelevant documentation, can lead to unforeseen and costly efforts concerning work and time.

C. A. R. 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 D. E. Knuth, goes even further. D. E. Knuth believes that "significantly better documentation of programs" can be best achieved "by considering programs to be works of literature" [12, p. 1]. D. E. 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 should 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 D. E. 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 parts) and generates two outputs: (1) a document in a formatting language, such as Knuth's LaTeX [13] (which may then be converted in a printable and viewable form, such as PDF) and (2) a compilable program in a programming language, such as Python or C (which may then be converted into an executable program). [12] The first process is called *weaving* and the second *tangling*. This process is illustrated in fig. 11.

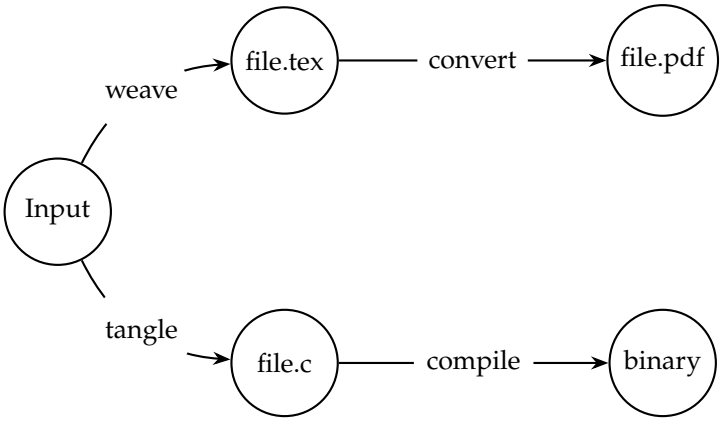


Figure 11: Illustration showing the processes of *weaving* and *tangling* documents from an input document. [12]

SEVERAL LITERATE PROGRAMMING (LP) SYSTEMS WERE EVALUATED during the first phase of this thesis: CWEB [2], Noweb [3], lit [4],

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

PyLiterate [5], pyWeb [6] and Babel [7]. All of these tools have their strengths and weaknesses. However, none of these systems fulfill all the needed requirements of this project: (1) Provide "pretty printing" [8] 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. [8]

[5] https://github.com/bslatkin/pyliterate

[6] http://pywebtool.sourceforge.net/

[7] http://orgmode.org/worg/org-contrib/babel/

[8] pretty printing refers to content-based formatting (e.g. line color and indentation to improve readability).

Ultimately a further literate programming system, nuweb [9], was chosen as it fulfills all these requirements. It has adapted and simplified the ideas of FunnelWeb [10]. 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 is not designed for the pretty printing of source code, it does provide an option to display source code as listings. This facility has been modified to support visualizing the expansion of parts as well as to use syntax highlighting of the code within LaTeX.

[9] http://nuweb.sourceforge.net/

[10] http://www.ross.net/funnelweb/

The nuweb system provides several commands to process files. All commands begin with an at sign (@). Whenever a file or part does not contain any commands the file is copied unprocessed. nuweb provides a single executable program, which processes the input files and generates the output files (weaving and tangling, in document formatting language and as source code respectively).

A fragment consists of scraps which in this project contain the source code. They may also contain for instance paragraphs for formatted text or mathematical equations.

Literate programming can be very expressive when all concepts are explicitly defined before implementation. D. E. Knuth sees this expressiveness an advantage as one is forced to clarify thoughts before programming [12, p. 13]. This is surely very true for small software but only partly true for larger software. The problem with larger software is, that when using literate programming, the documentation tends to be correspondingly large. *To overcome this problem* in this project, the actual implementation of the software is placed into to the appendix, see appendix "Code fragments".

Another problematic aspect is the implementation of repeating fragments or parts with similar but not identical technical details (such as imports or getter and setter methods). This might be interesting only for software developers or technically oriented readers who want to grasp all the details. *This aspect can be overcome* by moving recurring or uninteresting fragments to a separate file (see appendix "Code fragments").

To show the principles of literate programming, without annoying the reader, only an excerpt of some details is given here.

One of the more interesting things of the software might be the definition of a node and its loading from external files. These two aspects are shown below. More details of this example would go beyond the scope of this thesis.

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 four parts:

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

(2) A constructor — creator of an initial instance of a class.

(3) Various methods — actions which can be executed by the object.

(4) Slots — receive signals from other objects.

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

Implementing the class called "NodeDefinition" means simply defining a scrap called "*Node definition declaration*" using the above structure. The scrap does not have any content at the moment, except references to other scraps, that build the body of the scrap. These will be defined later on. Figure 12 shows the scrap.

⟨ *Node definition declaration* 49 ⟩ ≡

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

    # Signals

    ⟨ Node definition constructor  50⟩

    # Methods

    # Slots
◇
```

Figure 12: Declaration of the node definition class.

Fragment never referenced.

The constructor might be the first thing to implement. In Python the constructor defines properties of a class [11]. In context of the program, one might come up with the properties in Table 9 defining a node definition.

[11] 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.

Implementing the constructor of the node definition may now follow from the properties defined in Table 9. Figure 13 shows the scrap containing the definition of the constructor.

One of the problems mentioned before can be seen in fig. 13: it shows a rather simple constructor without any useful logic. Additional modules would be needed, e.g. Qt bindings for Python or system modules. At this point the implementation of node definitions will not be detailed further, as this is beyond the scope of this thesis.

The following pages demonstrate the use of literate programming to document and manage the code of the loading of node definitions.

Node definitions will be loaded from external files in JSON format. This happens within the node controller component (not shown here). 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.

Property	Description
ID	A global unique identifier.
Name	The name of the node, e.g. "Sphere".
Description	A description of the node's purpose.
Inputs	Parameters given to the node. These may have distinct types, e.g. scalars as floating point numbers or character strings of type text, references to other nodes.
Outputs	Values delivered by the node.
Definitions	A list of the node's definitions. This may be an actual definition of a (shader-) function in terms of an implicit surface.
Invocation	The format of a call to the node definition, including placeholders which will be replaced by parameters.
Parts	Defines text that may be processed when calling the node. Contains code which can be interpreted directly.
Nodes	The children a node has (child nodes). These entries are references to other nodes only.
Parameters	A list of the node's inputs and outputs in form of tuples. Each tuple is composed of two parts: (1) a reference to another node and (2) a reference to an input parameter or an output value of that node. If the first reference is not set, this means that the parameter is internal.

Table 9: Properties/attributes of a node definition.

⟨ *Node definition constructor* 50 ⟩ ≡

```
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 49.

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.

⟨ *Load node definitions* 51a ⟩ ≡

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

Figure 14: Head of the method that loads node definitions from external JSON files. Words written in uppercase are class constants.

Fragment defined by 51ab.
Fragment never referenced.

When loading node definitions, there are two cases (and consequences) at the first instance: (1) the directory containing the node definitions exists, so the node definitions may be loaded or (2) the directory does not exit. In the first case the directory possibly containing node definitions is being searched for such files, in the second case a warning message is logged. This is shown in fig. 15.

⟨ *Load node definitions* 51b ⟩+ ≡

```
if os.path.exists(self.nodes_path):
    ⟨ Find and load node definition files 52a, ... ⟩
else:
    ⟨ Output warning when directory with node definitions does not exist 54a ⟩◇
```

Fragment defined by 51ab.
Fragment never referenced.

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

In the first case, when the directory containing the node definitions exists, files containing node definitions are searched. The files are searched by wildcard pattern matching the extension: `*.node`. This can be seen in fig. 16.

In the second case, a warning or error message must be generated. This is detailed at the end of this section.

Having searched for node definition files, there are again two cases, similar as before: (1) files (possibly) containing node definitions exist within the source directory, or (2) no files with the ending `.node` exist. Again, as before, in the first case the node definitions will be loaded, in the second case a warning message will be logged, which is shown in fig. 17.

In the case (1) when node definitions are present, they are loaded from the file system, parsed and then stored in an object

⟨ *Find and load node definition files* 52a ⟩ ≡

```
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 52ab.
Fragment referenced in 51b.

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

⟨ *Find and load node definition files* 52b ⟩+ ≡

```
if num_node_definitions > 0:
    ⟨ Load found node definitions 52c, ... ⟩
else:
    ⟨ Output warning when no node definitions are found 54b ⟩◇
```

Fragment defined by 52ab.
Fragment referenced in 51b.

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

in memory. To maintain readability, these text parts (fragments or scraps) are is encapsulated in a method, `load_node_definition_from_file_name`, which is not shown here. If the node definition cannot be loaded or parsed a null object (Python `None`) is being returned. This can be seen in fig. 18.

⟨ *Load found node definitions* 52c ⟩ ≡

```
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 52c, 53.
Fragment referenced in 52b.

Figure 18: Loading and parsing of the node definitions found within the folder containing (possibly) node definition files.

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. This process is explained in detail in the next chapter.

⟨ *Load found node definitions* 53 ⟩+ ≡

```
    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 162 ⟩

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

Fragment defined by 52c, 53.
Fragment referenced in 52b.

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.

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

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

⟨ *Output warning when directory with node definitions does not exist* 54a ⟩ ≡

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

Fragment referenced in 51b.

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

or

(2) no files containing node definitions are found.

⟨ *Output warning when no node definitions are found* 54b ⟩ ≡

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

Fragment referenced in 52b.

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

Agile software development

Software engineering always invokes 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 of methodologies. For medium to large projects however, using certain methodologies or principles is necessary to ensure (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]

Examples of risks of software development [16] are: incomplete and misunderstood needs, delays and schedule slippage, increased error rate, new changes requested, superfluity of features, etc. [16]

Traditional software engineering methodologies, such as the waterfall model or incremental development, struggle with the management of change. In case of the waterfall model changes are deprecated or lead to followup projects. In the case of incremental development, the phases can be very long, which allows only slow reaction to change requests. Figure 22 and Figure 23 illustrate these methodologies.

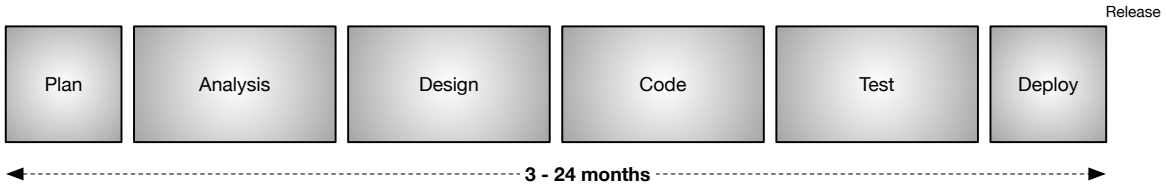


Figure 22: Phases of the water fall methodology. [17, p. 16]

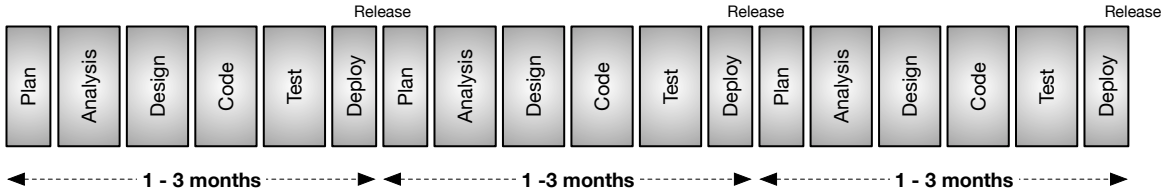


Figure 23: Phases of iterative development. [17, p. 16]

By applying some alternative basic principles, agile development methodologies try to overcome these problems. These principles may vary depending on the used methodology, but the fundamental ones are: (1) rapid feedback, (2) minimize complexity, (3) incremental change, (4) no fear of changes and (5) high 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 at the end of the work of the previous project ("QDE — a visual animation system, architecture" [5]), some needs were still continuously changing or even not yet well-defined. An exact planning, analysis and design, as traditional methodologies require it, would not be very practical. The following Figure 24 illustrates the use of an agile methodology.

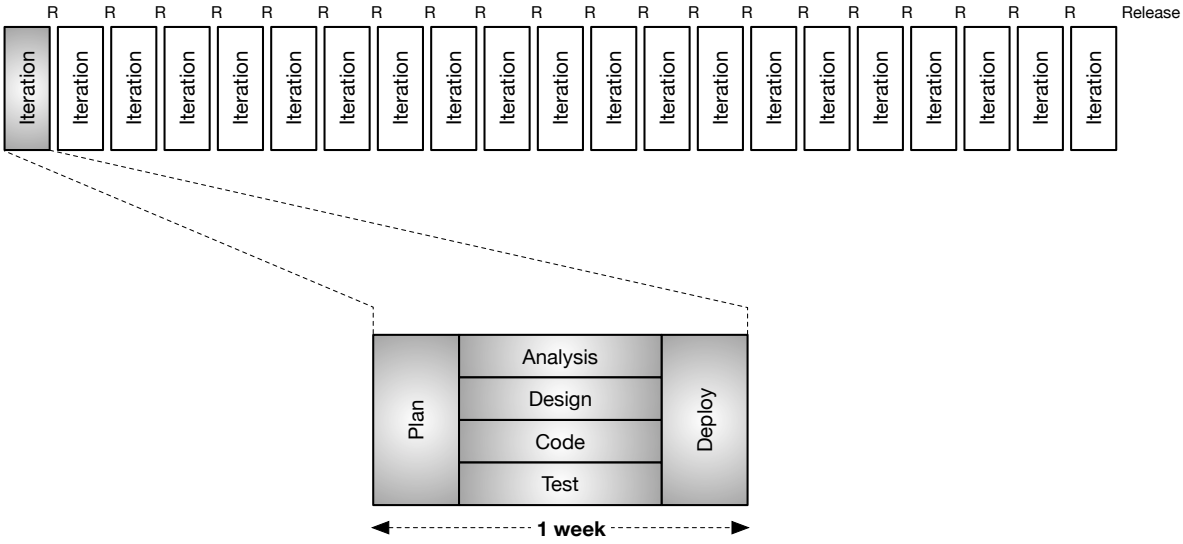


Figure 24: Iterations in the extreme programming methodology and phases of an interation. [17, p. 18]

Implementation

The previous chapter introduced the methodologies that are required for understanding the results of this thesis.

This chapter contours three sections. The first section shows the software architecture that was developed and that is used for the program implemented. Aspects of the literate form of this program implemented are shown in the second section. The main concepts and the components of the program are shown in the third section.

Software architecture

The software architecture defines the significant decisions of the program implemented, such as the selection of structural elements, their behavior and their interfaces. [6] The architecture is derived from the experiences based on the former projects, "Volume ray casting — basics & principles" [4] and "QDE — a visual animation system, architecture" [5], which build the fundamentals of this thesis, see chapter "Fundamentals".

Three aspects define the software architecture:

(1) an architectural software design pattern,

(2) layers and

(3) signals and slots, which allow 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." [18, p. 16]

To separate data from its representation and to ensure a coherent design, a combination of the model-view-controller (MVC) [19] and the model-view-view model pattern (MVVM) [20], [21] is used

as architectural software design pattern. This decision is based on experiences from the previous projects and allows individual parts to be modified and reused. 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 pattern used. Table 10 provides a description of the components. Figure 25 shows an overview of the components of the editor (the colored items) including their communication in an informal way. Additionally the user as well as the display is shown (in gray color). As the player component only plays animations, no view models are needed and therefore only the MVC pattern is used, which is shown in fig. 26.

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 tree view, Scene view
View model	"Model of a view", the abstraction of the view, provides a specialization of the model which the view can use for data-binding. It also stores the state and may provide complex operations.	Scene tree 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 tree controller, scene controller, node controller

Table 10: Description of the types of components of the software design pattern used. [20], [21]

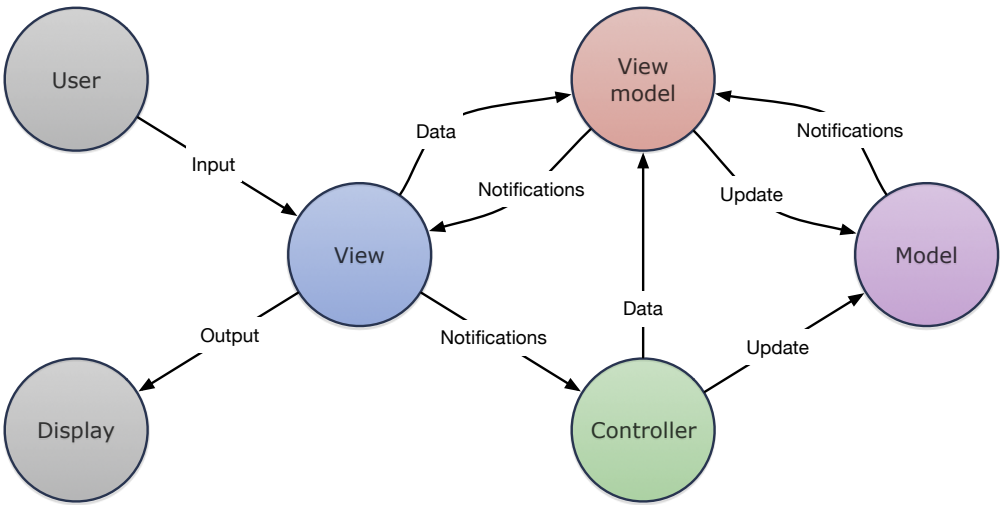


Figure 25: Components of the used pattern for the editor and their communication.

THE QT FRAMEWORK which is used, offers a very similar pattern

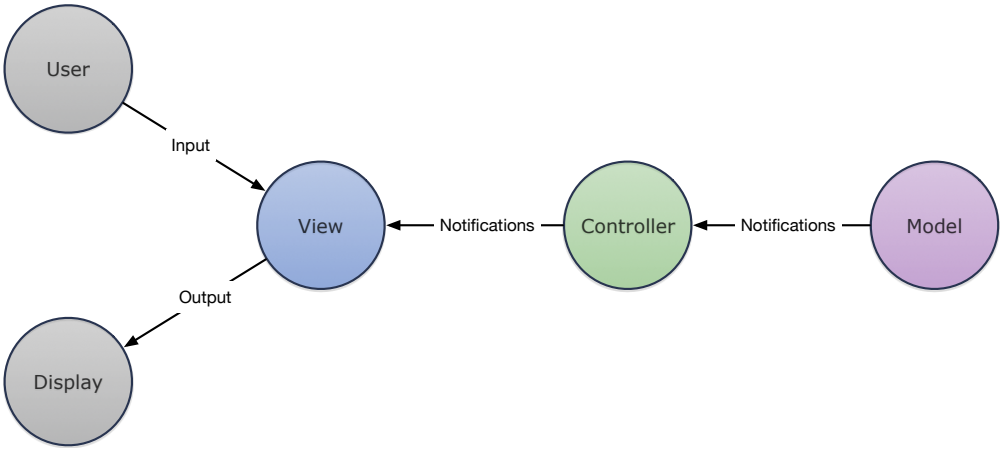


Figure 26: Components of the used pattern for the player and their communication.

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 indexes, which are references to items of data. [22] "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." [22] Figure 27 shows this model/view pattern.

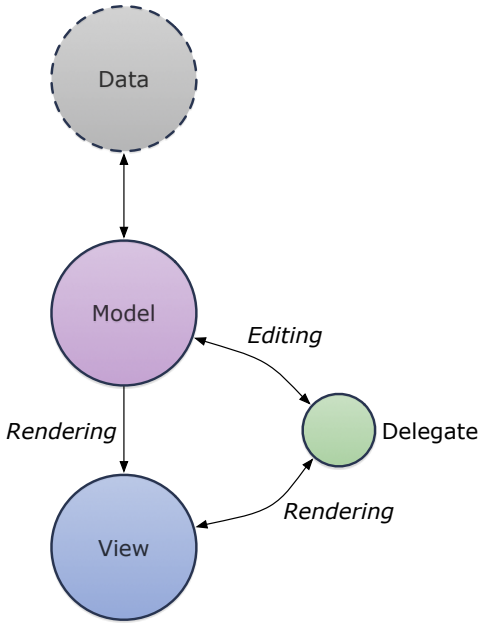


Figure 27: Qt's model/view pattern. [22]

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 this project. This is mainly due to two reasons: (1) the developed and intended components use no data source other than external files and (2) the concept of using model indexes may add flexibility but also introduces overhead.

THE SCENE TREE COMPONENT of the editor was developed using the Qt class for the abstract item model, which uses the model/view pattern. This showed that the usage of this pattern introduces unnecessary overhead and requires more effort to implement, while not using the advantages of features of the pattern. Therefore a decision was taken against the usage of the pattern in this case.

Layers

TO REDUCE COUPLING AND DEPENDENCIES a relaxed layered architecture is used, as written in section "Software architecture". In contrast to a strict layered architecture, which allows any layer to call only services or interfaces from the layer below, the relaxed layered architecture allows higher layers to communicate with any lower layer. Table 11 provides a graphical overview as well as a description of the layers. The colors have no significance except to distinguish the layers visually for the reader.

Layer	Description	Examples
Graphical user interface (GUI)	All elements of the graphical user interface, views.	Scene tree view, scene view, render view
Graphical user interface domain (GUI domain)	View models.	Scene tree view model, node view model
Application	Controller (workflow objects).	Main application, scene tree controller, scene controller, node controller
Domain	Data models according to the logic of the application.	Scene model, parameter model, node definition model, node domain model
Technical services	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 timer, arrays or other data classes.	Colors, common, constants, flags

Table 11: Layers of the program implemented.

Coupling and cohesion, signals and slots

WHENEVER DESIGNING AND DEVELOPING software, coupling and cohesion can occur and may pose a problem if not considered early and well enough.

COUPLING measures how strongly a component is connected to other components, or has knowledge of them or depends on them. High coupling impedes the readability and maintainability of software, so programmers should strive towards low coupling. C. Larman states that the principle of low coupling applies to many dimensions of software development and that it is a major objective 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, it is "a measure of the strength of association of the elements within a module". [23, p. 52] Low (or poor) cohesion does not imply that a component works only by itself, indeed it probably collaborates with many other objects. But low cohesion tends to create high (poor) coupling. It is therefore desirable 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 informally in fig. 28 and fig. 32.

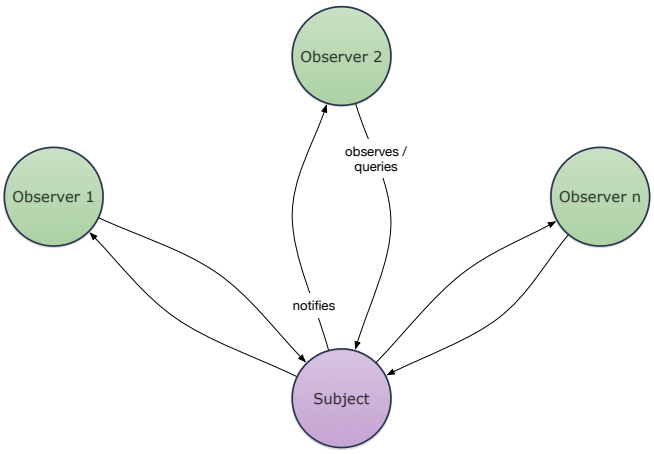


Figure 28: The observer pattern. [18]

A SIGNAL IS AN OBSERVABLE EVENT. A slot is a potential observer, typically a method of an object. An example of an observer is given in fig. 29.

SLOTS ARE REGISTERED AS OBSERVERS to signals as shown in fig. 30.

```
class Observer:

    @slot(type)
    def on_signal(self, parameter):
        """Listens on the signal 'signal'. Expects the
        signal to send the parameter 'parameter' of
        type 'type'."""

        ...
```

Figure 29: An example of a class called "Observer" with a slot called "on_signal". The slot expects the signal to send a parameter of type "type".

```
subject.signal.connect(
    observer.on_signal
)
```

Figure 30: The "on_signal" slot (which is a method of the object "observer") is registered to the signal "signal" of the object "subject".

WHENEVER A SIGNAL IS EMITTED, the emitting class must call all the registered observers for that signal as shown in fig. 31.

```
subject.signal.emit(some_parameter)
```

Figure 31: The signal "signal" of the object "subject" is emitted. The signal contains a parameter called "some_parameter". This means that the emitting class, "Signal", will call the registered method "on_signal" of the observer called "observer".

THE RELATIONSHIP BETWEEN SIGNALS AND SLOTS is 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. A relationship between a signal and a slot is shown in fig. 32

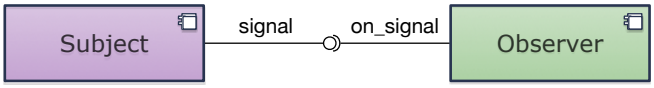


Figure 32: An observer is listening to a signal sent by a subject. The subject emits the signal and calls then the observers that are registered for that signal (or the registered slots of the observers respectively).

SIGNALS CAN CARRY ADDITIONAL INFORMATION, such as single values or even references to objects. A simple example of a signal-slot relationship is loading node definitions from files. The node controller, when it loads node definitions, could emit two signals to inform other components. For example, signal 1 (1) gives the number of node definitions to load, and signal (2) the index of the last loaded node definition and a reference to it. This information could for example be used in a dialog showing the progress of loading from the file system.

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

Figure 33: 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 34: An example of emitting a signal including a value and a reference to an object.

Literate programming

DOCUMENTATION IS CRUCIAL TO THE MAINTENANCE OR MODIFICATION of any software project. However, all too frequently the documentation is not done properly or is even neglected because of seemingly low benefit related to effort. No documentation at all, outdated or irrelevant documentation can cause unforeseen cost and time overruns. Using the literate programming paradigm prevents these problems, as the software as well as the documentation are derived from a literate program. For this thesis the LP system nuweb was used, as described under section "Literate programming".

USING LITERATE PROGRAMMING TO DEVELOP SOFTWARE requires a different way of thinking from traditional methodologies. 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 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 compilation or running 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 becomes lengthy and hard to read, especially when including the full implementation of technical details.

THESE PROBLEMS in the writing of this thesis were overcome by moving the implementation into the appendix, see Part "Appendix" and by outsourcing technical parts into a separate file, see Appendix "Code

fragments".

Program

To recall, the objective of this thesis is the design and development of a program for modeling, composing and rendering real-time computer graphics by providing a graphical toolbox.

Using the introduced methodologies (see chapter "Methodologies") and the developed software architecture (see section "Software architecture") a program was implemented.

The program implemented should to have two main components: the *editor* and *player*.

The editor component provides a graphical system for modeling, composing and rendering of scenes. It allows composing scenes into an animation and saving the animation in an external file. Rendering is done using the shown sphere tracing algorithm combined with Phong shading.

The player component simply plays an animation which has been created with the editor component. This includes loading and rendering of all scenes.

Due to time constraints, however, only the editor component was implemented. Figure 35 shows an image of the program implemented.

For the implementation the following tools were used: the Python programming language [12], the Qt cross-platform application development framework [13], the PyQt5 bindings [14] for Qt and OpenGL [15].

[12] version 3.5.2, `http://www.python.org`

[13] version 5.7, `https://www.qt.io/`

[14] version 5.7, `https://riverbankcomputing.com/software/pyqt/intro`

[15] version 3.3, `https://www.opengl.org/`

The quintessence of both components is to output respectively to read a data structure in the JSON [24] format which defines an animation. This data structure provides an animation which contains scenes, which contain nodes. This data structure is evaluated and the result is shader-specific code which is executed by the graphical processing unit (GPU) and seen by the viewer.

An animation is simply a composition of scenes which run in a sequential order within a defined time span.

A scene is a composition of nodes stored in the form of a directed graph.

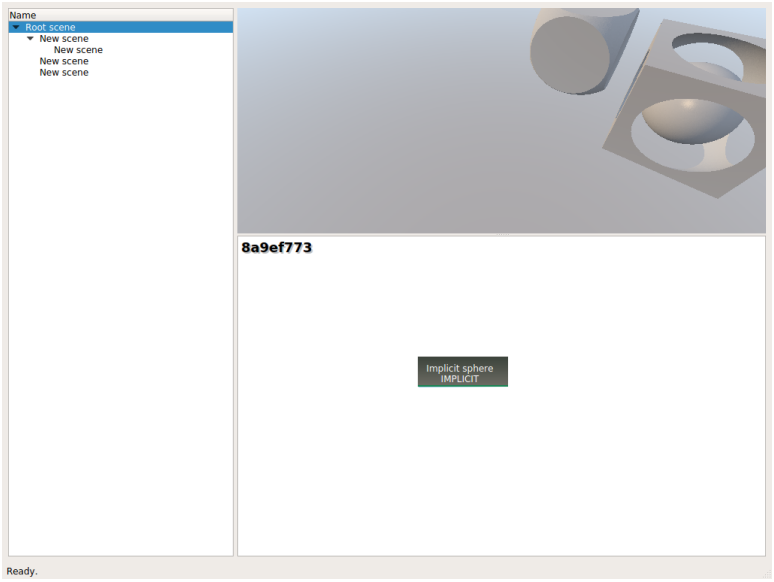


Figure 35: The implemented editor component.

NODES are instances of node definitions and define the content of a scene and therefore of an animation.

NODE DEFINITIONS provide content in a specific structure, shown in Table 12.

CONTENT is whatever a node definition provides in terms of the definitions but the output has always to be an atomic type as defined in table 13.

AN EXAMPLE of an node definition of type *implicit* for rendering a sphere is given in subsection "Node graph".

SUBSEQUENT EACH COMPONENT OF THE EDITOR is shown in a component diagram in adapted form [8, pp. 653 – 654] and an entity relationship diagram [8, pp. 501 ff.] (if appropriate), followed by a description of the component. The component diagram is used to show the signals that a component emits and receives. The entity relationship diagram is used to show the relationships between components. Only the relations immediately related to the presented component are shown, because the diagrams would otherwise be too crowded and confusing.

TO PRESERVE CLARITY all components are described in discrete sections of this chapter. Although the implementation of the components is very specific, in terms of the programming language, their logic may be reused later for the player component.

Property	Description
ID	A global unique identifier.
Name	The name of the node, e.g. "Sphere".
Description	A description of the node's purpose.
Inputs	Parameters given to the node. These may have distinct types, e.g. scalars as floating point numbers or character strings of type text, references to other nodes.
Outputs	Values delivered by the node.
Definitions	A list of the node's definitions. This may be an actual definition of a (shader-) function in terms of an implicit surface.
Invocation	The format of a call to the node definition, including placeholders which will be replaced by parameters.
Parts	Defines text that may be processed when calling the node. Contains code which can be interpreted directly.
Nodes	The children a node has (child nodes). These entries are references to other nodes only.
Parameters	A list of the node's inputs and outputs in form of tuples. Each tuple is composed of two parts: (1) a reference to another node and (2) a reference to an input parameter or an output value of that node. If the first reference is not set, this means that the parameter is internal.

Table 12: Properties/attributes of a node definition.

Atomic type	Description
Generic	A global unique identifier.
Float	A floating point value.
Text	Characters as text string.
Scene	Used for nesting scenes.
Image	An image, typically a texture.
Dynamic	Dynamic values, e.g. time or sine values.
Mesh	Triangle based meshes.
Implicit	Implicit objects.

Table 13: Atomic types, that define a node (definition).

Editor

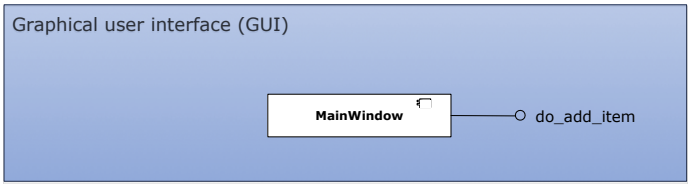


Figure 36: Component diagram of the editor component.

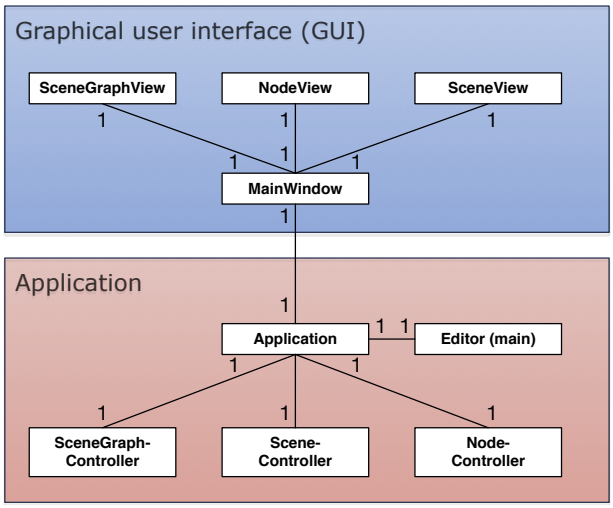


Figure 37: Entity relationship diagram of the editor component.

The editor component is the main component, which acts as entry point for the application and ties all components together. The `Application` class sets up all the controllers and the main window. The `MainWindow` class sets up all the view-related components, therefore the scene tree view, the scene view and the renderer.

Scene tree

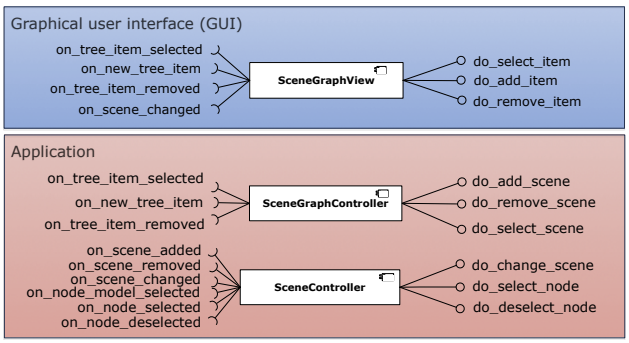


Figure 38: Component diagram of the scene tree component.

The scene tree component enables the scenes of the animation to be managed. User interaction is provided through a tree-like view, which lets the user add, remove and select scenes.

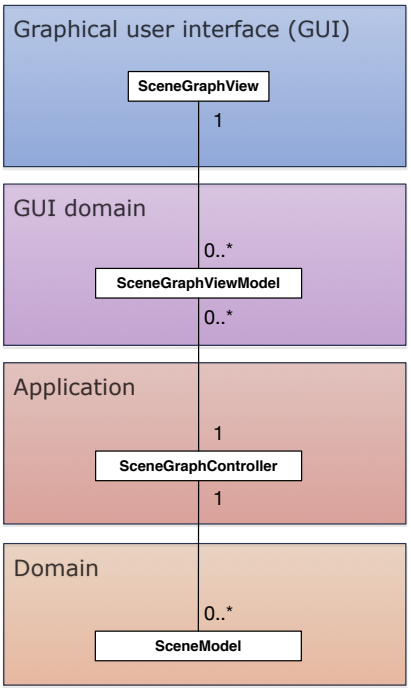


Figure 39: Entity relationship diagram of the scene tree component.

Node graph

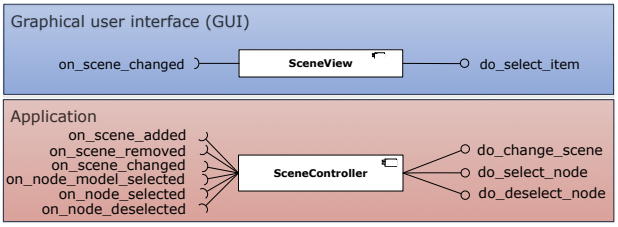


Figure 40: Component diagram of the node graph component.

THE NODE GRAPH COMPONENT enables the nodes of a scene to be managed. The nodes of a scene define its content.

EACH NODE HAS PARAMETERS which define the properties of a node. Such a parameter is for example the radius of a sphere for a node providing a sphere of type implicit.

NODES HAVE MULTIPLE INPUTS AND ONE OUTPUT where each is of a specific type, as described in table 13. Inputs point to parameters of a node whereas outputs provide values. An output of a node may be connected to the input of another node. Every scene has a fixed input by default which is currently limited to implicit types (outputs) and acts as output of the scene.

THE *content* OF A NODE is depending on the type of the node. The basis of the content is built by three things however: (1) its *definition*, (2) its *invocation* and (3) its so called *part*.

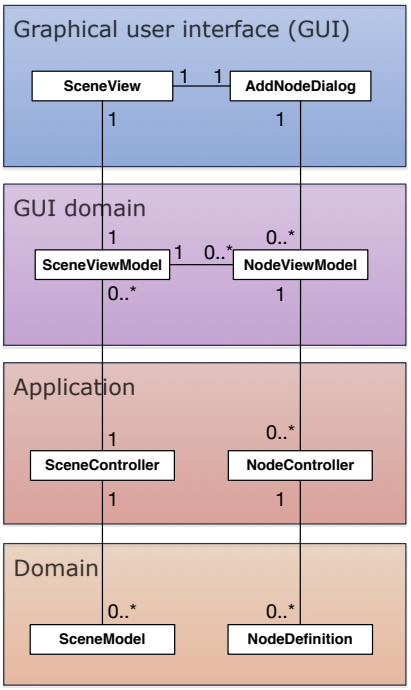


Figure 41: Entity relationship diagram of the node graph component.

A DEFINITION OF A NODE is essentially a method written in OpenGL Shading Language which defines the function of the node. fig. 42 shows an example of such a definition. It defines an implicit sphere with a certain radius at a certain position.

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

Figure 42: Implementation of an implicit sphere in the OpenGL Shading Language (GLSL) as definition in JSON format.

THE *invocation* OF A NODE is the call to a method defined by the definition of a node. Invocations are also written in the OpenGL Shading Language. fig. 43 shows an example of a invocation.

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

Figure 43: Calling of the previously defined GLSL function of an implicit sphere as invocation in JSON format.

A *part* OF A NODE defines what happens when a node is evaluated. This means usually evaluating the inputs of the node and use them as parameters. fig. 43 shows an example of a part.

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

Figure 44: The *part* of the node providing an implicit sphere in JSON format. Here the node node has two parameters: a radius and a position. The value for the radius is derived from the first input of the node, the position is a fixed vector. As the node is of type implicit it will be executed by a shader on the graphics processing unit.

EVALUATION OF NODES is done in two ways, the node is (1) directly selected or (2) connected to a scene (either through another node or through the scene's input) and that scene is evaluated.

NODES ARE DERIVED FROM node definitions. The workflow object of the component, the `NodeController` class, reads node definitions from the file system. A dialog window allows to add the read node definitions as instances to a scene. When a node is selected it is rendered by the renderer component.

Rendering

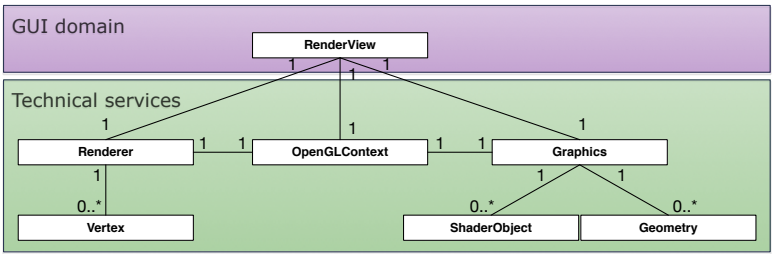


Figure 45: Entity relationship diagram of the renderer component.

THE RENDERING COMPONENT renders nodes and scenes. Nodes are rendered according to their type whereby only nodes of type implicit are currently supported. For rendering scenes the fixed input is evaluated recursively.

OPENGL IS USED FOR RENDERING. Due to the usage of "modern" OpenGL, everything that is rendered is rendered through a shader defined in the OpenGL Shading Language (GLSL).

AS ALGORITHM FOR RENDERING the sphere tracing algorithm is used as described in section "Rendering" and shown in fig. 46.

```
// Casts a ray from given origin in given direction. Stops
// at given maximal distance and after the given amount of
// steps. Maintains given precision.
vec3 castRay(in vec3 rayOrigin, in vec3 rayDirection,
             in float maxDistance,
             in float precision, in int steps)
{
    float latest          = precision * 2.0;
    float currentDistance = 0.0;
    float result          = -1.0;
    vec3  ray             = vec3(0);

    for(int i = 0; i < steps; i++) {
        if (abs(latest)     < precision ||
            currentDistance > maxDistance) {
                continue;
        }

        ray = rayOrigin + rayDirection * currentDistance;
        latest = scene1(ray);
        currentDistance += latest;
    }

    if (currentDistance < maxDistance) {
        result = currentDistance;
    }

    return result;
}
```

Figure 46: The sphere tracing algorithm as implemented.

FOR THE SHADING of objects Phong shading is used as described in section "Rendering" and shown in fig. 47.

LIGHTING AND MATERIALS are currently implemented statically. For lighting a light source is hard-coded within the shader. Figure 48 shows the implementation of the light source. Concerning materials currently the same material for all objects is used. Figure 49 shows the implementation of the material.

```
// Calculates the lighting for the given position, normal and direction,
// the given light (position and color) respecting the 'material'.
//
// This is mainly applying the phong lighting model.
//
// Returns the calculated color as three-dimensional vector.
vec3 calcLighting(in vec3 position, in vec3 normal, in vec3 rayDirection,
                  in vec3 material, in vec3 lightPosition, in vec3 lightColor,
                  in float currentDistance)
{
    vec3 color             = material;

    vec3 lightDirection    = normalize(lightPosition);
    vec3 reflection        = reflect(rayDirection, normal);

    vec3  directColor      = vec3(1.0, 1.0, 1.0);
    float kDirectLight     = 0.1;
    vec3  direct           = kDirectLight * directColor;

    vec3  ambientColor     = vec3(0.5, 0.7, 1.0);
    float kAmbient         = 1.2;
    float ambientExponent  = clamp(0.5 + 0.5 * normal.y, 0.0, 1.0);
    vec3  ambient          = kAmbient * ambientExponent * ambientColor;

    vec3  diffuseColor     = vec3(1.0, 0.85, 0.55);
    float kDiffuse         = 1.20;
    float expDiffuse       = clamp(dot(lightDirection, normal), 0.0, 1.0);
    vec3  diffuse          = kDiffuse * expDiffuse * diffuseColor;

    vec3  specularColor    = vec3(1.0, 0.85, 0.55);
    float kSpecular        = 1.2;
    float specularFactor   = 160.0;
    float specularExponent = pow(
                                clamp(dot(reflection, normal), 0.0, 1.0),
                                specularFactor
                             );
    vec3  specular         = kSpecular * specularExponent * specularColor * expDiffuse;

    vec3 light             = direct + diffuse + specular + ambient;
    color                  = color * light;
    color                  = mix(
                                color,
                                vec3(0.8, 0.9, 1.0),
                                1.0 - exp(-0.002 * currentDistance * currentDistance)
                             );

    return color;
}
```

Figure 47: Phong shading as implemented.

```
vec3 light1Color = vec3(0.9, 0.49, 0.83);
vec3 light1Position = vec3(0.6, 0.7, 1.5);
color = calcLighting(position, normal, rayDirection,
                     material, light1Position,
                     light1Color, currentDistance);
```

Figure 48: The hard-coded light source as implemented in the shader.

```
// Calculates the material based on a given distance.
vec3 calcMaterial(in float currentDistance)
{
    return 0.45 + 0.3 * sin(vec3(0.05, 0.08, 0.10) * (currentDistance - 10.0));
}
```

Figure 49: The hard-coded material as implemented in the shader.

Discussion and conclusion

The previous chapter showed the results of this thesis.

This chapter offers a discussion of the results and a conclusion.

In the first section the results of this thesis are interpreted. The first section builds the basis for the second section, the conclusion. The second section summarizes the most important results relating to the objectives and provides an outlook what might follow up this thesis.

Discussion

The results of this thesis are a software architecture, a literate program and an implement program. The software architecture builds the basis for the program implemented. The literate program documents the program implemented while at the same time providing the actual implementation.

Software architecture

The software architecture is defined by three aspects: an architectural software design pattern, layers and signals and slots.

The used architectural software design pattern is a combination of the model-view-view model and the model-view-controller software design patterns. The pattern separates data from its representation and ensures a coherent design.

A relaxed layered architecture is used to reduce coupling and dependencies. The architecture allows higher layers to communicate with any lower layer.

Signals and slots, which are a generalized implementation of the observer pattern, are used to allow communication between components. This prevents high coupling and low cohesion.

Literate program

The developed literate program explains how a program for modeling, composing and rendering real time computer graphics

using sphere tracing is developed. The used literate programming paradigm allows to explain to human beings what the computer shall do instead of focusing on instructing the computer what to do by writing program code and forcing readers to read the program code.

THE LITERATE PROGRAM STARTS with a description of what will be achieved. From this description the main components are derived and then subsequently implemented. This shows also the overall structure of the literate program: a certain concept is first introduced by describing it and then implemented.

Program

USING THE INTRODUCED METHODOLOGIES and the developed software architecture the editor component of the program was implemented.

ANIMATIONS ARE THE BUILDING BLOCKS of the editor component and contain scenes, which contain nodes. Currently the developed editor component allows one animation to be managed.

SCENES BUILD THE BASIS of an animation. They can be managed using a tree like structure. Each scene holds nodes in a graph structure.

NODES REPRESENT THE CONTENT of a scene and therefore of an animation. Nodes are defined by node definitions which are read from the file system. The program implemented supports currently only nodes which define implicit objects.

ANIMATIONS ARE EVALUATED for rendering. Evaluating an animation means evaluating scenes which means evaluating nodes. The result is OpenGL Shading Language specific code which is executed by the graphical processing unit (GPU) and seen by the viewer.

RENDERING IS DONE using the sphere tracing algorithm. For shading the Phong shading technique is used.

Conclusion

THE MAIN OBJECTIVE OF THIS THESIS is the design and development of a program for modeling, composing and rendering real time computer graphics by providing a graphical toolbox. To reach this main objective additional objectives were defined, which can be found at subsection "Objectives and limitations"

ALL OF THE OBJECTIVES COULD BE REACHED, not all of the objectives and components are as elaborated as originally intended however. The program implemented provides at this time no possibility

to store and load animations and supports only nodes which define implicit objects. As the whole node structure is very generalized (by using definitions for nodes) new node types can be implemented without much effort. At this time no connections between nodes are possible, the program implemented only allows the evaluation of a single node at a time. Point light sources are directly implemented (hard-coded) within the shader and not as nodes as intended. The same applies for materials of objects.

The use of the literate programming paradigm was a whole new experience to the author.

Literate programming has a lot of advantages, such as focusing on explaining to human beings what the computer shall do in terms of ideas and concepts instead of instructing the computer what to do by writing only program code. As combining the documentation with code in a single document builds the basis of literate programming, the documentation of the program is inherent. This prevents unforeseen cost and time overruns.

Literate programming faces also several problems however: it requires a different way of thinking from traditional methodologies, it is very expressive and verbose, and the documentation can become lengthy and hard to read, especially when including the full implementation of technical details. Although these problems were overcome in the writing of this thesis, they led to significant overhead which affected the whole thesis in terms of less productivity. When someone is used to the paradigm these problems can be prevented upfront and the paradigm may be not much more costly than using traditional methodologies (at least for smaller programs).

Sphere tracing, the algorithm used for rendering, is able to produce high quality, realistic looking images in real time without being overly complex. It has however a clear disadvantage: the de facto way of representing the surface of objects using triangle based meshes cannot be used directly. Instead, distance functions have to be used for modeling the surfaces as seen from any view point. This disadvantage may be the reason that sphere tracing seems not to be used production. However, the algorithm seems to coming into use in production, for example for calculating ambient occlusion [25] or soft shadows [26]. Time will tell if the method will establish itself further and become practicable for rendering conventional meshes.

As this thesis has a limited time frame of one semester, not all desired topics could be treated and not all of the set objectives and components are as elaborated as originally intended.

Further topics for the continuation of this thesis might be additional features for the editor component and the development of the

player component, which can be used as standalone program.

ADDITIONAL FEATURES FOR THE EDITOR component could be the following: 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).

ADDITIONAL FEATURES FOR RENDERING could be the implementation of ambient occlusion, the implementation of realistic materials based on the bidirectional reflectance distribution function and the acceleration of the sphere tracing algorithm.

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.

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 Appendix "Code fragments".

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 [16] version 3.5.x or above
- PyQt5 [17] version 5.7 or above
- OpenGL [18] version 3.3 or above

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

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

[18] `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.[19]

[19] `https://docs.python.org/3/reference/import.html#packages`

To prevent having multiple modules having the same name, name spaces are used.[20] 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.

[20] `https://docs.python.org/3/tutorial/classes.html#python-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 [21].

However, PEP number 8 recommends to either import modules directly or to import the needed functionality directly. [22]. As defined by the coding style, subsection "Framework for implementation", imports are done by the `from Foo import Bar` syntax.

[21] https://docs.python.org/3/tutorial/modules.html

[22] 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.

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 are 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." [23]

[23] `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 rather be confusing. Instead it is more intuitive to have only a minimal entry point which is clearly visible as such. Therefore the main

⟨ *Main entry point* 87a ⟩ ≡

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

Fragment referenced in 190.

Figure 50: Main entry point of the editor application.

Editor → Main entry point

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 [24].

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]

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

⟨ *Main application declarations* 87b ⟩ ≡

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

    ⟨ Main application constructor 88a ⟩
    ⟨ Main application methods 117a ⟩◇
```

Fragment referenced in 191a.

Figure 51: Main application class of the editor application.

Editor → Application

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.

⟨ *Main application constructor* 88a ⟩ ≡

```
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 88b, ... ⟩
    ⟨ Set up components for main application 90b ⟩
    ⟨ Add root node for main application 97b ⟩
    ⟨ Set model for scene graph view 106b ⟩
    ⟨ Load nodes 181b ⟩
    self.main_window.show()◇
```

Fragment referenced in 87b.

Figure 52: Constructor of the editor application class.

Editor → Application → 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.

⟨ *Set up internals for main application* 88b ⟩ ≡

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

Fragment defined by 88b, 117b.
Fragment referenced in 88a.

Figure 53: Setting up the internals for the main application class.

Editor → Application → Constructor

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

Main window

HAVING A VERY BASIC IMPLEMENTATION of the main application, its view component, the main window, can now be implemented and then be set up by the main application.

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

⟨ *Main window declarations* 89a ⟩ ≡

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

    ⟨ Main window signals 89b ⟩

    ⟨ Main window methods 90a, ... ⟩
◇
```

Fragment referenced in 192a.

Figure 54: Main window class of the editor application.

Editor → Main window

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

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

⟨ *Main window signals* 89b ⟩ ≡

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

Fragment referenced in 89a.

Figure 55: 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* 90a ⟩ ≡

```
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 90a, 92.
Fragment referenced in 89a.

Figure 56: 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* 90b ⟩ ≡

```
⟨ Set up controllers for main application 105b, ... ⟩
⟨ Connect controllers for main application 136b, ... ⟩
⟨ Set up main window for main application 90c ⟩◇
```

Fragment referenced in 88a.

Figure 57: Setting up of components for the main application class.

Editor → Main application → Constructor

⟨ *Set up main window for main application* 90c ⟩ ≡

```
self.main_window = qde_main_window.MainWindow()
self.main_window.move(100, 100)
self.main_window.do_close.connect(self.quit)
⟨ Connect main window components 114b, ... ⟩◇
```

Fragment referenced in 90b.

Figure 58: Set up of the editor main window and its signals from within the main application.

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. [25]

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

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

Targeting a look as proposed in *QDE - a visual animation system. Software-Architektur.* p. 9, a simple grid layout does however not provide enough possibilities. Instead a horizontal box layout in combination with splitters is used.

Recalling the components, the following layout is approached, which can be seen in fig. 59:

- 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

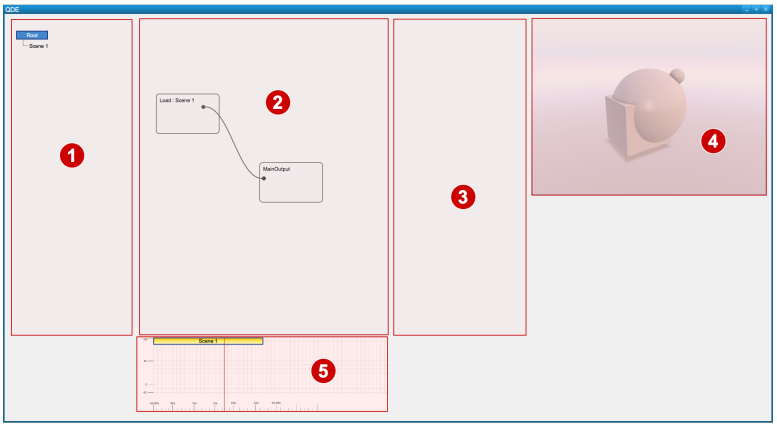


Figure 59: A mock up of the editor application showing its components.
1: Scene tree.
2: Node graph.
3: Parameter view.
4: Rendering view.
5: Time line.

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* 92 ⟩+ ≡

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

    self.setObjectName('MainWindow')
    self.setWindowTitle('QDE')
    self.resize(1024, 768)
    # 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 = gui_scene.SceneGraphView(self)
    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 135 ⟩
    # Set up parameter view in main window
    ⟨ Set up render view in main window 192b ⟩

    horizontal_splitter = QtWidgets.QSplitter()
    ⟨ Add render view to horizontal splitter in main window 192c ⟩
    # 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 136a ⟩

    horizontal_layout.addWidget(vertical_splitter)
◇
```

Fragment defined by 90a, 92.
Fragment referenced in 89a.

Figure 60: 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.

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* 93 ⟩ ≡

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

    # Signals

    ⟨ Scene model methods  94a ⟩

    # Slots
◇
```

Fragment referenced in 193a.

Figure 61: 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.

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

Concerning the user interface, a view model must fulfill the requirements posed by the user interface's corresponding component. In this case this are actually two components: the scene graph view as well as the scene view.

⟨ *Scene model methods* 94a ⟩ ≡

```
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 93.

Figure 62: The constructor of the scene model.

Editor → Scene model → Constructor

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

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

⟨ *Scene graph view model declarations* 94b ⟩ ≡

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

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

    # Signals

    ⟨ Scene graph view model constructor 95, ... ⟩
    ⟨ Scene graph view model methods 105a, ... ⟩

    # Slots
◇
```

Fragment referenced in 193b.

Figure 63: Definition of the scene graph view model class, which corresponds to an entry within the scene graph.

Editor → Scene graph view model

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

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

⟨ *Scene graph view model constructor* 95 ⟩ ≡

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

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

    super(SceneGraphViewModel, self).__init__(parent)

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

Fragment defined by 95, 104b.
Fragment referenced in 94b.

Figure 64: The constructor of the scene graph view model.

Editor → Scene graph view model → Constructor

As the scene graph shall be built as a tree structure, an appropriate data structure is needed. Qt provides the `QTreeWidget` class, but that class is in this case not suitable, as it does not separate the data from its representation, as stated by Qt: "Developers who do not need the flexibility of the Model/View framework can use this class to create simple hierarchical lists very easily. A more flexible approach involves combining a QTreeView with a standard item model. This allows the storage of data to be separated from its representation."[26]

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

Such a standard item model is `QAbstractItemModel`[27], which is used as a base class for the scene graph controller.

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

⟨ *Scene graph controller declarations* 96a ⟩ ≡

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

    ⟨ Scene graph controller signals 115 ⟩
    ⟨ Scene graph controller constructor 96b, ... ⟩
    ⟨ Scene graph controller methods 99, ... ⟩
    ⟨ Scene graph controller slots 112 ⟩
◇
```

Fragment referenced in 194b.

Figure 65: The scene graph controller, inheriting from `QAbstractItemModel`.

Editor → Scene graph controller

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

⟨ *Scene graph controller constructor* 96b ⟩ ≡

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

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

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

Fragment defined by 96b, 98.
Fragment referenced in 96a.

Figure 66: Constructor of the scene graph controller.

Editor → Scene graph controller → Constructor

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

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

⟨ *Scene graph controller add root node* 97a ⟩ ≡

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

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

Fragment referenced in 198b.

Figure 67: A method to add the root node from within the scene graph controller.

Editor → Scene graph controller → Methods

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

⟨ *Add root node for main application* 97b ⟩ ≡

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

Fragment referenced in 88a.

Figure 68: The root node of the scene graph being added by the main application.

Editor → Main application → Constructor

The scene graph controller must also provide the header data, which is used to display the header within the view (due to the usage of the Qt view model [22]). As header data the name of

the scenes as well as the number of nodes a scene contains shall be displayed.

⟨ *Scene graph controller constructor* 98 ⟩+ ≡

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

Fragment defined by 96b, 98.
Fragment referenced in 96a.

Figure 69: Initialization of the header data and the root node of the scene graph.

Editor → Scene graph controller → Constructor

As QAbstractItemModel is used as a basis for the scene graph controller, some methods must be implemented at very least: "When sub classing QAbstractItemModel, at the very least you must implement index(), parent(), rowCount(), columnCount(), and data(). These functions are used in all read-only models, and form the basis of editable models."[27]

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

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

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

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

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

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

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

⟨ *Scene graph controller methods* 99 ⟩ ≡

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

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

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

    if not parent.isValid():
        return self.createIndex(row, column, self.view_root_node)

    parent_node = parent.internalPointer()
    child_nodes = parent_node.children()

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

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

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 70: Implementation of QAbstractItemModel's index method for the scene graph controller.

Editor → Scene graph controller → Methods

⟨ *Scene graph controller methods* 100 ⟩+ ≡

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

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

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

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

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

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

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 71: Implementation of QAbstractItemModel's parent method for the scene graph controller.

Editor → Scene graph controller → Methods

⟨ *Scene graph controller methods* 101 ⟩+ ≡

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

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

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

    column_count = len(self.header_data) - 1

    return column_count
◇
```

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 72: Implementation of QAbstractItemModel's columnCount method for the scene graph controller.

Editor → Scene graph controller → Methods

In addition to the above mentioned methods, the `QAbstractItemModel` offers the method `headerData`, which "returns the data for the given role and section in the header with the specified orientation."[28]

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

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

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

It does not make sense however to use the list of nodes from the domain model, as the view model will hold references to graphical objects where as the domain model holds only pure data objects. Therefore it is necessary, that the scene view model keeps track of its nodes separately.

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

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

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

What is still missing, is the actual representation of the scene graph in terms of a view. Qt offers a plethora of widgets for implementing views. One such widget is `QTreeView`, which "implements

⟨ *Scene graph controller methods* 102 ⟩+ ≡

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

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

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

    if not parent.isValid():
        row_count = 1
    else:
        # Get the actual object stored by the parent. In this case
        # it is a SceneGraphViewModel.
        node = parent.internalPointer()

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

    return row_count
```

◇

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 73: Implementation of QAbstractItemModel's rowCount method for the scene graph controller.

Editor → Scene graph controller → Methods

⟨ *Scene graph controller methods* 103 ⟩+ ≡

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

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

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

    if not model_index.isValid():
        self.logger.debug("Model index is not valid")
        return None

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

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

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

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

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 74: Implementation of QAbstractItemModel's data method for the scene graph controller.

Editor → Scene graph controller → Methods

⟨ *Scene graph controller methods* 104a ⟩+ ≡

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

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

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

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

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

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 75: Implementation of QAbstractItemModel's headerData method for the scene graph controller.

Editor → Scene graph controller → Methods

⟨ *Scene graph view model constructor* 104b ⟩+ ≡

```
    self.nodes = []
◇
```

Fragment defined by 95, 104b.
Fragment referenced in 94b.

Figure 76: Scene graph view models hold references to the nodes they contain.

Editor → Scene graph view model → Constructor

⟨ *Scene graph view model methods* 105a ⟩ ≡

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

    return len(self.nodes)
◇
```

Fragment defined by 105a, 193c.
Fragment referenced in 94b.

Figure 77: The number of (graphical) nodes which a scene graph view model contains implemented as a property.

Editor → Scene graph view model → Methods

⟨ *Set up controllers for main application* 105b ⟩ ≡

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

Fragment defined by 105b, 137b, 181a.
Fragment referenced in 90b.

Figure 78: The scene graph controller gets initialized within the main application.

Editor → Main application → Constructor

a tree representation of items from a model. This class is used to provide standard hierarchical lists that were previously provided by the QListView class, but using the more flexible approach provided by Qt's model/view architecture." [29] Therefore `QTreeView` is used as basis for the scene graph view.

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

⟨ *Scene graph view declarations* 105c ⟩ ≡

```
⟨ Scene graph view decorators 195b ⟩
class SceneGraphView(QtWidgets.QTreeView):
    """The scene graph view widget.
    A widget for displaying and managing the scene graph.
    """

    ⟨ Scene graph view signals 108b, ... ⟩
    ⟨ Scene graph view constructor 106a, ... ⟩
    ⟨ Scene graph view methods 108a ⟩
    ⟨ Scene graph view slots 107, ... ⟩
◇
```

Fragment referenced in 195a.

Figure 79: Scene graph view, based on Qt's QTreeView.

Editor → Scene graph view

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

For being able to display something, the scene graph view needs a controller to work with. In terms of Qt, the controller is

⟨ *Scene graph view constructor* 106a ⟩ ≡

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

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

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

Fragment defined by 106a, 109b, 110.
Fragment referenced in 105c.

Figure 80: Constructor of the scene graph view.

Editor → Scene graph view → Constructor

called a model, as due its model/view architecture. This model may although not be set too early, as otherwise problems arise. It may only then be added, when the depending components are properly initialized, e.g. when the root node has been added.

⟨ *Set model for scene graph view* 106b ⟩ ≡

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

Fragment referenced in 88a.

Figure 81: The scene graph controller is being set as the scene graph view's model.

Editor → Main application → Constructor

BUT SCENES SHALL NOT ONLY BE DISPLAYED, instead it shall be possible to work with them. What shall be achieved, are three things: (1) Adding and removing scenes, (2) renaming scenes and (3) switching between scenes.

TO SWITCH BETWEEN SCENES it is necessary to emit what scene was selected. This is needed to tell the other components, such as the node graph for example, that the scene has changed.

Through the `selectionChanged` signal the scene graph view already provides a possibility to detect if another scene was selected. This signal emits an item selection in terms of model indices although.

As this is very view- and model-specific, it would be easier for other components if the selected scene is emitted directly. To emit the selected index of the currently selected scene directly, the slot `on_tree_item_selected` is introduced.

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

⟨ *Scene graph view slots* 107 ⟩ ≡

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

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

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

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

    selected_item = selected.first()
    selected_index = selected_item.indexes()[0]
    self.do_select_item.emit(selected_index)
    self.logger.debug(
        "Tree item was selected: %s" % selected_index
    )◇
```

Fragment defined by 107, 111.
Fragment referenced in 105c.

Figure 82: Slot which is called when the selection within the scene graph view is changed.

Editor → Scene graph view → Slots

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

⟨ *Scene graph view methods* 108a ⟩ ≡

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

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

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

    super(SceneGraphView, self).setModel(model)

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

    # Set the index to the first node of the model
    self.setCurrentIndex(model.index(0, 0))
    self.logger.debug("Root node selected")◇
```

Fragment referenced in 105c.

Figure 83: The setModel method, provided by QTreeView's interface, which is begin overwritten for being able to trigger the on_tree_item_selected slot whenever the selection in the scene graph view has changed.

Editor → Scene graph view → Methods

As stated in the above code fragment, `on_tree_item_selected` emits another signal containing a reference to the currently selected scene, which needs to be implemented as well.

⟨ *Scene graph view signals* 108b ⟩ ≡

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

Fragment defined by 108b, 109a.
Fragment referenced in 105c.

Figure 84: The signal that is being emitted when a scene within the scene graph view was selected. Note that the signal includes the model index of the selected item.

Editor → Scene graph view → Signals

ADDING AND REMOVING OF A SCENE are implemented in a similar manner as the selection of an item was implemented. However, the tree widget does not provide direct signals for those cases as it is

the case when selecting a tree item, instead own signals, slots and actions have to be used.

⟨ *Scene graph view signals* 109a ⟩+ ≡

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

Fragment defined by 108b, 109a.
Fragment referenced in 105c.

Figure 85: Signals that get emitted whenever a scene is added or removed.

Editor → Scene graph view → Signals

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

ADDING OF A SCENE ITEM shall happen when pressing the a key on the keyboard.

⟨ *Scene graph view constructor* 109b ⟩+ ≡

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

Fragment defined by 106a, 109b, 110.
Fragment referenced in 105c.

Figure 86: Introduction of an action for adding a new scene, which reacts upon the "A" key being pressed on the keyboard.

Editor → Scene graph view → Constructor

THE REMOVAL OF A SELECTED NODE shall be triggered upon the press of the `delete` and the `backspace` key on the keyboard.

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

ONE OF THE MENTIONED OTHER COMPONENTS is the scene graph controller. He needs to be informed whenever a scene was added, removed or selected, so that he is able to manage his data model correspondingly.

DESPITE HAVING THE SLOTS FOR ADDING, REMOVING AND SELECT-

⟨ *Scene graph view constructor* 110 ⟩+ ≡

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

Fragment defined by 106a, 109b, 110.
Fragment referenced in 105c.

Figure 87: Introduction of an action for removing a new scene, which reacts upon the "delete" key being pressed on the keyboard.

Editor → Scene graph view → Constructor

ing scene graph items implemented, the actual methods for adding and removing scenes, `on_tree_item_added` and `on_tree_item_removed`, are still missing.

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

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

The same logic applies when removing a scene.

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

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

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

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

As stated before, in section "Literate programming", it was an intention of this project to develop the application test driven. Due to the required amount of work when developing test driven, it was abstained from this intention.

But nevertheless, it would be very handy to have at least some

⟨ *Scene graph view slots* 111 ⟩+ ≡

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

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

    selected_indexes = self.selectedIndexes()

    # Sanity check: is actually an item selected?
    if len(selected_indexes) > 0:
        selected_item = selected_indexes[0]
        self.do_add_item.emit(selected_item)
        ⟨ Scene graph view log tree item added 119b ⟩

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

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

    selected_indexes = self.selectedIndexes()

    # Sanity check: is actually an item selected? And has that
    # item a parent?
    # We only allow removal of items with a valid parent, as we
    # do not want to have the root item removed.
    if len(selected_indexes) > 0:
        selected_item = selected_indexes[0]
        if selected_item.parent().isValid():
            self.do_remove_item.emit(selected_item)
            ⟨ Scene graph view log tree item removed 119c ⟩
        else:
            self.logger.warn("Root scene cannot be deleted")
    else:
        self.logger.warn('No item selected for removal')
◇
```

Fragment defined by 107, 111.
Fragment referenced in 105c.

Figure 88: Slots which emit themselves a signal whenever a scene is added from the scene graph or removed respectively.

Editor → Scene graph view → Slots

⟨ *Scene graph controller slots* 112 ⟩ ≡

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

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

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

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

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

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

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

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

Fragment referenced in 96a.

Figure 89: Slots to handle adding, removing and selecting of tree items within the scene graph. The slots take a model index as argument (coming from QAbstractItemModel). This is analogous to the scene graph view.

Editor → Scene graph controller → Slots

⟨ *Scene graph controller methods* 113 ⟩+ ≡

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

    if not parent.isValid():
        return False

    parent_node = parent.internalPointer()
    self.beginInsertRows(parent, row, row + count - 1)
    domain_model  = scene_domain.SceneModel(
        parent_node.domain_object
    )
    view_model = scene_gui_domain.SceneGraphViewModel(
        row=row,
        domain_object=domain_model,
        parent=parent_node
    )
    self.endInsertRows()

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

    return True
◇
```

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 90: Method for adding new scenes in terms of a domain model as well as a scene graph view model.

Editor → Scene graph controller → Methods

⟨ *Scene graph controller methods* 114a ⟩+ ≡

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

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

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

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

    return True
◇
```

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

Figure 91: Method for removing scenes. Note that this is mainly done by getting the object related to the given model index and setting the parent of that object to a nil object.

Editor → Scene graph controller → Methods

⟨ *Connect main window components* 114b ⟩ ≡

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

Fragment defined by 114b, 143c, 189b, 191b.
Fragment referenced in 90c.

Figure 92: The scene graph view's signals for adding, removing and selecting a scene are connected to the corresponding slots from the scene graph controller. Or, in other words, the controller/data reacts to actions invoked by the user interface.

Editor → Main application → Constructor

⟨ *Scene graph controller signals* 115 ⟩ ≡

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

Fragment referenced in 96a.

Figure 93: Signals emitted by the scene graph controller, in terms of domain models, whenever a scene is added, removed or selected.

Editor → Scene graph controller → Signals

idea what the code is doing at certain places and at certain times.

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

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

Logging

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

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

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

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

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

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

⟨ *Main application methods* 117a ⟩ ≡

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

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

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

Fragment referenced in 87b.

Figure 94: A method for setting up the logging, provided by the main application. If there exists an external configuration file for logging, this file is used for configuring the logging facility. Otherwise the standard configuration is used.

Editor → Main application → Methods

⟨ *Set up internals for main application* 117b ⟩+ ≡

```
self.setup_logging()◇
```

Fragment defined by 88b, 117b.
Fragment referenced in 88a.

Figure 95: Set up of the logging from within the main application class.

Editor → Main application → Constructor

THE CONSEQUENCE OF PROVIDING logging on a class basis, as stated before, is, that each class has to instantiate a logging instance. To prevent the repetition of the same code fragment over and over, Python's decorator pattern is used [30].

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

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

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

⟨ *Set logger name* 118a ⟩ ≡

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

Fragment referenced in 200a.

- Provide an easy to use interface for logging.

⟨ *Logger interface* 118b ⟩ ≡

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

return cls◇
```

Fragment referenced in 200a.

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

THE LOGGING FACILITY MAY NOW BE USED wherever it is useful to log something. Such a place is for example the adding and removal of scenes in the scene graph view.

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

Now, having the scene graph component as well as an interface to log messages throughout the application implemented, the next component may be approached.

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

⟨ *With logger example* 119a ⟩ ≡

```
from qde.editor.foundation import common

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

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

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

◇
```

Fragment never referenced.

Figure 96: An example of how to use the logging decorator in a class.

⟨ *Scene graph view log tree item added* 119b ⟩ ≡

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

Fragment referenced in 111.

Figure 97: The scene graph view logs a corresponding message whenever an item is added from the scene graph. Note, that this logging only happens in *debug* mode.

Editor → Scene graph view → Methods

⟨ *Scene graph view log tree item removed* 119c ⟩ ≡

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

Fragment referenced in 111.

Figure 98: The scene graph view logs a corresponding message whenever an item is removed from the scene graph. Note, that this logging only happens in *debug* mode.

Editor → Scene graph view → Methods

Node graph

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

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

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

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

Nodes Building blocks of a real time animation.

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

Scenes A composition of nodes, connected by edges.

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

Nodes

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

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

MULTIPLE KINDS OF NODE REPRESENTATIONS shall be supported by the system: images, triangle based meshes and solid modeling through function modeling (using signed distance functions for modeling implicit surfaces). Whereas triangle based meshes may either be loaded from externally defined files (e.g. in the Filmbox (FBX), the Alembic (ABC) or the Object file format (OBJ)) or directly be generated using procedural mesh generation.

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

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

WHEN THINKING OF NODES USED FOR SOLID MODELING however, it may appear, that they may be evaluated directly, without the need for pre-processing, as they are fully implementable using shader code only. This is kind of misleading however, as each node has its own definition which has to be added to shader and this definition is then used in a mapping function to compose the scene. This would mean to add a definition of a node over and over again, when spawning multiple instances of the same node type, which results in overhead bloating the shader. It is therefore necessary to pre-process solid modeling nodes too, exactly as triangle mesh based and image nodes, for being able to use multiple instances of the same node type within a scene while having the definition added only once.

ALL OF THESE THOUGHTS SUM UP in one central question for the implementation: Shall objects be predefined within the code (and therefore only nodes accepted whose type and sub type match those of predefined nodes) or shall all objects be defined externally using files?

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

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

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

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

EACH NODE CAN HAVE MULTIPLE INPUTS AND AT LEAST ONE OUTPUT. The inputs may be either be atomic types (which have to be defined) or references to other nodes. The same applies to the outputs.

A NODE CONSISTS ALSO OF A DEFINITION. In terms of implicit surfaces this section contains the actual definition of a node in terms of the implicit function. In terms of triangle based meshes this is the part where the mesh and all its prerequisites as vertex array buffers and vertex array objects are set up or used from a given context.

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

A NODE SHALL BE ABLE TO HAVE ONE OR MORE PARTS. A part typically contains the "body" of the node in terms of code and represents therefore the code-wise implementation of the node. A part

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

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

EACH NODE CAN HAVE MULTIPLE CONNECTIONS. A connection is composed of an input and an output plus a reference to a part. The input respectively the output may be zero, what means that the part of the input or output is internal.

Or, a bit more formal:

⟨ *Connections between nodes in EBNF notation* 123 ⟩ ≡

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

Figure 99: Connections between nodes in EBNF notation.

Fragment never referenced.

RECAPITULATING THE ABOVE MADE THOUGHTS a node is essentially composed by the following elements:

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

THE INPUTS AND OUTPUTS MAY BE PARAMETERS OF AN ATOMIC TYPE, as stated above. This seems like a good point to define the atomic types the system will have:

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

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

Component	Description
ID	A global unique identifier (UUID[1])
Name	The name of the node, e.g. "Cube".
Description	A description of the node's purpose.
Inputs	A list of the node's inputs. The inputs may either be parameters (which are atomic types such as float values or text input) or references to other nodes.
Outputs	A list of the node's outputs. The outputs may also either be parameters or references to other nodes.
Definitions	A list of the node's definitions. This may be an actual definition by a (shader-) function in terms of an implicit surface or prerequisites as vertex array buffers in terms of a triangle based mesh.
Invocation	A list of the node's invocations or calls respectively.
Parts	Defines parts that may be processed when evaluating the node. Contains code which can be interpreted directly.
Nodes	The children a node has (child nodes). These entries are references to other nodes only.
Connections	A list of connections of the node's inputs and outputs. Each connection is composed by two parts: A reference to another node and a reference to an input or an output of that node. Is the reference not set, that is, its value is zero, this means that the connection is internal.

Table 14: Components a node is composed of.

For identification of the atomic types, an enumerator is used. Python provides the `enum` module, which provides a convenient interface for using enumerations[31].

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

Figure 100: Types of a node wrapped in a class, implemented as an enumerator.

Editor → Types → Node type

⟨ *Node type declarations* 124 ⟩ ≡

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

    GENERIC  = 0
    FLOAT    = 1
    TEXT     = 2
    SCENE    = 3
    IMAGE    = 4
    DYNAMIC  = 5
    MESH     = 6
    IMPLICIT = 7
◇
```

Fragment referenced in 201.

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

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

⟨ *Parameter declarations* 125 ⟩ ≡

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

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

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

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

Fragment defined by 125, 126, 203.
Fragment referenced in 202.

Figure 101: The atomic type class which builds the basis for node parameters. Note that the type of an atomic type is defined by the before implemented node type.

Editor → Parameters → Atomic type

As the word atomic indicates, these types are atomic, meaning there only exists one explicit instance per type, which is therefore static. As can be seen in the code fragment below, the atomic types are parts of node definitions themselves. Only the creation of the generic atomic type is shown, the rest is omitted and can be found at Appendix "Code fragments".

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

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

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

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

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

⟨ *Parameter declarations* 126 ⟩+ ≡

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

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

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

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

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

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

        return node_definition_part

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

Fragment defined by 125, 126, 203.
Fragment referenced in 202.

Figure 102: A class which creates and holds all atomic types of the editor. Note that at this point only an atomic type for generic nodes is being created.

Editor → Parameters → Atomic types

⟨ *Node domain model declarations* 127a ⟩ ≡

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

    # Signals

    ⟨ Node domain model constructor  127b, ... ⟩

    ⟨ Node domain model methods  130a ⟩

    # Slots
◇
```

Fragment referenced in 207b.

Figure 103: Definition of the node (domain) model.

Editor → Node model

⟨ *Node domain model constructor* 127b ⟩ ≡

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

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

    self.id_    = id_
    self.name = name

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

Fragment defined by 127b, 131c.
Fragment referenced in 127a.

Figure 104: Constructor of the node (domain) model.

Editor → Node model → Constructor

⟨ *Node view model declarations* 128a ⟩ ≡

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

    # Constants
    WIDTH = 120
    HEIGHT = 40

    # Signals
    ⟨ Node view model signals 214c ⟩

    ⟨ Node view model constructor 128b, ... ⟩

    ⟨ Node view model methods 129a, ... ⟩
◇
```

Fragment referenced in 208a.

Figure 105: Definition of the node view model.

Editor → Node view model

⟨ *Node view model constructor* 128b ⟩ ≡

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

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

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

    self.position = QtCore.QPoint(0, 0)

    self.width  = NodeViewModel.WIDTH
    self.height = NodeViewModel.HEIGHT
◇
```

Fragment defined by 128b, 131a, 208b.
Fragment referenced in 128a.

Figure 106: Constructor of the node view model.

Editor → Node view model → Constructor

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

⟨ *Node view model methods* 129a ⟩ ≡

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

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

    return self.domain_object.type_
◇
```

Fragment defined by 129ab, 130b, 131b, 132a, 209, 215.
Fragment referenced in 128a.

Figure 107: The type attribute of the node view model as property.

Editor → Node view model → Methods

⟨ *Node view model methods* 129b ⟩+ ≡

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

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

    return self.domain_object.name
◇
```

Fragment defined by 129ab, 130b, 131b, 132a, 209, 215.
Fragment referenced in 128a.

Figure 108: The name attribute of the node view model as property.

Editor → Node view model → Methods

THE DOMAIN MODEL DOES NOT PROVIDE ACCESS to its type at the moment however. The type is directly derived from the primary output of a node. If a node has no outputs at all, its type is assumed to be generic.

CONCERNING THE DRAWING OF NODES (or painting, as Qt calls it) , each node type may be used multiple times. But instead of re-creating the same image representation over and over again, it makes sense to create it only once per node type. Qt provides `QPixmap` and `QPixmapCache` for this use case.

EACH NODE HAS A CACHE KEY ASSIGNED, which is used to identify

⟨ *Node domain model methods* 130a ⟩ ≡

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

    type_ = types.NodeType.GENERIC

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

    return type_◇
```

Fragment referenced in 127a.

Figure 109: The type attributes of the node domain model as property.

Editor → Node (domain) model → Methods

⟨ *Node view model methods* 130b ⟩+ ≡

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

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

    ⟨ Node view model methods paint  132b, ... ⟩
◇
```

Fragment defined by 129ab, 130b, 131b, 132a, 209, 215.
Fragment referenced in 128a.

Figure 110: The paint method of the node view model. When a pixmap is being created, it gets cached immediately, based on its type, status and its selection status. If a pixmap already existing for a given tripe, type, status and selection, that pixmap is used.

Editor → Node view model → Methods

that node.

⟨ *Node view model constructor* 131a ⟩+ ≡

```
    self.cache_key = None
◇
```

Fragment defined by 128b, 131a, 208b.
Fragment referenced in 128a.

Figure 111: The cache key is being initialized within a node's constructor.

Editor → Node view model → Constructor

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

⟨ *Node view model methods* 131b ⟩+ ≡

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

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

Fragment defined by 129ab, 130b, 131b, 132a, 209, 215.
Fragment referenced in 128a.

Figure 112: A method which creates a cache key based on the type, the status and the state of selection of a node.

Editor → Node view model → Methods

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

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

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

⟨ *Node domain model constructor* 131c ⟩+ ≡

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

Fragment defined by 127b, 131c.
Fragment referenced in 127a.

Figure 113: The status of the node is being initialized within the node's constructor.

Editor → Node domain model → Constructor

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

⟨ *Node view model methods* 132a ⟩+ ≡

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

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

    return self.domain_object.status
◇
```

Fragment defined by 129ab, 130b, 131b, 132a, 209, 215.
Fragment referenced in 128a.

Figure 114: The status of a node view model is obtained by accessing the domain model's status.

Editor → Node domain model → Methods

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

⟨ *Node view model methods paint* 132b ⟩ ≡

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

Fragment defined by 132bc, 133, 214a.
Fragment referenced in 130b.

Figure 115: A cache key is being created when no cache key for the given attributes is found.

Editor → Node view model → Methods → Paint

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

⟨ *Node view model methods paint* 132c ⟩+ ≡

```
pixmap = QtGui.QPixmapCache.find(self.cache_key)
◇
```

Fragment defined by 132bc, 133, 214a.
Fragment referenced in 130b.

Figure 116: Based on the created or retrieved cache key a pixmap is being searched for.

Editor → Node view model → Methods → Paint

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

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

⟨ *Node view model methods paint* 133 ⟩+ ≡

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

Fragment defined by 132bc, 133, 214a.
Fragment referenced in 130b.

Figure 117: If no pixmap is found, a new pixmap is being created for the provided key and stored.

Editor → Node view model → Methods → Paint

Scene view

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

⟨ *Scene view declarations* 134a ⟩ ≡

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

    # Signals
    ⟨ Scene view signals 195c ⟩

    ⟨ Scene view constructor 134b, ... ⟩
    ⟨ Scene view methods 184a ⟩
    ⟨ Scene view slots 143b ⟩
◇
```

Fragment referenced in 195a.

Figure 118: Definition of the scene view component, derived from the QGraphicsView component.

Editor → Scene view

⟨ *Scene view constructor* 134b ⟩ ≡

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

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

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

Fragment defined by 134b, 183b.
Fragment referenced in 134a.

Figure 119: Constructor of the scene view component.

Editor → Scene view → Constructor

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

⟨ *Set up scene view in main window* 135 ⟩ ≡

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

Fragment referenced in 92.

Figure 120: The scene view component is being set up by the main window.

Editor → Main window → Methods → Setup UI

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

Connecting the view models of the scene graph view and the

⟨ *Add scene view to vertical splitter in main window* 136a ⟩ ≡

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

Fragment referenced in 92.

Figure 121: The scene view component is being added to the main window's vertical splitter.

Editor → Main window → Methods → Setup UI

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

⟨ *Connect controllers for main application* 136b ⟩ ≡

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

Fragment defined by 136bc.
Fragment referenced in 90b.

Figure 122: Whenever a scene is selected in the scene graph, the scene graph controller informs the scene controller about that selection.

Editor → Main application → Constructor

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

⟨ *Connect controllers for main application* 136c ⟩+ ≡

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

Fragment defined by 136bc.
Fragment referenced in 90b.

Figure 123: Whenever a scene is added to or removed from the scene graph, the scene graph controller informs the scene controller about those actions.

Editor → Main application → Constructor

THE SCENE VIEW MODELS REPRESENT A CERTAIN SCENE of the scene graph and hold the nodes of a specific scene. A scene view model is of type `QGraphicsScene`.

FOR BEING ABLE TO DISTINGUISH DIFFERENT SCENES, their identifier will be drawn at the top left position.

THE SCENE CONTROLLER DOES NOT DIRECTLY MANAGE SCENES.

⟨ *Scene controller declarations* 137a ⟩ ≡

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

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

    # Signals
    ⟨ Scene controller signals 143a, ... ⟩

    ⟨ Scene controller constructor 140 ⟩

    # Methods

    # Slots
    ⟨ Scene controller slots 141, ... ⟩
◇
```

Fragment referenced in 194b.

Figure 124: Definition of the scene controller.

Editor → Scene controller

⟨ *Set up controllers for main application* 137b ⟩+ ≡

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

Fragment defined by 105b, 137b, 181a.
Fragment referenced in 90b.

Figure 125: The scene controller being set up by the main application.

Editor → Main application → Constructor

⟨ *Scene view model declarations* 138a ⟩ ≡

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

    # Constants
    WIDTH = 15
    HEIGHT = 15

    # Signals

    ⟨ Scene view model constructor 138b ⟩
    ⟨ Scene view model methods 139, ... ⟩

    # Slots
◇
```

Fragment referenced in 193b.

Figure 126: Definition of the scene view model.

Editor → Scene view model

⟨ *Scene view model constructor* 138b ⟩ ≡

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

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

   super(SceneViewModel, self).__init__(parent)

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

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

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

Fragment referenced in 138a.

Figure 127: Constructor of the scene view model component.

Editor → Scene view model → Constructor

⟨ *Scene view model methods* 139 ⟩ ≡

```
def drawBackground(self, painter, rect):
    super(SceneViewModel, self).drawBackground(painter, rect)

    scene_rect = self.sceneRect()

    # Draw scene identifier
    text_rect = QtCore.QRectF(scene_rect.left()   + 4,
                              scene_rect.top()    + 4,
                              scene_rect.width()  - 4,
                              scene_rect.height() - 4)
    message = str(self)
    painter.save()
    font = painter.font()
    font.setBold(True)
    font.setPointSize(14)
    painter.setFont(font)
    painter.setPen(QtCore.Qt.lightGray)
    painter.drawText(text_rect.translated(2, 2), message)
    painter.setPen(QtCore.Qt.black)
    painter.drawText(text_rect, message)
    painter.restore()

    # Draw insert at marker
    width    = node_gui_domain.NodeViewModel.WIDTH
    height   = node_gui_domain.NodeViewModel.HEIGHT
    gradient = Qt.QLinearGradient(0, 0, width, 0)
    color    = self.palette().highlight().color()
    color.setAlpha(127)
    gradient.setColorAt(0, color)
    color.setAlpha(0)
    gradient.setColorAt(1, color)
    brush = QtGui.QBrush(gradient)
    painter.save()
    painter.translate(QtCore.QPoint(
        self.insert_at.x() * node_gui_domain.NodeViewModel.WIDTH,
        self.insert_at.y() * node_gui_domain.NodeViewModel.HEIGHT
    ))
    QtWidgets.qDrawPlainRect(
        painter, 0, 0, width + 1, height + 1, color, 0, brush
    )
    gradient.setColorAt(0, color)
    painter.setPen(QtGui.QPen(QtGui.QBrush(gradient), 0))
    painter.drawLine(0, 0, 0, height)
    painter.drawLine(0, 0, width, 0)
    painter.drawLine(0, height, width, height)
    painter.restore()

◇
```

Fragment defined by 139, 194a, 198a.
Fragment referenced in 138a.

Figure 128: The method to draw the background of a scene. It is used to draw the identifier of a scene at the top left position of it.

Editor → Scene view model → Methods

It has to react upon the signals sent by the scene graph controller. Additionally it needs to keep track of the currently selected scene, by holding a reference to that. The common identifier is the identifier of the domain model.

⟨ *Scene controller constructor* 140 ⟩ ≡

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

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

    super(SceneController, self).__init__(parent)

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

Fragment referenced in 137a.

Figure 129: Constructor of the scene controller. As can be seen, the scene controller holds all scenes (as a dictionary) and keeps track of the currently active scene.

Editor → Scene controller → Constructor

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

Whenever a scene is deleted, it needs to delete the scene from its known scenes as well.

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

As can be seen in fig. 132, the scene controller emits a signal that the scene was changed, containing the view model of the new scene.

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

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

⟨ *Scene controller slots* 141 ⟩ ≡

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

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

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

◇

Fragment defined by 141, 142ab, 197.
Fragment referenced in 137a.

Figure 130: The slot which gets triggered whenever a new scene is added via the scene graph.

Editor → Scene controller → Slots

⟨ *Scene controller slots* 142a ⟩+ ≡

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

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

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

Fragment defined by 141, 142ab, 197.
Fragment referenced in 137a.

Figure 131: The slot which gets triggered whenever a scene is removed via the scene graph.

Editor → Scene controller → Slots

⟨ *Scene controller slots* 142b ⟩+ ≡

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

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

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

Fragment defined by 141, 142ab, 197.
Fragment referenced in 137a.

Figure 132:

Editor → Scene controller → Slots

⟨ *Scene controller signals* 143a ⟩ ≡

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

Fragment defined by 143a, 196.
Fragment referenced in 137a.

Figure 133: The signal which is emitted when the scene has been changed by the scene graph controller and that scene is known to the scene controller.

Editor → Scene controller → Signals

⟨ *Scene view slots* 143b ⟩ ≡

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

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

Fragment referenced in 134a.

Figure 134: The slot of the scene view, which gets triggered whenever the scene changes. The scene interface, provided by QGraphicsView, is then invalidated to trigger the rendering of the scene view.

Editor → Scene view → Slots

⟨ *Connect main window components* 143c ⟩+ ≡

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

Fragment defined by 114b, 143c, 189b, 191b.
Fragment referenced in 90c.

Figure 135: The main application connects the scene view's signal that the scene was changed with the corresponding slot of the scene controller.

Editor → Main application → Constructor

Nodes

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

⟨ *Implicit sphere node* 144 ⟩ ≡

```
{
    "name": "Implicit sphere",
    "id_": "16d90b34-a728-4caa-b07d-a3244ecc87e3",
    "description": "Definition of a sphere by using implicit surfaces",
    "inputs": [
        ⟨ Implicit sphere node inputs 145a ⟩
    ],
    "outputs": [
        ⟨ Implicit sphere node outputs 145b ⟩
    ],
    "definitions": [
        ⟨ Implicit sphere node definitions 146a ⟩
    ],
    "invocations": [
        ⟨ Implicit sphere node invocations 146b ⟩
    ],
    "parts": [
        ⟨ Implicit sphere node parts 147 ⟩
    ],
    "nodes": [
    ],
    "connections": [
        ⟨ Implicit sphere node connections 148 ⟩
    ]
}◇
```

Fragment referenced in 210a.

Figure 136: Definition of a node for an implicitly defined sphere.

Implicit sphere node

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

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

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

⟨ *Implicit sphere node inputs* 145a ⟩ ≡

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

Fragment referenced in 144.

Figure 137: Radius of the implicit sphere node as input.

Implicit sphere node → Inputs

represents an implicit surface.

⟨ *Implicit sphere node outputs* 145b ⟩ ≡

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

Fragment referenced in 144.

Figure 138: The output of the implicit sphere node, which is of the atomic type implicit.

Implicit sphere node → Outputs

THE DEFINITION OF THE NODE IS THE ACTUAL IMPLEMENTATION of a sphere as a implicit surface.

THE INVOCATION OF THE NODE is simply calling the above definition using the parameters of the node, which is in this case the radius.

THE PARAMETERS ARE IN CASE OF IMPLICIT SURFACES uniform variables of the type of the parameter, as implicit surfaces are rendered by the fragment shader. The uniform variables are defined by a type and an identifier, whereas in the case of parameters their identifier is used.

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

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

⟨ *Implicit sphere node definitions* 146a ⟩ ≡

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

Fragment referenced in 144.

Figure 139: Implementation of the sphere in the OpenGL Shading Language (GLSL).

Implicit sphere node → Definitions

⟨ *Implicit sphere node invocations* 146b ⟩ ≡

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

Fragment referenced in 144.

Figure 140: The position of the implicit sphere node as invocation.

Implicit sphere node → Invocations

⟨ *Implicit sphere node parts* 147 ⟩ ≡

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

Fragment referenced in 144.

Figure 141: The "body" of the implicit sphere node as node part.

Implicit sphere node → Parts

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

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

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

⟨ *Implicit sphere node connections* 148 ⟩ ≡

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

Fragment referenced in 144.

Figure 142: Mapping of the connections of the implicit sphere node. Note that the inputs and outputs are internal, therefore the node references are 0.

Implicit sphere node → Connections

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

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

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

The node controller will then scan that directory containing the node definitions and load each one.

Node definitions will contain parts. The parts within a node definition are used to create corresponding parts within instances of themselves. The parts are able to create values based on the atomic types through functions.

The part of a node definition holds an identifier as well as an expression to create a function for creating and handling values

⟨ *Node controller declarations* 149a ⟩ ≡

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

    A controller managing nodes.
    """

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

    # Signals
    ⟨ Node controller signals 184b, ... ⟩

    ⟨ Node controller constructor 149b, ... ⟩
    ⟨ Node controller methods 150a, ... ⟩

    ⟨ Node controller slots 213 ⟩
◇
```

Fragment referenced in 210b.

Figure 143: Definition of the node controller.

Editor → Node controller

⟨ *Node controller constructor* 149b ⟩ ≡

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

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

    super(NodeController, self).__init__(parent)

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

Fragment defined by 149b, 151b, 158b, 177.
Fragment referenced in 149a.

Figure 144: Constructor of the node controller.

Editor → Node controller → Constructor

⟨ *Node controller methods* 150a ⟩ ≡

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

    ⟨ Node controller load nodes method  157a, ... ⟩◇
```

Fragment defined by 150a, 161, 167, 176.
Fragment referenced in 149a.

Figure 145: A method that loads node definitions from external files from within the node controller.

Editor → Node controller → Methods

⟨ *Node definition part domain model declarations* 150b ⟩ ≡

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

    # Signals

    ⟨ Node definition part domain model constructor  151a ⟩

    # Methods

    # Slots
◇
```

Fragment referenced in 207b.

Figure 146: Definition of a part of a node definition.

Editor → Node definition part

which will be used when evaluating a node. Further it provides a function which allows to instantiate itself as part of a node (instance).

⟨ *Node definition part domain model constructor* 151a ⟩ ≡

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

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

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

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

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

Fragment referenced in 150b.

Figure 147: Constructor of the node definition part.

Editor → Node definition part

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

⟨ *Node controller constructor* 151b ⟩+ ≡

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

Fragment defined by 149b, 151b, 158b, 177.
Fragment referenced in 149a.

Figure 148: The node controller keeps track of node definition parts.

Editor → Node controller → Constructor

The code snippet defining the constructor of a node definition part, fig. 147, uses a function called `create_value_function` of the `functions` module.

THAT BRINGS UP THE CONCEPT OF VALUE FUNCTIONS. Value functions are one of the building blocks of a node. They are used to evaluate a node value-wise through its inputs.

⟨ *Node domain module methods* 152 ⟩ ≡

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

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

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

    return value_function
◇
```

Fragment defined by 152, 172.
Fragment referenced in 207b.

Figure 149: Helper function which creates a value function from the given value.

Editor → Node domain model → Module methods

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

THE VALUE FUNCTION RELIES STRONGLY ON THE CONCEPT OF NODE PARTS, which is not defined yet. A part of a node is actually an instance of an atomic type (which is usually an input) within an instance of a node definition.

A PART OF A NODE HAS A FUNCTION, which gets called whenever a part of a node is being processed.

WHEN A PART OF A NODE IS BEING PROCESSED, also its inputs are processed. Whenever an input (value) changes, the node part needs to handle the changes. There are three possible types of changes: nothing has changed, the value (of the function) has changed or the sub tree (inputs) has changed.

FINALLY ALL NODES WILL BE COMPOSED OF ATOMIC TYPES. When building the node definition from the JSON input, the (atomic) part of the node definition is fetched from the node controller. Therefore it is necessary to provide parts for the atomic types before loading all the node definitions.

HAVING THE ATOMIC TYPES AVAILABLE AS PARTS, the node definitions themselves may be loaded. There is only one problem to that: there is nothing to hold the node definitions. Therefore the node definition domain model is introduced.

THE DEFINITION OF A NODE is quite similar to a node itself. As the

⟨ *Node part domain model value function declarations* 153 ⟩ ≡

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

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

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

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

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

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

        return new_function

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

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

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

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

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

        # TODO: Handle events

        return context◇
```

Fragment referenced in 155a.

Figure 150: Definition of the value function class which is used within nodes.

Editor → Value function

⟨ *Node part domain model default value function declarations* 154 ⟩ ≡

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

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

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

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

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

        return self

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

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

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

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

        return context◇
```

Fragment referenced in 155a.

Figure 151: Definition of the default value function class, which is derived from the value function class.

Editor → Default value function

⟨ *Node part domain model declarations* 155a ⟩ ≡

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

    ⟨ Node part domain model function declarations 156a ⟩
    ⟨ Node part domain model value function declarations 153 ⟩
    ⟨ Node part domain model default value function declarations 154 ⟩

    # Signals

    ⟨ Node part domain model constructor 155b ⟩

    # Methods

    # Slots
◇
```

Fragment referenced in 207b.

Figure 152: The node part class.

Editor → Node part

⟨ *Node part domain model constructor* 155b ⟩ ≡

```
  def __init__(self, id_, default_function,
               type_=types.NodeType.GENERIC, script=None):
    """constructor.

    :param id_: the identifier of the node part.
    :type  id_: uuid.uuid4
    :param default_function: the default function of the part.
    :type default_function: function
    :param type_: the type of the node part.
    :type type_: qde.editor.foundation.types.NodeType
    :param script: the script of the part.
    :type script: str
    """

    self.id_              = id_
    self.function_        = default_function
    self.default_function = default_function
    self.script           = script
    self.type_            = type_◇
```

Fragment referenced in 155a.

Figure 153: Constructor of the node part class.

Editor → Node part

⟨ *Node part domain model function declarations* 156a ⟩ ≡

```
class Function(object):
    """Represents the function of a part of a node."""

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

        self.has_changed = True
        self.evaluation_index = 0
        self.changed_state = \
            types.StateChange.VALUE.value | \
            types.StateChange.SUBTREE.value

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

        message = QtCore.QCoreApplication.translate(
            __class__.__name__,
            "This method must be implemented in a child class"
        )
        raise NotImplementedError(message)

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

        message = QtCore.QCoreApplication.translate(
            __class__.__name__,
            "This method must be implemented in a child class"
        )
        raise NotImplementedError(message)
◇
```

Fragment referenced in 155a.

Figure 154: Definition of the function class which is used in parts of nodes.

Editor → Function

⟨ *Node part state changed declarations* 156b ⟩ ≡

```
class StateChange(enum.Enum):
    """Possible changes of state."""

    NOTHING  = 0
    VALUE    = 1
    SUBTREE  = 2◇
```

Fragment referenced in 201.

Figure 155: A class which holds the possible values of a state change of a node part.

Editor → State change

⟨ *Node controller load nodes method* 157a ⟩ ≡

```
for atomic_type in parameter.AtomicTypes.atomic_types:
    if atomic_type.id_ not in self.node_definition_parts:
        self.node_definition_parts[atomic_type.id_] = atomic_type
        self.logger.info(
            "Added atomic type %s: %s",
            atomic_type.type_, atomic_type.id_
        )
    else:
        self.logger.warn((
            "Already knowing node part for atomic type %s."
            "This should not happen"
        ), atomic_type.type_)◇
```

Fragment defined by 157a, 159.
Fragment referenced in 150a.

Figure 156: The node controller provides the atomic types which build the basis of the part of a node.

Editor → Node controller → Methods → Load nodes

⟨ *Node definition domain model declarations* 157b ⟩ ≡

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

    # Signals

    ⟨ Node definition domain model constructor 158a ⟩
    ⟨ Node definition domain model methods 178, ... ⟩

    # Slots
◇
```

Fragment referenced in 207b.

Figure 157: Definition of the node definition class, which represents the definition of a node.

Editor → Node definition

definition of a node may be changed, the flag `was_changed` is added. Further a node definition holds all instances of itself, meaning nodes.

⟨ *Node definition domain model constructor* 158a ⟩ ≡

```
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.definitions = []
    self.invocations = []
    self.parts       = []
    self.nodes       = []
    self.connections = []
    self.instances   = []
    self.was_changed = False◇
```

Fragment referenced in 157b.

Figure 158: Constructor of the node definition class.

Editor → Node definition → Constructor

THE NODE CONTROLLER IS NOW ABLE to instantiate nodes definitions and keep them in a list. The controller manages both, the domain and the view models. As they both share the same ID, as the view model is being created from the data of the domain model, only one entry is necessary. The entry in the dictionary will therefore hold a tuple, containing the domain and the view model, identified by their common identifier.

⟨ *Node controller constructor* 158b ⟩+ ≡

```
    self.node_definitions = {}
◇
```

Fragment defined by 149b, 151b, 158b, 177.
Fragment referenced in 149a.

Figure 159: The node controller holds a dictionary containing node definitions.

Editor → Node controller → Constructor

THE NODE CONTROLLER SCANS the `node` sub directory, containing the node definitions, for files ending in `node`.

⟨ *Node controller load nodes method* 159 ⟩+ ≡

```
if os.path.exists(self.nodes_path):
    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)
    if num_node_definitions > 0:
        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)
            if node_definition is not None:
                node_definition_view_model =\
                node_gui_domain.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 162 ⟩

        t1 = time.perf_counter()
        self.logger.info(
            "Loading node definitions took %.10f seconds",
            (t1 - t0)
        )
    else:
        message = QtCore.QCoreApplication.translate(
            __class__.__name__, "No node definitions found."
        )
        self.logger.warn(message)
else:
    message = QtCore.QCoreApplication.translate(
        __class__.__name__, "No node definitions found."
    )
    self.logger.warn(message)
◇
```

Fragment defined by 157a, 159.
Fragment referenced in 150a.

Figure 160: The node controller loads and parses node definition files from the file system.

Editor → Node controller → Methods → Load nodes

If a file containing a node definition is found, its identifier is extracted from the file name. If the node definition is not known yet, it gets loaded and added to the list of known node definitions.

Whenever a new node definition gets loaded, other components need to be informed about the fact, that a new node definition is available. However, as the signal emits a view model, the loaded node definition cannot be emitted directly. Instead a view model needs to be created, which will then be emitted.

The loading of the node definition itself is simply about parsing the various sections and handling them correspondingly. To prevent the node controller from being bloated, the parsing is done in a separate module responsible for JSON specific tasks.

Not all parts of node definitions are defined yet: inputs, outputs, other node definitions, connections, definitions, invocations and parts.First the building of the node definition inputs is defined.

There are a few things missing, which are used in the above code fragment: the possibility to create values from given parameters, the actual node definition input as domain model and getting the node definition part identified by the given atomic identifier.

The creation of values from given parameters is done within the parameter module, as this is something very parameter specific. Therefore a static method is defined, which returns an instance of an atomic type, e.g. a float value or a scene.

For the specific value instances a generic value interface is defined. This interface holds a reference to the atomic type of the value and defines what type the function of a value is.

Then an interface for setting and getting values is defined.

Now the specific value types are implemented, based either on the generic or the concrete value interface, depending on the type. Here just two implementations are given as an example. The other implementations can be found at Appendix "Code fragments".

The definition of the input of a node definition is still missing however.

The code snippet defining the constructor of a node definition input uses a function called `create_default_value_function` of the `functions` module. This function creates a default value function based on the given default value.

With this last implementation all the parts needed for creating and handling node definition inputs are defined, which leads to the

⟨ *Node controller methods* 161 ⟩+ ≡

```
def load_node_definition_from_file_name(self, file_name):
    """Loads a node definition from the given file name.
    If no such file exists, None is returned.

    :param file_name: the file name to load.
    :type  file_name: str

    :return: the loaded node definition and its identifier or
             None
    :rtype:  qde.editor.domain.node.NodeDefinition or None
    """

    if not os.path.exists(file_name):
        self.logger.warn((
            "Tried to load node definition from file %s, "
            "but the file does not exist"
        ), file_name)
        return None

    # Extract the definition identifier from the file name,
    # which is "uuid4.node".
    definition_id = os.path.splitext(
        os.path.basename(file_name)
    )[0]

    if definition_id in self.node_definitions:
        self.logger.warn(
            ("Should load node definition from file %s,"
            "but is already loaded"),
            file_name
        )
        return self.node_definitions[definition_id]

    try:
        with open(file_name) as definition_fh:
            node_definition = json.Json.load_node_definition(
                self, definition_fh
            )
            self.logger.debug(
                "Loaded node definition %s from file %s",
                definition_id, file_name
            )
            # TODO: Trigger (loading) callback
            return node_definition
    except json.json.decoder.JSONDecodeError as exc:
        self.logger.warn(
            ("There was an error loading the node"
             "definition %s: %s"),
            definition_id, exc
        )
        return None◇
```

Fragment defined by 150a, 161, 167, 176.
Fragment referenced in 149a.

Figure 161: A method which tries to load a node definition from the file system using the provided file name.

Editor → Node controller → Methods → Load node definition from file name

⟨ *Node controller load node definition emit* 162 ⟩ ≡

```
self.do_add_node_view_definition.emit(node_definition_view_model)
◇
```

Fragment referenced in 53, 159.

Figure 162: Whenever a new node definition gets loaded, the node controller emits a corresponding signal containing the node view model for the loaded node definition.

Editor → Node controller → Methods → Load node definitions

next implementation. The outputs of a node definition. The outputs are in the same way implemented as the inputs of a node definition.

THE DOMAIN MODEL OF THE NODE DEFINITION OUTPUT is very similar to the input, has less attributes although.

A NODE DEFINITION MAY CONTAIN REFERENCES to other node definitions, therefore it is necessary to parse them. The parsing is similar to that of the inputs and outputs.

As can be seen in the above code fragment, the node definition is returned by the node controller. This is very similar to getting the node definition part from the node controller.

THE NODE CONTROLLER HOLDS A REFERENCE TO THE ROOT NODE of the root scene of the system. This scene acts as an entry point when evaluating the scene graph.

CURRENTLY THERE IS NO POSSIBILITY TO ADD OUTPUTS to a node definition. Adding an output simply adds that output to the list of outputs the node definition has. Furthermore that output needs to added for each instance of that node definition as well.

Having the reading and parsing of inputs, outputs and other node definition implemented, the reading and parsing of connections, definitions, invocations and parts still remains.

THE READING AND PARSING of connections, definitions and invocation is very straightforward and very similar to the one of the node definitions. Therefore it will not be shown in detail. Details are found at Appendix "Code fragments".

THE LAST PART WHEN LOADING A NODE DEFINITION is reading and parsing the code part of the node.

FINALLY THE NODE CONTROLLER NEEDS TO BE INSTANTIATED by the main application and the loading of the node definitions needs to be triggered. The loading may although not be triggered at the same place as the signals for reacting upon new node definitions need to be in place first.

LOADING OF NODE DEFINITIONS is done right before the main window is shown, as at that point all necessary connections between

⟨ *JSON methods* 163 ⟩ ≡

```
classmethod
def load_node_definition(cls, node_controller, json_file_handle):
    """Loads a node definition from given JSON input.

    :param node_controller: reference to the node controller
    :type node_controller: qde.editor.application.node.\
                           NodeController
    :param json_file_handle: an open file handle containing
                             JSON data
    :type json_file_handle: file

    :return: a node definition
    :rtype: qde.editor.domain.node.NodeDefinition
    """

    o = json.load(json_file_handle)

    name        = str(o['name'])
    id_         = uuid.UUID(o['id_'])
    description = str(o['description'])

    ⟨ Parse node definition inputs 164a ⟩
    ⟨ Parse node definition outputs 164b ⟩
    ⟨ Parse node definition children 164c ⟩
    ⟨ Parse node definition connections 164d ⟩
    ⟨ Parse node definition definitions 165a ⟩
    ⟨ Parse node definition invocations 165b ⟩

    node_definition             = node.NodeDefinition(id_)
    node_definition.name        = name
    node_definition.description = description
    node_definition.inputs      = inputs
    node_definition.outputs     = outputs
    node_definition.nodes       = node_definitions
    node_definition.connections = connections
    node_definition.definitions = definitions
    node_definition.invocations = invocations

    # TODO: Check if this part can be above the def. instance
    ⟨ Parse node definition parts 165c ⟩
    node_definition.parts = parts

    # TODO: Do a consistency check
    node_definition.was_changed = False

    return node_definition◇
```

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

Figure 163: A class method of the JSON module, which loads a node definition from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Load node definition

⟨ *Parse node definition inputs* 164a ⟩ ≡

```
inputs = []
for input in o['inputs']:
    node_definition_input = cls.build_node_definition_input(
        node_controller, input
    )
    inputs.append(node_definition_input)◇
```

Fragment referenced in 163.

Figure 164: The inputs of a node definition are parsed and then added to the list of known inputs.

Editor → JSON → Methods → Load node definition

⟨ *Parse node definition outputs* 164b ⟩ ≡

```
outputs = []
for output in o['outputs']:
    node_definition_output = cls.build_node_definition_output(
        node_controller, output
    )
    outputs.append(node_definition_output)◇
```

Fragment referenced in 163.

Figure 165: The outputs of a node definition are parsed and then added to the list of known outputs.

Editor → JSON → Methods → Load node definition

⟨ *Parse node definition children* 164c ⟩ ≡

```
node_definitions = {}
for node_def in o['nodes']:
    definition_id, node_definition = cls.build_node_definition(
        node_def
    )
    node_definitions[definition_id] = node_definition◇
```

Fragment referenced in 163.

Figure 166: The child nodes of a node definition are parsed and then added to the list of known child nodes.

Editor → JSON → Methods → Load node definition

⟨ *Parse node definition connections* 164d ⟩ ≡

```
connections = []
for conn in o['connections']:
    connection = cls.build_node_definition_connection(conn)
    connections.append(connection)◇
```

Fragment referenced in 163.

Figure 167: The connections of a node definition are parsed and then added to the list of known connections.

Editor → JSON → Methods → Load node definition

⟨ *Parse node definition definitions* 165a ⟩ ≡

```
definitions = []
for d in o['definitions']:
    definition = cls.build_node_definition_definition(d)
    definitions.append(definition)◇
```

Fragment referenced in 163.

Figure 168: The definitions of a node definition are parsed and then added to the list of known definitions.

Editor → JSON → Methods → Load node definition

⟨ *Parse node definition invocations* 165b ⟩ ≡

```
invocations = []
for i in o['invocations']:
    invocation = cls.build_node_definition_invocation(i)
    invocations.append(invocation)◇
```

Fragment referenced in 163.

Figure 169: The invocations of a node definition are parsed and then added to the list of known invocations.

Editor → JSON → Methods → Load node definition

⟨ *Parse node definition parts* 165c ⟩ ≡

```
parts = []
for p in o['parts']:
    part = cls.build_node_definition_part(
        node_controller, node_definition, p
    )
    parts.append(part)◇
```

Fragment referenced in 163.

Figure 170: The parts of a node definition are parsed and then added to the list of known parts.

Editor → JSON → Methods → Load node definition

⟨ *JSON methods 166* ⟩+ ≡

```
classmethod
def build_node_definition_input(cls, node_controller, json_input):
    """Builds and returns a node definition input from the given
    JSON input data.

    :param node_controller: a reference to the node controller
    :type  node_controller: qde.editor.application.node.NodeController
    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: a node definition input
    :rtype:  qde.editor.domain.node.NodeDefinitionInput
    """

    input_id             = uuid.UUID(json_input['id_'])
    name                 = str(json_input['name'])
    atomic_id            = uuid.UUID(json_input['atomic_id'])
    description          = str(json_input['description'])
    node_definition_part = node_controller.\
        get_node_definition_part(
                atomic_id
    )

    default_value_str = ""
    default_value_entry = json_input['default_value']
    default_value = parameter.create_value(
        default_value_entry['type_'],
        default_value_entry['value']
    )

    min_value = float(json_input['min_value'])
    max_value = float(json_input['max_value'])

    node_definition_input = node.NodeDefinitionInput(
        input_id,
        name,
        node_definition_part,
        default_value
    )
    node_definition_input.description = description
    node_definition_input.min_value = min_value
    node_definition_input.max_value = max_value

    cls.logger.debug(
        "Built node definition input for node definition %s",
        atomic_id
    )
    return node_definition_input
◇
```

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

Figure 171: A class method of the JSON module, which builds the input of a node definition from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition input

⟨ *Node controller methods* 167 ⟩+ ≡

```
def get_node_definition_part(self, id_):
    """Returns the node definition part identified by the given
    identifier.

    If no such part is available, a generic part with that
    identifier is being created.

    :param id_: the identifier of the part of the node definition
                to get.
    :type  id_: uuid.uuid4

    :return: the node definition part identified by the given
             identifier.
    :rtype: qde.editor.domain.node.NodeDefinitionPart
    """

    if str(id_) not in self.node_definition_parts:
        self.logger.warn((
            "Part %s of the node definition was not found."
            "Creating a generic one."
        ), id_)

        type_ = types.NodeType.GENERIC
        def create_func(id_, default_function, name, type_):
            node_part = node.NodePart(id_, None)
            node_part.type_ = type_
            node_part.name = name
            return node_part
        node_definition_part = node.NodeDefinitionPart(id_)
        node_definition_part.type_ = type_
        node_definition_part.creator_function = create_func
        self.node_definition_parts[id_] = node_definition_part
        return node_definition_part
    else:
        return self.node_definition_parts[str(id_)]
```

◇

Fragment defined by 150a, 161, 167, 176.
Fragment referenced in 149a.

Figure 172: A method of the node controller, which returns a node definition part by a provided identifier. If no node definition part is found for the given identifier, a new node definition part is created.

Editor → Node controller → Methods → Get node definition part

⟨ *Parameter domain module methods* 168 ⟩ ≡

```
def create_value(type_, value_string):
    """Creates an object of the given type with the given value.

    :param type_: the type of the value to create.
    :type  type_: str
    :param value_string: the value that the value shall have.
    "type  value_string: str

    :return: a value-type of the given type with the given value.
    :rtype: qde.editor.domain.parameter.Value
    """

    if type_.lower() == "float":
        float_value = float(value_string)
        return FloatValue(float_value)
    elif type_.lower() == "text":
        return TextValue(value_string)
    elif type_.lower() == "image":
        return ImageValue()
    elif type_.lower() == "scene":
        return SceneValue()
    elif type_.lower() == "generic":
        return GenericValue()
    elif type_.lower() == "dynamic":
        return DynamicValue()
    elif type_.lower() == "mesh":
        return MeshValue()
    elif type_.lower() == "implicit":
        return ImplicitValue()
    else:
        message = QtCore.QCoreApplication.translate(
            __module__.__name__, "Unknown type for value provided"
        )
        raise Exception(message)◇
```

Fragment referenced in 202.

Figure 173: Method of the parameter module, which creates an object of a specific value instance based on the provided type of the value.

Editor → Parameter → Create value

⟨ *Parameter domain model value generic interface* 169a ⟩ ≡

```
class ValueInterface(object):
    """Generic value interface."""

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

        self.function_type = None

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        message = QtCore.QCoreApplication.translate(
            __module__.__name__,
            "This method must be implemented in a child class"
        )
        raise NotImplementedError(message)◇
```

Fragment referenced in 202.

Figure 174: Interface as basis for the value specific instances.

Editor → Parameter → Value interface

⟨ *Parameter domain model value interface* 169b ⟩ ≡

```
class Value(ValueInterface):
    """Value interface for setting and getting values."""

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

        :param value: the value that shall be held
        :type  value: object
        """

        super(Value, self).__init__()
        self.value = value◇
```

Fragment referenced in 202.

Figure 175: Class which provides an interface to the value of the value specific instances.

Editor → Parameter → Value

⟨ *Parameter domain model float value* 170a ⟩ ≡

```
class FloatValue(Value):
    """A class holding float values."""

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

        :param float_value: the float value that shall be held
        :type  float_value: float
        """

        super(FloatValue, self).__init__(float_value)
        self.function_type = types.NodeType.FLOAT

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return FloatValue(self.value)◇
```

Fragment referenced in 202.

Figure 176: Implementation of the float value type.

Editor → Parameter → FloatValue

⟨ *Parameter domain model scene value* 170b ⟩ ≡

```
class SceneValue(ValueInterface):
    """A class holding scene values."""

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

        super(SceneValue, self).__init__()
        self.function_type = types.NodeType.SCENE

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return SceneValue()◇
```

Fragment referenced in 202.

Figure 177: Implementation of the scene value type.

Editor → Parameter → SceneValue

⟨ *Node definition input domain model declarations* 171a ⟩ ≡

```
class NodeDefinitionInput(object):
    """Represents an input of a definition of a node."""

    # Signals

    ⟨ Node definition input domain model constructor 171b ⟩

    # Methods

    # Slots
◇
```

Fragment referenced in 207b.

Figure 178: Implementation of the input of the definition of a node.

Editor → Node definition input

⟨ *Node definition input domain model constructor* 171b ⟩ ≡

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

    :param id_: the identifier of the definition
    :type  id_: uuid.uuid4
    :param name: the name of the definition
    :type  name: str
    :param node_definition_part: the atomic part of the
                                 node definition
    :type node_definition_part: TODO
    :param default_value: the default value of the input
    :type default_value: qde.editor.domain.parameter.Value
    """

    self.id_                 = id_
    self.name                = name
    self.node_definition_part = node_definition_part
    self.description         = ""
    self.min_value           = -100000
    self.max_value           = 100000

    self.default_function = create_default_value_function(
        default_value
    )◇
```

Fragment referenced in 171a.

Figure 179: Constructor of the input of the definition of a node.

Editor → Node definition input → Constructor

⟨ *Node domain module methods* 172 ⟩+ ≡

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

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

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

    return value_function◇
```

Fragment defined by 152, 172.
Fragment referenced in 207b.

Figure 180: Function that creates a default value function based on a provided value.

Editor → Node → Methods → Create default value function

signals and slots are in place.

NOW NODE DEFINITIONS ARE BEING LOADED AND PARSED. Although there is no possibility to select and instantiate the node definitions yet. To allow the instantiation of nodes, a (user interface) component is necessary: A dialog for adding nodes to the currently active scene. It will access all the loaded nodes and provide an interface for selecting a node definition which then will be instantiated.

THE KEY IDEA OF THE ADD NODE DIALOG is to have multiple columns where each column defines a specific node type. The node definitions of each type are then vertically listed per column. As these columns are tightly tied to the add node dialog, the declaration of the column class is part of the add node dialog.

THE DIALOG FOR ADDING A NODE INSTANCE from a node definition shall only be shown from within a scene, that is from within the scene view. Therefore the add node dialog is added to the scene view.

WHENEVER THE SCENE VIEW IS FOCUSED and the tabulator key is being pressed, the dialog for adding a node shall be shown. For achieving this, the `event` method of the scene view needs to be overwritten.

PRESSING THE TABULATOR KEY when the scene view is active, brings up the dialog to add a node, but the dialog is empty. This is due to the circumstance, that the node controller is not informing whenever he receives a new node definition and that no other component is listening.

THE NODE CONTROLLER HAS TO EMIT a signal whenever he reads

⟨ *JSON methods* 173 ⟩+ ≡

```
classmethod
def build_node_definition_output(cls, node_controller, json_input):
    """Builds and returns a node definition output from the given
    JSON input data.

    :param node_controller: a reference to the node controller
    :type  node_controller: qde.editor.application.node.NodeController
    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: a node definition output
    :rtype:  qde.editor.domain.node.NodeDefinitionOutput
    """

    output_id             = uuid.UUID(json_input['id_'])
    name                  = str(json_input['name'])
    atomic_id             = uuid.UUID(json_input['atomic_id'])
    node_definition_part = node_controller.get_node_definition_part(
        atomic_id
    )

    node_definition_output = node.NodeDefinitionOutput(
        output_id,
        name,
        node_definition_part
    )

    cls.logger.debug(
        "Built node definition output for node definition %s",
        atomic_id
    )
    return node_definition_output
◇
```

Figure 181: A class method of the JSON module, which builds the output of a node definition from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition output

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

⟨ *Node definition output domain model declarations* 174a ⟩ ≡

```
class NodeDefinitionOutput(object):
    """Represents an output of a definition of a node."""

    # Signals

    ⟨ Node definition output domain model constructor 174b ⟩
    ⟨ Node definition output domain model methods 214b ⟩

    # Slots
◇
```

Fragment referenced in 207b.

Figure 182: Implementation of the output of the definition of a node.

Editor → Node definition output

⟨ *Node definition output domain model constructor* 174b ⟩ ≡

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

    :param id_: the identifier of the definition
    :type  id_: uuid.uuid4
    :param name: the name of the definition
    :type  name: str
    :param node_definition_part: the atomic part of the node
                                 definition
    :type node_definition_part: qde.editor.domain.node.\
                                NodeDefinitionPart
    """

    self.id_                = id_
    self.name               = name
    self.node_definition_part = node_definition_part◇
```

Fragment referenced in 174a.

Figure 183: Constructor of the output of the definition of a node.

Editor → Node definition input → Constructor

⟨ *JSON methods* 175 ⟩+ ≡

```
classmethod
def build_node_definition(cls, node_controller, json_input):
    """Builds and returns a node definition from the given JSON
    input data.

    :param node_controller: a reference to the node controller
    :type  node_controller: qde.editor.application.node.\
                            NodeController
    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: a dictionary containing the node definition at the
             index of the definition identifier.
    :rtype:  dict
    """

    definition_id   = uuid.UUID(json_input['id_'])
    atomic_id       = uuid.UUID(json_input['atomic_id'])

    node_definition, node_view_model = node_controller.\
        get_node_definition(
                atomic_id
    )

    cls.logger.debug(
        "Built node definition for node definition %s",
        atomic_id
    )
    return (definition_id, node_definition)
◇
```

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

Figure 184: A class method of the JSON module, which builds the definition of a node from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition

⟨ *Node controller methods* 176 ⟩+ ≡

```
def get_node_definition(self, id_):
    """Returns the node definition identified by the given
    identifier.

    If no such definition is available, it will be tried to load
    the definition. If this is not possible as well, None will
    be returned.

    :param id_: the identifier of the node definition to get.
    :type  id_: uuid.uuid4

    :return: the node definition identified by the given
             identifier or None.
    :rtype:  qde.editor.domain.node.NodeDefinition or None
    """

    self.logger.debug(
        "Getting node definition %s",
        id_
    )

    if str(id_) in self.node_definitions:
        return self.node_definitions[str(id_)]
    elif self.root_node is not None and id_ == self.root_node.id_:
        return self.root_node
    else:
        # The node definition was not found, try to load it
        # from node definition files.
        file_name = os.path.join(
            self.nodes_path,
            id_,
            self.nodes_extension
        )
        node_definition = self.\
            load_node_definition_from_file_name(
                    file_name
        )
        if node_definition is not None:
            node_definition_view_model = node_gui_domain.\
                NodeViewModel(
                        id_=node_definition.id_,
                        domain_object=node_definition
                )
            self.node_definitions[node_definition.id_] = (
                node_definition,
                node_view_model
            )
            return (node_definition, node_view_model)
        else:
            return None◇
```

Fragment defined by 150a, 161, 167, 176.
Fragment referenced in 149a.

Figure 185: A method of the node controller, which returns a node definition by a provided identifier. If no node definition is found for the given identifier, a new node definition is created by loading the definition from the file system.

Editor → Node controller → Methods → Get node definition

⟨ *Node controller constructor* 177 ⟩+ ≡

```
    # TODO: Load from configuration?
    self.root_node = node_domain.NodeDefinition(
        NodeController.ROOT_NODE_ID
    )
    self.root_node.name = QtCore.QCoreApplication.translate(
        __class__.__name__,
        'Root'
    )
    root_node_output = node_domain.NodeDefinitionOutput(
        NodeController.ROOT_NODE_OUTPUT_ID,
        QtCore.QCoreApplication.translate(
            __class__.__name__,
            'Output'
        ),
        parameter.AtomicTypes.Generic
    )
    self.root_node.add_output(root_node_output)
    self.logger.debug(
        "Created root node %s",
        NodeController.ROOT_NODE_ID
    )◇
```

Fragment defined by 149b, 151b, 158b, 177.
Fragment referenced in 149a.

Figure 186: The root node of the system is manually created by the node controller and is also a node definition.

Editor → Node controller → Constructor

a new node definition. The signal itself is emitting a view model of the read node definition.

Now other components may listen and receive view models of newly added node definitions. In this specific case it is is the dialog for adding a node which needs to listen to the added signal. The listening is done by the slot `on_node_definition_added`.

As the idea of the dialog is to have one column per node type, the column needs to be fetched first, based on the type name of the given node definition. Then a sub frame is created which holds a representation of the node definition. This representation is rendered like an actual instance of a node. Its behavior is like a button, meaning it can be clicked. Clicking on a representation of a node definition adds an instance of the clicked node definition to the currently active scene at the cursor position where the dialog for adding a node was opened.

A node definition may already be present although. If this is the case the process will be stopped.

Getting or creating a column is about calling the corresponding method, as the task is abstracted into a method to maintain read-

⟨ *Node definition domain model methods 178* ⟩ ≡

```
def add_output(self, node_definition_output):
    """Adds the given output to the beginning of the list of
outputs and also to all instances of this node definition.

    :param node_definition_output: the output to add.
    :type  node_definition_output: qde.editor.domain.node.\
                          NodeDefinitionOutput
    """

    self.add_output_at(len(self.outputs), node_definition_output)

def add_output_at(self, index, node_definition_output):
    """Adds the given output to the list of outputs at the given
    index position and also to all instances of this node
    definition.

    :param index: the position in the list of outputs where the
                  new output shall be added at.
    :type  index: int
    :param node_definition_output: the output to add.
    :type  node_definition_output: qde.editor.domain.node.\
                                   NodeDefinitionOutput

    :raise: an index error when the given index is not valid.
    :raises: IndexError
    """

    if index < 0 or index > len(self.outputs):
        raise IndexError()

    self.outputs.insert(index, node_definition_output)

    for instance in self.instances:
        instance.add_output_at(
            index,
            node_definition_output.create_instance()
        )

    # TODO: Insert connection if output is atomic

    self.was_changed = True◇
```

Fragment defined by 178, 179.
Fragment referenced in 157b.

Figure 187: Methods which add a given output of a node definition to a node definition. The first method adds the output at the end of the list of outputs, the second adds the output at the given index.

Editor → Node definition → Methods

⟨ *Node definition domain model methods* 179 ⟩+ ≡

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

    type_ = types.NodeType.GENERIC

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

    return type_◇
```

Fragment defined by 178, 179.
Fragment referenced in 157b.

Figure 188: Type property of a node definition. If the node definition uses outputs, the type is derived by its primary output. Otherwise a generic type is assumed.

Editor → Node definition → Methods

ability.

The method to get a column, `get_or_create_column_by_name`, tries to get a column by the given name and if no column by that name exists, it creates a new column using the given name.

Therefore, if a column by the given name already exists, the reference to the found column is returned.

If no column by the given name exists, a new column using the given name is created.

For adding the representation of the node definition to a column, the creation of a sub frame is necessary.

The node representation is then created and added to the above created sub frame. At this moment the presentation is simply a label.

On thing that stands out in the above code fragment, is the clickable label class. This label is nothing other than normal label emitting a signal called `clicked` when receiving a mouse press event.

For being able to react whenever such a label is clicked, it is necessary to handle the `clicked` signal of the label. Up to now all signals emitted the necessary objects. As the `clicked` signal is very generic, it does not emit an object. It is nevertheless necessary to emit the chosen node definition.

Finally the created sub frame is added to the found or created

⟨ *JSON methods* 180 ⟩+ ≡

```
classmethod
def build_node_definition_part(cls, node_controller, parent, json_input):
    """Builds and returns a node definition part from the given JSON
    input data.

    :param node_controller: a reference to the node controller
    :type  node_controller: qde.editor.application.node.NodeController
    :param parent: the parent of the node definition part
    :type  parent: qde.editor.domain.node.NodeDefinition
    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: the built part of the node definition
    :rtype:  qde.editor.domain.node.NodeDefinitionPart
    """

    part_id         = uuid.UUID(json_input['id_'])
    name            = str(json_input['name'])

    script_lines = []
    for script_line in json_input['script']:
        script_lines.append(str(script_line))
    script = "\n".join(script_lines)

    type_string = json_input['type_']
    type_ = types.NodeType[type_string.upper()]

    node_definition_part = node.NodeDefinitionPart(part_id)
    node_definition_part.name = name
    node_definition_part.type_ = type_
    node_definition_part.script = script
    node_definition_part.parent = parent

    node_controller.node_definition_parts[part_id] =\
        node_definition_part

    cls.logger.debug(
        "Built part for node definition %s",
        part_id
    )
    return node_definition_part
◇
```

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

Figure 189: A class method of the JSON module, which builds a part of the definition of a node from a file handle (pointing to a JSON file containing a node definition).

Editor → JSON → Methods → Build node definition part

⟨ *Set up controllers for main application* 181a ⟩+ ≡

```
1 self.node_controller = node.NodeController()◇
```

Fragment defined by 105b, 137b, 181a.
Fragment referenced in 90b.

Figure 190: Instantiation of the node controller from within the main application.

Editor → Main application → Constructor

⟨ *Load nodes* 181b ⟩ ≡

```
1 self.node_controller.load_nodes()◇
```

Fragment referenced in 88a.

Figure 191: Loading of nodes is triggered by the main application right after instantiating the node controller.

Editor → Main application → Constructor

column.

If the node definition is not yet known, it is saved to the list of known node definitions. Otherwise a warning is being shown.

The above defined slot needs to be triggered as soon as a new node definition is being added. This is done within the main window, by connecting the slot with the `do_add_node_view_definition` signal.

⟨ *Add node dialog declarations* 182 ⟩ ≡

```
common.with_logger
class AddNodeDialog(QtWidgets.QDialog):
    """Class for adding nodes to a scene view."""

    # Signals

    ⟨ Add node dialog column declaration 183a ⟩

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

        :param parent: the parent of this dialog.
        :type  parent: QtGui.QWidget
        """

        super(AddNodeDialog, self).__init__(parent)

        self.columns               = {}
        self.node_definitions      = {}
        self.chosen_node_definition = None

        self.setFixedSize(parent.width(), parent.height())
        self.setWindowTitle("Add node")

        layout = QtWidgets.QHBoxLayout(self)
        layout.setContentsMargins(10, 10, 10, 10)
        layout.setSizeConstraint(Qt.QLayout.SetFixedSize)
        self.setLayout(layout)

    ⟨ Add node dialog methods 186a ⟩

    # Slots
    ⟨ Add node dialog slots 185a ⟩
◇
```

Fragment referenced in 212b.

Figure 192: Definition of a dialog to add nodes to the currently active scene. The nodes are ordered in columns according to their type.

Editor → Add node dialog

⟨ *Add node dialog column declaration* 183a ⟩ ≡

```
class Column(object):
    """Class representing a column within the add node dialog."""

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

        self.frame         = None
        self.sub_frames    = []
        self.label         = None
        self.v_box_layout  = None
◇
```

Fragment referenced in 182.

Figure 193: Class representing column within the dialog to create new node instances.

Editor → Add node dialog → Column

⟨ *Scene view constructor* 183b ⟩+ ≡

```
    self.add_node_dialog = node.AddNodeDialog(self.parent())
◇
```

Fragment defined by 134b, 183b.
Fragment referenced in 134a.

Figure 194: The dialog for adding new node instances is initialized by the scene view.

Editor → Scene view → Constructor

⟨ *Scene view methods* 184a ⟩ ≡

```
def event(self, event):
    if (
            event.type() == Qt.QEvent.KeyPress and
            event.key()  == QtCore.Qt.Key_Tab
    ):
        self.logger.debug("Tabulator was pressed")

        # Sanity check: Open the dialog only if it is not
        # opened already.
        if not self.add_node_dialog.isVisible():
            current_scene = self.scene()
            assert current_scene is not None
            insert_at = current_scene.insert_at
            self.logger.debug("Cursor at %s", insert_at)
            insert_position = QtCore.QPoint(
                insert_at.x() * node_gui_domain.NodeViewModel.WIDTH,
                insert_at.y() * node_gui_domain.NodeViewModel.HEIGHT
            )
            insert_position = self.mapToGlobal(self.mapFromScene(
                insert_position
            ))
            self.add_node_dialog.move(insert_position)
            add_dialog_result = self.add_node_dialog.exec()

            # At this point we are sure, that this dialog instance
            # was handled properly, so accepting the event might
            # be sane here.
            event.accept()

            if add_dialog_result == QtWidgets.QDialog.Accepted:
                ⟨ Handle node definition chosen 195d ⟩
                return True
            else:
                return False

    return super(SceneView, self).event(event)
◇
```

Fragment referenced in 134a.

Figure 195: The event method of the scene view is overwritten for being able to show the dialog for adding new instances of nodes when the tabulator key is pressed.

Editor → Scene view → Methods

⟨ *Node controller signals* 184b ⟩ ≡

```
do_add_node_view_definition = QtCore.pyqtSignal(node_gui_domain.NodeViewModel)◇
```

Fragment defined by 184b, 212c.
Fragment referenced in 149a.

Figure 196: The signal of the node controller that is emitted whenever a node definition was read.

Editor → Node controller → Signals

⟨ *Add node dialog slots* 185a ⟩ ≡

```
QtCore.pyqtSlot(node_gui_domain.NodeViewModel)
def on_node_definition_added(self, node_view_model):
    """Slot which is called whenever a new node definition is added.

    :param node_view_model: The newly added node definition.
    :type  node_view_model: qde.editor.gui_domain.node.NodeDefinitionViewModel
    """

    self.logger.debug("Got new node definition: %s", node_view_model)

    node_name = node_view_model.domain_object.name
    type_name = node_view_model.domain_object.type_.name
    ⟨ On node definition added implementation 185b ⟩◇
```

Fragment referenced in 182.

Figure 197: The slot of the dialog to add a new node that is called whenever a new node definition is added.

Editor → Add node dialog → Slots

⟨ *On node definition added implementation* 185b ⟩ ≡

```
⟨ Check if the node definition is already known 185c ⟩
⟨ Get or create column by type name 185d ⟩
⟨ Create sub frame for given node definition 187b ⟩
⟨ Create button for given node definition and add to sub frame 188a, ... ⟩
⟨ Add sub frame to column 188c ⟩
⟨ Save the node definition to list of known nodes 189a ⟩◇
```

Fragment referenced in 185a.

Figure 198: Implementation of the slot of the dialog to add a new node that is called whenever a new node definition is added.

Editor → Add node dialog → Slots

⟨ *Check if the node definition is already known* 185c ⟩ ≡

```
if node_view_model.id_ not in self.node_definitions:
◇
```

Fragment referenced in 185b.

Figure 199: It is checked that the given node definition is not already known.

Editor → Add node dialog → Slots

⟨ *Get or create column by type name* 185d ⟩ ≡

```
    column = self.get_or_create_column_by_name(type_name)◇
```

Fragment referenced in 185b.

Figure 200: The column which the node definition belongs to is depending on the node definition's type. If a column for the type of the node definition already exists, that column is used. Otherwise a new column is created for the type.

Editor → Add node dialog → Slots

⟨ *Add node dialog methods* 186a ⟩ ≡

```
def get_or_create_column_by_name(self, column_name):
    """Gets the column for the given column name.
    If there is no column for the given column name available,
    a new column using the given column name is created.

    :param column_name: the name of the column to get or create.
    :type  column_name: str

    :return: the column for the given column name.
    :rtype:  AddNodeDialog.Column
    """

    ⟨ Get existing column object by name 186b ⟩
    ⟨ Create new column object based on name 187a ⟩

    return column◇
```

Fragment referenced in 182.

Figure 201: The method for getting or creating a column by type name.

Editor → Add node dialog → Methods

⟨ *Get existing column object by name* 186b ⟩ ≡

```
if column_name in self.columns:
    column = self.columns[column_name]◇
```

Fragment referenced in 186a.

Figure 202: If a column by the given name exists that column is returned.

Editor → Add node dialog → Methods

⟨ *Create new column object based on name* 187a ⟩ ≡

```
else:
    frame = QtWidgets.QFrame(self)
    self.layout().addWidget(frame)
    frame.setContentsMargins(0, 0, 0, 0)

    row = QtWidgets.QVBoxLayout(frame)
    row.setContentsMargins(0, 0, 0, 0)

    caption = "<h2>{0}</h2>".format(column_name)
    label = QtWidgets.QLabel(caption, frame)
    label.setContentsMargins(4, 2, 4, 2)
    label_font = QtGui.QFont()
    label_font.setFamily(label_font.defaultFamily())
    label_font.setBold(True)
    label_font.setUnderline(True)
    label.setFont(label_font)

    row.addWidget(label)
    row.addStretch(1)

    column = AddNodeDialog.Column()
    column.frame = frame
    column.label = column_name
    column.v_box_layout = row
    self.columns[column_name] = column◇
```

Fragment referenced in 186a.

Figure 203: When no column for a specific name exists the column is created.

Editor → Add node dialog → Methods

⟨ *Create sub frame for given node definition* 187b ⟩ ≡

```
    sub_frame = QtWidgets.QFrame(column.frame)
    sub_frame_column = QtWidgets.QHBoxLayout(sub_frame)
    sub_frame_column.setContentsMargins(0, 0, 0, 0)
    sub_frame_column.setSpacing(0)◇
```

Fragment referenced in 185b.

Figure 204: A sub frame is created for each new column.

Editor → Add node dialog → Methods

⟨ *Create button for given node definition and add to sub frame* 188a ⟩ ≡

```
    button_label = gui_helper.ClickableLabel(node_name, sub_frame)
    button_label.setContentsMargins(4, 0, 4, 0)
    button_label.setSizePolicy(
        Qt.QSizePolicy.Expanding, Qt.QSizePolicy.Preferred
    )
    sub_frame_column.addWidget(button_label)◇
```

Fragment defined by 188ab.
Fragment referenced in 185b.

Figure 205: For the type of the given node definition a button is created and added to the previously created sub frame.

Editor → Add node dialog → Methods

⟨ *Create button for given node definition and add to sub frame* 188b ⟩+ ≡

```
    def _add_node_button_clicked(node_view_model):
        self.chosen_node_definition = node_view_model
        self.accept()

    button_label.do_add_node.connect(functools.partial(
        _add_node_button_clicked, node_view_model
    ))◇
```

Fragment defined by 188ab.
Fragment referenced in 185b.

Figure 206: A callback function is created which gets triggered whenever a button is clicked. The callback function stores the selected node definition (type) and closes the dialog.

Editor → Add node dialog → Methods

⟨ *Add sub frame to column* 188c ⟩ ≡

```
    column.v_box_layout.insertWidget(
        column.v_box_layout.count() - 1, sub_frame
    )
    column.sub_frames.append(sub_frame)◇
```

Fragment referenced in 185b.

Figure 207: The created sub frame is added to the layout of the column and appended to the list of sub frames of the column.

Editor → Add node dialog → Methods

⟨ *Save the node definition to list of known nodes* 189a ⟩ ≡

```
    self.node_definitions[node_view_model.id_] = node_view_model
    self.logger.debug("Added node definition %s", node_view_model)
    # TODO: Handle shortcuts

else:
    self.logger.warn(
        "Node definition %s is already known",
        node_view_model
    )
◇
```

Fragment referenced in 185b.

Figure 208: The node definition is saved to the list of known node definitions.

Editor → Add node dialog → Slots

⟨ *Connect main window components* 189b ⟩+ ≡

```
self.node_controller.do_add_node_view_definition.connect(
    self.main_window.scene_view.add_node_dialog.\
    on_node_definition_added
)◇
```

Fragment defined by 114b, 143c, 189b, 191b.
Fragment referenced in 90c.

Figure 209: The main application connects the node controller's signal that a new node definition was added with the corresponding slot of the dialog to add nodes.

Editor → Main application → Constructor

Code fragments

"../src/editor.py" 190≡

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

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

# System imports
import sys

# Project imports
from qde.editor.application import application

⟨ Main entry point  87a⟩
◇
```

"../src/qde/editor/application/application.py" 191a≡

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

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

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

# Project imports
from qde.editor.foundation import common
from qde.editor.application import node
from qde.editor.application import scene
from qde.editor.gui import main_window as qde_main_window


⟨ Main application declarations 87b ⟩
◇
```

⟨ Connect main window components 191b ⟩+ ≡

```
self.main_window.scene_view.do_add_node.connect(
    self.node_controller.on_node_added
)
self.node_controller.do_add_node_model.connect(
    self.scene_controller.on_node_model_added
)
self.scene_controller.do_select_node.connect(
    self.main_window.render_view.on_node_selected
)
self.scene_controller.do_deselect_node.connect(
    self.main_window.render_view.on_node_deselected
)
◇
```

Fragment defined by 114b, 143c, 189b, 191b.
Fragment referenced in 90c.

"../src/qde/editor/gui/main_window.py" 192a≡

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

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

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

# Project imports
from qde.editor.foundation import common
from qde.editor.gui import render as gui_render
from qde.editor.gui import scene as gui_scene


⟨ Main window declarations 89a ⟩
◇
```

⟨ *Set up render view in main window* 192b ⟩ ≡

```
self.render_view = gui_render.RenderView(self)
self.render_view.setObjectName('render_view')
self.render_view.setMinimumSize(300, 300)◇
```

Fragment referenced in 92.

⟨ *Add render view to horizontal splitter in main window* 192c ⟩ ≡

```
horizontal_splitter.addWidget(self.render_view)◇
```

Fragment referenced in 92.

"../src/qde/editor/domain/scene.py" 193a≡

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

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

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

# Project imports


⟨ Scene model declarations 93⟩
◇
```

"../src/qde/editor/gui_domain/scene.py" 193b≡

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

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

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

# Project imports
from qde.editor.foundation import common
from qde.editor.gui_domain import node as node_gui_domain

⟨ Scene graph view model declarations 94b⟩
⟨ Scene view model declarations 138a⟩
◇
```

⟨ *Scene graph view model methods* 193c⟩+ ≡

```
def __str__(self):
    """Return the string representation of the current object."""

    return str(self.id_)[0:8]
◇
```

Fragment defined by 105a, 193c.
Fragment referenced in 94b.

⟨ *Scene view model methods* 194a ⟩+ ≡

```
def __str__(self):
    """Return the string representation of the current object."""

    return str(self.id_)[0:8]
◇
```

Fragment defined by 139, 194a, 198a.
Fragment referenced in 138a.

"../src/qde/editor/application/scene.py" 194b≡

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

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

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore

# Project imports
from qde.editor.foundation import common
from qde.editor.domain     import node  as node_domain
from qde.editor.gui_domain import node  as node_gui_domain
from qde.editor.domain     import scene as scene_domain
from qde.editor.gui_domain import scene as scene_gui_domain

⟨ Scene graph controller declarations  96a ⟩
⟨ Scene controller declarations  137a ⟩
◇
```

"../src/qde/editor/gui/scene.py" 195a≡

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

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

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

# Project imports
from qde.editor.foundation import common
from qde.editor.gui_domain import node as node_gui_domain
from qde.editor.gui_domain import scene
from qde.editor.gui import node

⟨ Scene graph view declarations 105c ⟩
⟨ Scene view declarations 134a ⟩
◇
```

⟨ *Scene graph view decorators* 195b ⟩ ≡

```
common.with_logger
◇
```

Fragment referenced in 105c.

⟨ *Scene view signals* 195c ⟩ ≡

```
do_add_node = QtCore.pyqtSignal(uuid.UUID)◇
```

Fragment referenced in 134a.

⟨ *Handle node definition chosen* 195d ⟩ ≡

```
node_definition_vm = self.add_node_dialog.chosen_node_definition
self.logger.debug(
    "Node instance shall be added: %s",
    node_definition_vm
)
self.do_add_node.emit(node_definition_vm.id_)◇
```

Fragment referenced in 184a.

⟨ *Scene controller signals* 196 ⟩+ ≡

```
do_select_node = QtCore.pyqtSignal(node_gui_domain.NodeViewModel)
do_deselect_node = QtCore.pyqtSignal()  # TODO: Send deselected node as well?
◇
```

Fragment defined by 143a, 196.
Fragment referenced in 137a.

⟨ *Scene controller slots* 197 ⟩+ ≡

```
QtCore.pyqtSlot(node_domain.NodeModel, node_gui_domain.NodeViewModel)
def on_node_model_added(self, node_domain_model, node_view_model):
    self.logger.debug("Shall add node domain model: %s", node_domain_model)

    assert self.current_scene is not None

    # Add node view model to scene view model
    self.current_scene.addItem(node_view_model)
    insert_pos_x = self.current_scene.insert_at.x() * node_gui_domain.NodeViewModel.WIDTH
    insert_pos_y = self.current_scene.insert_at.y() * node_gui_domain.NodeViewModel.HEIGHT
    node_view_model.setPos(insert_pos_x, insert_pos_y)

    # Handle (de-)selection of the node view model
    node_view_model.do_select_node.connect(self.on_node_selected)
    node_view_model.do_deselect_node.connect(self.on_node_deselected)

    # Add node domain model to scene domain model
    self.current_scene.nodes.append(node_domain_model)

    self.logger.debug(
        "Node instance '%s' was added to current scene (%s) at %s",
        node_view_model,
        self.current_scene,
        node_view_model.pos()
    )
    # TODO: Check if still necessary
    # self.node_added.emit(self.current_scene)

QtCore.pyqtSlot()
def on_node_selected(self):
    """Gets triggered whenever a node was selected within the node graph
    view."""

    node_view_model = self.current_scene.selectedItems()[0]
    self.logger.debug("Node instance was selected: %s" % type(node_view_model))
    self.do_select_node.emit(node_view_model)

QtCore.pyqtSlot()
def on_node_deselected(self):
    """Gets triggered whenever a node was deselected within the node graph
    view."""

    self.logger.debug("Currently selected node instance was deselected")
    self.do_deselect_node.emit()
◇
```

Fragment defined by 141, 142ab, 197.
Fragment referenced in 137a.

⟨ *Scene view model methods* 198a ⟩+ ≡

```
def mouseReleaseEvent(self, event):
    super(SceneViewModel, self).mouseReleaseEvent(event)

    # TODO: Check boundary conditions
    # * Boundaries of scene
    # * Other nodes

    if (
            event.button() & QtCore.Qt.LeftButton
    ):
        new_x = event.scenePos().x() / node_gui_domain.NodeViewModel.WIDTH
        new_y = event.scenePos().y() / node_gui_domain.NodeViewModel.HEIGHT
        self.insert_at.setX(new_x)
        self.insert_at.setY(new_y)
        self.logger.debug(
            "Set insert at to %s, %s",
            new_x, new_y
        )
        self.invalidate()◇
```

Fragment defined by 139, 194a, 198a.
Fragment referenced in 138a.

⟨ *Scene graph controller methods* 198b ⟩+ ≡

```
⟨ Scene graph controller add root node  97a ⟩
◇
```

Fragment defined by 99, 100, 101, 102, 103, 104a, 113, 114a, 198b.
Fragment referenced in 96a.

"../logging.json" 199≡

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

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

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

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

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

"../src/qde/editor/foundation/common.py" 200a≡

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

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

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

# Project imports


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

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

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

    ⟨ Set logger name 118a ⟩
    ⟨ Logger interface 118b ⟩

⟨ Common methods 200b ⟩
◇
```

⟨ *Common methods* 200b ⟩ ≡

```
def multiply_colors(color1, color2):
    red   = (color1.redF()   * color2.redF()  ) * 255
    blue  = (color1.blueF()  * color2.blueF() ) * 255
    green = (color1.greenF() * color2.greenF()) * 255

    return QtGui.QColor(red, blue, green)
◇
```

Fragment referenced in 200a.

"../src/qde/editor/foundation/type.py" 201≡

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

# System imports
import enum

# Project imports


⟨ Node type declarations 124⟩
⟨ Node part state changed declarations 156b⟩


class Vertex(enum.Enum):
    """Possible types of vertices."""

    FULL     = 0
    SIMPLE   = 1
    PARTICLE = 2
    INSTANCE = 3


class GeometryPrimitive(enum.Enum):
    """Possible types of geometrical primitives."""

    TRIANGLE_LIST   = 0
    QUAD_LIST       = 1
    LINE_LIST       = 2
    SPRITE_LIST     = 3
    TRIANGLE_STRIPS = 4
    LINE_STRIPS     = 5
◇
```

"../src/qde/editor/domain/parameter.py" 202≡

```
# -*- coding: utf-8 -*-

"""Module for parameter-specific aspects."""

# System imports

# Project imports
from qde.editor.foundation import type as types
from qde.editor.domain import node

⟨ Parameter declarations 125, ... ⟩
⟨ Parameter domain model value generic interface 169a⟩
⟨ Parameter domain model value interface 169b⟩
⟨ Parameter domain model dynamic value 205b⟩
⟨ Parameter domain model float value 170a⟩
⟨ Parameter domain model generic value 205a⟩
⟨ Parameter domain model image value 204b⟩
⟨ Parameter domain model mesh value 206a⟩
⟨ Parameter domain model text value 204a⟩
⟨ Parameter domain model scene value 170b⟩
⟨ Parameter domain model implicit value 206b, ... ⟩
⟨ Parameter domain module methods 168⟩
◇
```

⟨ *Parameter declarations* 203 ⟩+ ≡

```
    FloatValue = create_node_definition_part.__func__(
        id_="468aea9e-0a03-4e63-b6b4-8a7a76775a1a",
        type_=types.NodeType.FLOAT
    )
    Text = create_node_definition_part.__func__(
        id_="e43bdd1b-a895-4bd8-8d5a-b401a63f7a6f",
        type_=types.NodeType.TEXT
    )
    Scene = create_node_definition_part.__func__(
        id_="bfb47e7text7-1b05-4864-8397-de30bf005ff8",
        type_=types.NodeType.SCENE
    )
    Image = create_node_definition_part.__func__(
        id_="21fd1960-1307-4b53-b7bf-d08f02757335",
        type_=types.NodeType.IMAGE
    )
    DynamicValue = create_node_definition_part.__func__(
        id_="68720ae3-8068-43ce-94d8-8705dc3b8bfe",
        type_=types.NodeType.DYNAMIC
    )
    Mesh = create_node_definition_part.__func__(
        id_="9791d341-b92c-43dd-954a-9d83b9020e43",
        type_=types.NodeType.MESH
    )
    Implicit = create_node_definition_part.__func__(
        id_="c019271c-35b6-425c-9ff2-a1d893111adb",
        type_=types.NodeType.IMPLICIT
    )

    atomic_types = [
        FloatValue,
        Text,
        Scene,
        Image,
        DynamicValue,
        Mesh,
        Implicit,
    ]

◇
```

Fragment defined by 125, 126, 203.
Fragment referenced in 202.

⟨ *Parameter domain model text value* 204a ⟩ ≡

```
class TextValue(Value):
    """A class holding values for text/string nodes."""

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

        :param string_value: the string value that shall be held
        :type  string_value: str
        """

        super(TextValue, self).__init__(string_value)
        self.function_type = types.NodeType.TEXT

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return TextValue(self.value)◇
```

Fragment referenced in 202.

⟨ *Parameter domain model image value* 204b ⟩ ≡

```
class ImageValue(ValueInterface):
    """A class holding values for image nodes."""

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

        super(ImageValue, self).__init__()
        self.function_type = types.NodeType.IMAGE

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return ImageValue()◇
```

Fragment referenced in 202.

⟨ *Parameter domain model generic value* 205a ⟩ ≡

```
class GenericValue(ValueInterface):
    """A class holding values for generic nodes."""

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

        super(GenericValue, self).__init__()
        self.function_type = types.NodeType.GENERIC

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return GenericValue()◇
```

Fragment referenced in 202.

⟨ *Parameter domain model dynamic value* 205b ⟩ ≡

```
class DynamicValue(ValueInterface):
    """A class holding values for dynamic nodes."""

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

        super(DynamicValue, self).__init__()
        self.function_type = types.NodeType.DYNAMIC

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return DynamicValue()◇
```

Fragment referenced in 202.

⟨ *Parameter domain model mesh value* 206a ⟩ ≡

```
class MeshValue(ValueInterface):
    """A class holding values for mesh nodes."""

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

        super(MeshValue, self).__init__()
        self.function_type = types.NodeType.MESH

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return MeshValue()◇
```

Fragment referenced in 202.

⟨ *Parameter domain model implicit value* 206b ⟩ ≡

```
class ImplicitValue(ValueInterface):
    """A class holding values for implicit surface nodes."""

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

        super(ImplicitValue, self).__init__()
        self.function_type = types.NodeType.IMPLICIT

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return ImplicitValue()◇
```

Fragment defined by 206b, 207a.
Fragment referenced in 202.

⟨ *Parameter domain model implicit value* 207a ⟩+ ≡

```
class ImplicitValue(ValueInterface):
    """A class holding values for implicit types."""

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

        super(ImplicitValue, self).__init__()
        self.function_type = types.NodeType.IMPLICIT

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

        :return: a clone of the currently set value
        :rtype:  qde.editor.domain.parameter.ValueInterface
        """

        return ImplicitValue()◇
```

Fragment defined by 206b, 207a.
Fragment referenced in 202.

"../src/qde/editor/domain/node.py" 207b≡

```
# -*- coding: utf-8 -*-

"""Module for node-specific aspects."""

# System imports

# Project imports
from qde.editor.foundation import type as types
from qde.editor.foundation import flag

⟨ Node domain model declarations 127a ⟩
⟨ Node part domain model declarations 155a ⟩
⟨ Node definition domain model declarations 157b ⟩
⟨ Node definition part domain model declarations 150b ⟩
⟨ Node definition input domain model declarations 171a ⟩
⟨ Node definition output domain model declarations 174a ⟩
⟨ Node definition connection domain model declarations 211a ⟩
⟨ Node definition definition domain model declarations 211b ⟩
⟨ Node definition invocation domain model declarations 212a ⟩
⟨ Node domain module methods 152, ... ⟩
◇
```

"../src/qde/editor/gui_domain/node.py" 208a≡

```
# -*- coding: utf-8 -*-

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

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

# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import flag

common.with_logger
⟨ Node view model declarations 128a⟩
◇
```

⟨ Node view model constructor 208b ⟩+ ≡

```
    self.setPos(self.position)
    self.setAcceptHoverEvents(True)
    self.setFlag(Qt.QGraphicsObject.ItemIsFocusable)
    self.setFlag(Qt.QGraphicsObject.ItemIsMovable)
    self.setFlag(Qt.QGraphicsObject.ItemIsSelectable)
    self.setFlag(Qt.QGraphicsObject.ItemClipsToShape)◇
```

Fragment defined by 128b, 131a, 208b.
Fragment referenced in 128a.

⟨ *Node view model methods* 209 ⟩+ ≡

```
def boundingRect(self):
    """Return the bounding rectangle of the node.

    :return: the bounding rectangle of the node.
    :rtype: Qt.QRectF
    """

    return Qt.QRectF(
        0, 0, self.width, self.height
    )

def create_pixmap(self):
    """Creation of the pixmap (=bitmap, the actual 'image')"""

    image = QtGui.QImage(self.boundingRect().size().toSize(),
                         QtGui.QImage.Format_ARGB32_Premultiplied)
    pixmap = QtGui.QPixmap.fromImage(image)
    pixmap.fill(QtCore.Qt.transparent)

    rect = self.boundingRect()

    painter = QtGui.QPainter()
    painter.begin(pixmap)
    painter.setRenderHint(QtGui.QPainter.Antialiasing)

    # Shape
    path = QtGui.QPainterPath()
    path.addRect(rect)
    # path.addRoundedRect(rect, 5, 5)
    painter.drawPath(path)

    # Color / gradient
    color = QtGui.QColor(255, 0, 0, 128)
    color.setHsv(color.hsvHue(), 160, 255)
    color_desaturated = color
    color_desaturated.setHsv(color.hsvHue(), 40, 255)
    top_color = QtGui.QColor(60, 70, 80)
    if self.status is not flag.NodeStatus.OK:
        top_color = QtGui.QColor(255, 0, 0)
    gradient_top_color = common.multiply_colors(
        top_color, color_desaturated
    )
    gradient_bottom_color = common.multiply_colors(
        QtGui.QColor(110, 120, 130), color_desaturated
    )
    rect_gradient = QtGui.QLinearGradient(
        QtCore.QPoint(0.0, 0.0), QtCore.QPoint(0.0, rect.height())
    )
    rect_gradient.setColorAt(0.0, gradient_top_color)
    rect_gradient.setColorAt(1.0, gradient_bottom_color)

    brush = QtGui.QBrush(rect_gradient)

    painter.fillPath(path, brush)
    painter.end()

    return pixmap
◇
```

Fragment defined by 129ab, 130b, 131b, 132a, 209, 215.
Fragment referenced in 128a.

"../nodes/16d90b34-a728-4caa-b07d-a3244ecc87e3.node" 210a≡

```
⟨ Implicit sphere node 144 ⟩◇
```

"../src/qde/editor/application/node.py" 210b≡

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

""" Module holding node related aspects concerning the application layer.
"""

# System imports
import glob
import inspect
import os
import time
import uuid
from PyQt5 import Qt
from PyQt5 import QtCore

# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import type as types
from qde.editor.technical  import json
from qde.editor.domain     import parameter
from qde.editor.domain     import node as node_domain
from qde.editor.gui_domain import node as node_gui_domain


⟨ Node controller declarations 149a ⟩
◇
```

⟨ *Node definition connection domain model declarations* 211a ⟩ ≡

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

    # Signals

    def __init__(self,
                 source_node_id, source_part_id,
                 target_node_id, target_part_id):
        """Constructor.

        :param source_node_id: the identifier of the source node.
        :type  source_node_id: uuid.uuid4
        :param source_part_id: the identifier of the part of the source node.
        :type  source_part_id: uuid.uuid4
        :param target_node_id: the identifier of the target node.
        :type  target_node_id: uuid.uuid4
        :param target_part_id: the identifier of the part of the target node.
        :type  target_part_id: uuid.uuid4
        """

        self.source_node_id = source_node_id
        self.source_part_id = source_part_id
        self.target_node_id = target_node_id
        self.target_part_id = target_part_id◇
```

Fragment referenced in 207b.

⟨ *Node definition definition domain model declarations* 211b ⟩ ≡

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

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

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

        self.id_ = id_
        self.script = script◇
```

Fragment referenced in 207b.

⟨ *Node definition invocation domain model declarations* 212a ⟩ ≡

```
class NodeDefinitionInvocation(object):
    """Represents an invocation of a definition of a node."""

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

        :param id_: the globally unique identifier of the definition.
        :type  id_: uuid.uuid4
        :param script: the script part of the invocation.
        :param script: str
        """

        self.id_ = id_
        self.script = script◇
```

Fragment referenced in 207b.

"../src/qde/editor/gui/node.py" 212b≡

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

"""Module holding node related aspects."""

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

# Project imports
from qde.editor.foundation import common
from qde.editor.gui_domain import node   as node_gui_domain
from qde.editor.gui_domain import helper as gui_helper


⟨ Add node dialog declarations  182 ⟩
◇
```

⟨ *Node controller signals* 212c ⟩+ ≡

```
do_add_node_model = QtCore.pyqtSignal(node_domain.NodeModel, node_gui_domain.NodeViewModel)◇
```

Fragment defined by 184b, 212c.
Fragment referenced in 149a.

⟨ *Node controller slots* 213 ⟩ ≡

```
QtCore.pyqtSlot(uuid.UUID)
def on_node_added(self, id):
    if id in self.node_definitions:
        node_definitions = self.node_definitions[id]

        domain_model = node_definitions[0]
        view_model   = node_definitions[1]

        # Create node domain model
        node_domain_model = node_domain.NodeModel(
            id_=domain_model.id_,
            name=domain_model.name
        )

        inputs = []
        for input in domain_model.inputs:
            self.logger.debug("Creating input %s", input.id_)
            input = node_domain.NodePart(input.id_, input.default_function)
            inputs.append(input)
        node_domain_model.inputs = inputs

        outputs = []
        for output in domain_model.outputs:
            self.logger.debug("Creating output %s of type %s", output.id_, output.type_)
            value = parameter.create_value(
                output.type_.name, ""
            )
            value_function = node_domain.create_value_function(value)
            output = node_domain.NodePart(output.id_, value_function, output.type_)
            outputs.append(output)
        node_domain_model.outputs = outputs

        definitions = []
        for definition in domain_model.definitions:
            self.logger.debug("Creating definition %s", definition.id_)
            value = parameter.create_value(
                types.NodeType.FLOAT.name, 0.0
            )
            value_function = node_domain.create_value_function(value)
            definition = node_domain.NodePart(
                definition.id_, value_function, types.NodeType.FLOAT, definition.script
            )
            definitions.append(definition)
        node_domain_model.definitions = definition

        invocations = []
        for invocation in domain_model.invocations:
            self.logger.debug("Creating invocation %s", invocation.id_)
            value = parameter.create_value(
                types.NodeType.FLOAT.name, 0.0
            )
            value_function = node_domain.create_value_function(value)
            invocation = node_domain.NodePart(
                invocation.id_, value_function, types.NodeType.FLOAT, invocation.script
            )
            invocations.append(invocation)
        node_domain_model.invocations = invocations

        parts = []
        for part in domain_model.parts:
            self.logger.debug("Creating part %s", part.id_)
            value = parameter.create_value(
                output.type_.name, ""
```

⟨ *Node view model methods paint* 214a ⟩+ ≡

```
painter.setClipRect(option.exposedRect)
painter.drawPixmap(0, 0, pixmap)

if self.isSelected():
    color = QtGui.QColor(23, 135, 84)
    painter.setPen(color)
    painter.setBrush(QtGui.QBrush(color, QtCore.Qt.SolidPattern))
    painter.drawRect(
        0,
        self.boundingRect().height() - 2,
        self.boundingRect().width(),
        self.boundingRect().height()
    )

# TODO: Use another color if bypassed or hidden
painter.setPen(QtCore.Qt.white)

# Label
painter.drawText(self.boundingRect().adjusted(1, -9, -9, -1), QtCore.Qt.AlignCenter, self.name)
painter.drawText(self.boundingRect().adjusted(1, 20, -9, -1), QtCore.Qt.AlignCenter, self.type_.name)◇
```

Fragment defined by 132bc, 133, 214a.
Fragment referenced in 130b.

⟨ *Node definition output domain model methods* 214b ⟩ ≡

```
property
def type_(self):
    """returns the type of the node definition output.

    :return: the type of the output given by the node definition part
    :rtype:  qde.editor.foundation.types.nodetype
    """

    return self.node_definition_part.type_
◇
```

Fragment referenced in 174a.

⟨ *Node view model signals* 214c ⟩ ≡

```
do_select_node = QtCore.pyqtSignal()
do_deselect_node = QtCore.pyqtSignal()◇
```

Fragment referenced in 128a.

⟨ *Node view model methods* 215 ⟩+ ≡

```
def mouseReleaseEvent(self, event):
    """Event that gets triggered when the mouse was released over the current
    node.

    :param event: the event.
    :type  event: PyQt5.QtGui.QMouseEvent
    """

    self.setSelected(self.isSelected())
    if self.isSelected():
        self.logger.debug("Node %s was selected", self.id_)
        self.do_select_node.emit()
    super(NodeViewModel, self).mouseReleaseEvent(event)

def itemChange(self, change, value):
    if (change == Qt.QGraphicsItem.ItemSelectedHasChanged):
        self.logger.debug("Node %s was deselected", self.id_)
        self.do_deselect_node.emit()

    return super(NodeViewModel, self).itemChange(change, value)
```

◇

Fragment defined by 129ab, 130b, 131b, 132a, 209, 215.
Fragment referenced in 128a.

"../src/qde/editor/foundation/flag.py" 216≡

```
# -*- coding: utf-8 -*-

"""Module for flag-specific aspects."""

# System imports
import enum

# Project imports


class NodeStatus(enum.Enum):
    """Statues which a node can have."""

    OK              = 0
    NO_INPUTS       = 1
    WRONG_INPUT     = 2
    INPUT_ERRONEOUS = 3
    INPUT_CYCLIC    = 4
    LINK_MISSING    = 5
    TOO_MANY_INPUTS = 6


class Geometry(enum.Enum):
    """Flags denoting the type of geometric buffer."""

    STATIC          = 1
    DYNAMIC         = 2
    INDEX_BUFFER_16 = 4
    INDEX_BUFFER_32 = 8


class BlendMode(enum.Enum):
    """Flags denoting the available blend modes."""

    ZERO        = 1
    ONE         = 2
    SRCCOLOR    = 3
    INVSRCCOLOR = 4
    SRCALPHA    = 5
    INVSRCALPHA = 6
    DSTALPHA    = 7
    INVDSTALPHA = 8
    DSTCOLOR    = 9
    INVDSTCOLOR = 10


class BlendOperations(enum.Enum):
    """Flags denoting the available blend operations."""

    ADD    = 1
    SUB    = 2
    INVSUB = 3
    MIN    = 4
    MAX    = 5


class Culling(enum.Enum):
    """Flags denoting the available modes for culling."""

    NONE  = 1
    FRONT = 2
    BACK  = 3

```

"../src/qde/editor/foundation/constant.py" 217≡

```
# -*- coding: utf-8 -*-

"""Module for constants."""

# System imports
import enum

# Project imports


class Graphics(object):
    """Constants related to graphical things."""

    MAXIMUM_TEXTURES       = 5
    MAXIMUM_RENDER_TARGETS = 5


class Buffer(enum.Enum):
    """Constants for buffers."""

    VERTEX_BUFFER_DYNAMIC = 0
    INDEX_BUFFER_DYNAMIC  = 1
    FRAME_BUFFER_DYNAMIC  = 2


class ShaderData(enum.Enum):
    """Constants for shader data."""

    CAMERA = 1
◇
```

"../src/qde/editor/technical/json.py" 218a≡

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

""" Module holding JSON related aspects.
"""

# System imports
import json
import uuid

# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import type as types
from qde.editor.domain import node
from qde.editor.domain import parameter


⟨ JSON module declarations 218b ⟩
◇
```

⟨ *JSON module declarations* 218b ⟩ ≡

```
common.with_logger
class Json(object):
    """Class handling JSON relevant tasks.
    """

    ⟨ JSON methods 163, ... ⟩
◇
```

Fragment referenced in 218a.

⟨ *JSON methods* 219 ⟩+ ≡

```
classmethod
def build_node_definition_connection(cls, json_input):
    """Builds and returns a connection for a node definition from the given
    JSON input data.

    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: the connection of a node definition.
    :rtype:  qde.editor.domain.node.NodeDefinitionConnection
    """

    source_node_id = uuid.UUID(json_input['source_node'])
    source_part_id = uuid.UUID(json_input['source_part'])
    target_node_id = uuid.UUID(json_input['target_node'])
    target_part_id = uuid.UUID(json_input['target_part'])

    node_definition_connection = node.NodeDefinitionConnection(
        source_node_id,
        source_part_id,
        target_node_id,
        target_part_id
    )

    cls.logger.debug("Built node definition connection")
    return node_definition_connection
◇
```

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

⟨ *JSON methods* 220 ⟩+ ≡

```
classmethod
def build_node_definition_definition(cls, json_input):
    """Builds and returns a definition for a node definition from the given
    JSON input data.

    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: the definition of a node definition.
    :rtype:  qde.editor.domain.node.NodeDefinitionDefinition
    """

    definition_id = uuid.UUID(json_input['id_'])
    script        = str(json_input['script'])

    node_definition_definition = node.NodeDefinitionDefinition(
        definition_id,
        script
    )

    cls.logger.debug("Built node definition definition")
    return node_definition_definition
◇
```

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

⟨ *JSON methods* 221 ⟩+ ≡

```
classmethod
def build_node_definition_invocation(cls, json_input):
    """Builds and returns a invocation for a node definition from the given
    JSON input data.

    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: the invocation of a node definition.
    :rtype:  qde.editor.domain.node.NodeDefinitionInvocation
    """

    invocation_id = uuid.UUID(json_input['id_'])
    script        = str(json_input['script'])

    node_definition_invocation = node.NodeDefinitionInvocation(
        invocation_id,
        script
    )

    cls.logger.debug("Built node definition invocation")
    return node_definition_invocation
◇
```

Fragment defined by 163, 166, 173, 175, 180, 219, 220, 221.
Fragment referenced in 218b.

"../src/qde/editor/gui_domain/helper.py" 222≡

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

"""Module holding graphical user interface related helper classes and
methods."""

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

# Project importss
from qde.editor.foundation import common
from qde.editor.gui_domain import node


common.with_logger
class ClickableLabel(QtWidgets.QLabel):
    """Class providing a label object which emits a signal called 'clicked'
    when receiving a mouse press event."""

    # Signals
    do_add_node = QtCore.pyqtSignal()

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

        :param text: the text, that the label will show.
        :type text: str
        :param parent: the parent object of this label.
        :type parent: Qt.QObject
        """

        super(ClickableLabel, self).__init__(text, parent)
        parent.installEventFilter(self)
        label_font = QtGui.QFont()
        label_font.setFamily(label_font.defaultFamily())
        self.setFont(label_font)
        self.logger.debug(self.font())

    def eventFilter(self, object, event):
        if event.type() == QtCore.QEvent.Enter:
            font = self.font()
            font.setUnderline(True)
            self.setFont(font)
            return True
        elif event.type() == QtCore.QEvent.Leave:
            font = self.font()
            font.setUnderline(False)
            self.setFont(font)
            return True

        return False

    ⟨ Mouse press event of clickable label 223a⟩
◇
```

⟨ *Mouse press event of clickable label* 223a ⟩ ≡

```
    def mousePressEvent(self, event):
        """Event handler when a mouse button was pressed on this label. Emits a
        signal called 'do_add_node'.

        :param event: the event which occurred.
        :type event: Qt.QMouseEvent
        """

        self.do_add_node.emit()
        super(ClickableLabel, self).mousePressEvent(event)
◇
```

Fragment referenced in 222.

"../src/qde/editor/technical/graphics.py" 223b≡

```
# -*- coding: utf-8 -*-

"""Module providing an abstraction layer for graphics."""

# System imports
import jinja2
import numpy as np
import os
import sys
from PyQt5 import QtCore
from PyQt5 import QtGui

# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import constant
from qde.editor.foundation import flag
from qde.editor.technical import geometry


⟨ Graphics class 224 ⟩
⟨ Render state class 225a ⟩
⟨ Shader data class 225b ⟩
⟨ Shader object class 226a ⟩
◇
```

⟨ *Graphics class* 224 ⟩ ≡

```
common.with_logger
class Graphics(object):
    """A class providing an abstraction layer for the graphical processing
    unit."""

    __singleton_instance = None

    ⟨ Graphics constructor 226b ⟩
    ⟨ Graphics activate render state 226c ⟩
    ⟨ Graphics activate render target 227a ⟩
    ⟨ Graphics activate all render targets 227b ⟩
    ⟨ Graphics add geometry 228 ⟩
    ⟨ Graphics add shader 229 ⟩
    ⟨ Graphics clear 230a ⟩
    ⟨ Graphics clear render state 230b ⟩
    ⟨ Graphics create buffer 231a ⟩
    ⟨ Graphics create FBO 231b ⟩
    ⟨ Graphics create IBO 232a ⟩
    ⟨ Graphics create VAO 232b ⟩
    ⟨ Graphics create VBO 232c ⟩
    ⟨ Graphics deactivate all render targets 233c ⟩
    ⟨ Graphics deactivate current render state 233a ⟩
    ⟨ Graphics deactivate render target 233b ⟩
    ⟨ Graphics load geometry 234 ⟩
    ⟨ Graphics get attribute location 236a ⟩
    ⟨ Graphics get uniform location 236b ⟩
    ⟨ Graphics load shader from template 237 ⟩
    ⟨ Graphics render geometry 238 ⟩
    ⟨ Graphics render implicit 240b ⟩
    ⟨ Graphics set clear color 240c ⟩
    ⟨ Graphics set up IBO 241 ⟩
    ⟨ Graphics set up VAO 242 ⟩
    ⟨ Graphics set up VBO 243 ⟩
    ⟨ Graphics set value for uniform 244 ⟩
    ⟨ Graphics convert vertices to array 245a ⟩

    # Slots

    # Classmethods

    ⟨ Graphics create instance 245b ⟩
    ⟨ Graphics get instance 246a ⟩
◇
```

Fragment referenced in 223b.

⟨ *Render state class* 225a ⟩ ≡

```
common.with_logger
class RenderState(object):
    """A class representing the state of rendering."""

    def __init__(self):
        self.blend_source      = flag.BlendMode.ONE
        self.blend_destination = flag.BlendMode.ZERO
        self.blend_operation   = flag.BlendOperations.ADD
        self.cull_mode         = flag.Culling.NONE
        self.depth_func        = flag.Depth.LESS
        self.depth_test        = True
        self.depth_write       = True
        self.render_target     = None
        self.shader_object     = None
        self.textures          = []
        self.texture_flags     = {}
        self.viewport          = QtCore.QRect()
        self.write_colors      = True

        tm = flag.TextureMaterial
        flags = tm.CLAMP or tm.BILINEAR
        for i in range(constant.Graphics.MAXIMUM_TEXTURES):
            self.texture_flags[i] = flags
◇
```

Fragment referenced in 223b.

⟨ *Shader data class* 225b ⟩ ≡

```
common.with_logger
class ShaderData(object):
    """A class holding GLSL shader related data."""


    class Data(object):
        """A class for holding arbitrary structures."""

        pass

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

        self._type = type
        self.data  = self.Data()
◇
```

Fragment referenced in 223b.

⟨ *Shader object class* 226a ⟩ ≡

```
common.with_logger
class ShaderObject(object):
    """A class representing an OpenGL Shading Language (GLSL) shader object
    containing a compute, a fragment and a vertex shader, as well as the
    shader's data an program."""

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

        self.compute_shader  = None
        self.fragment_shader = None
        self.program         = None
        self.shader_data     = None
        self.vertex_shader   = None
◇
```

Fragment referenced in 223b.

⟨ *Graphics constructor* 226b ⟩ ≡

```
def __init__(self, opengl_context, size, path="data"):
    """Constructor."""

    self.ctx    = opengl_context
    self.size   = size
    self.path   = path

    self.gl     = self.ctx.version_functions
    self.width  = size.width()
    self.height = size.height()

    self.render_state   = RenderState()

    self.geometries     = []
    self.shader_objects = []
◇
```

Fragment referenced in 224.

⟨ *Graphics activate render state* 226c ⟩ ≡

```
def activate_render_state(self):
    """Clears the buffer bits and activates the render targets of the
    current rendering state."""

    self.clear()
    self.activate_render_targets()
◇
```

Fragment referenced in 224.

⟨ *Graphics activate render target* 227a ⟩ ≡

```
def activate_render_target(self, rt):
    """Activates (binds) the given render target.

    :param rt: the render target to activate/bind.
    :type rt:  TODO
    """

    if rt is not None:
        if rt.isValid() and not rt.isBound():
            rt.bind()
            self.clear()
        else:
            self.logger.warn(
                "Render target (id=%s) is fishy, cannot bind. Valid: %s",
                rt.handle(), rt.isValid()
            )
◇
```

Fragment referenced in 224.

⟨ *Graphics activate all render targets* 227b ⟩ ≡

```
def activate_render_targets(self):
    """Activates (binds) all available render targets."""

    rt = self.render_state.render_target
    self.activate_render_target(rt)
◇
```

Fragment referenced in 224.

⟨ *Graphics add geometry* 228 ⟩ ≡

```
def add_geometry(self, flags, vertex_type, primitive_type,
                 fill_callback=None, fill_callback_params=None):
    """TODO: Describe method."""

    new_geometry = geometry.Geometry()
    # TODO: Fix this!
    # new_geometry.is_dynamic = (flags & qflags.Geometry.dynamic.value)
    new_geometry.is_dynamic = True
    new_geometry.vertex_type = vertex_type
    new_geometry.primitive_type = primitive_type
    new_geometry.fill_callback = fill_callback
    new_geometry.fill_callback_params = fill_callback_params
    # TODO: Fix this!
    # geometry.is_indexed = ((flags & qflags.Geometry.index_buffer_16.value) or
    #                        (flags & qflags.Geometry.index_buffer_32.value))
    new_geometry.is_indexed = True

    # TODO: Do this dynamically!
    new_geometry.vertex_size = sys.getsizeof(geometry.SimpleVertex)
    # TODO: geometry.index_size  = ?
    new_geometry.is_loading = False
    new_geometry.used_indices = 0
    new_geometry.used_vertices = 0
    new_geometry.max_indices = 0
    new_geometry.max_vertices = 0
    new_geometry.ibo = None
    new_geometry.vbo = None

    self.geometries.append(new_geometry)

    return new_geometry
◇
```

Fragment referenced in 224.

⟨ *Graphics add shader* 229 ⟩ ≡

```
def add_shader(self, name):
    """Creates and returns a OpenGL Shader Language (GLSL) object based on
    given files. Searches for a vertex and fragment shader templates
    matching the pattern 'given_name.vs.tpl'' and 'given_name.fs.tpl'
    respectively.

    :param name: the name of the shader and the shader template files.
    :type  name: str

    :return: a linked shader as a shader object
    :rtype:  qde.editor.technical.graphics.ShaderObject
    """

    shader_object = ShaderObject()
    path   = "{0}{1}{2}".format(
        self.path, os.path.sep, "shaders"
    )
    vertex_shader   = QtGui.QOpenGLShader(QtGui.QOpenGLShader.Vertex)
    fragment_shader = QtGui.QOpenGLShader(QtGui.QOpenGLShader.Fragment)
    program         = QtGui.QOpenGLShaderProgram(self.ctx)

    vs_tpl_name = "{0}.vs.tpl".format(name)
    vertex_shader.setObjectName(vs_tpl_name)
    fs_tpl_name = "{0}.fs.tpl".format(name)
    fragment_shader.setObjectName(fs_tpl_name)

    vs_tpl = self.load_shader_template(path, vs_tpl_name)
    fs_tpl = self.load_shader_template(path, fs_tpl_name)

    assert vertex_shader.compileSourceCode(vs_tpl.render()), \
        "Could not compile vertex shader for '%s' at '%s'" % (name, path)
    assert fragment_shader.compileSourceCode(fs_tpl.render()), \
        "Could not compile fragment shader for '%s' at '%s'" % (name, path)

    program.addShader(vertex_shader)
    program.addShader(fragment_shader)
    program.link()
    program.setObjectName(name)

    shader_object.vertex_shader = vertex_shader
    shader_object.fragment_shader = fragment_shader
    shader_object.program = program

    self.shader_objects.append(shader_object)

    return shader_object
```

◇

Fragment referenced in 224.

⟨ *Graphics clear* 230a ⟩ ≡

```
def clear(self):
    """Clears the color and depth buffer bits."""

    # TODO: Move this to render_state
    self.logger.debug("Clearing")
    bitmask = self.gl.GL_COLOR_BUFFER_BIT or self.gl.GL_DEPTH_BUFFER_BIT
    self.gl.glClear(bitmask)
◇
```

Fragment referenced in 224.

⟨ *Graphics clear render state* 230b ⟩ ≡

```
def clear_render_state(self):
    """Clears the current render state. """

    self.render_state.depth_test = True
    self.render_state.depth_write = True
    self.render_state.depth_func = qflags.Depth.less

    self.render_state.cull_mode = qflags.Culling.back

    self.render_state.blend_source      = qflags.BlendMode.one
    self.render_state.blend_destination = qflags.BlendMode.zero
    self.render_state.blend_operation   = qflags.BlendOperations.add
    self.render_state.write_colors      = True

    tm = qflags.TextureMaterial
    flags = tm.clamp or tm.bilinear
    for i in range(qcns.Graphics.MAXIMUM_TEXTURES):
        self.render_state.texture_flags[i] = flags

    self.render_state.viewport.setWidth(self.width)
    self.render_state.viewport.setHeight(self.height)
◇
```

Fragment referenced in 224.

⟨ *Graphics create buffer* 231a ⟩ ≡

```
def create_buffer(self, buffer_type):
    """Creates a buffer object of given type.

    :param buffer_type: type of the buffer to create.
    :type  buffer_type: TODO

    :return: the created buffer object.
    :rtype:  TODO
    """

    buffer = QtGui.QOpenGLBuffer(buffer_type)
    buffer.create()

    return buffer
◇
```

Fragment referenced in 224.

⟨ *Graphics create FBO* 231b ⟩ ≡

```
def create_fbo(self, size):
    """Creates a framebuffer object with the provided size.

    :param size: size of the render target.
    :type  size: PyQt5.QtCore.QSize

    :return: the created framebuffer object.
    :rtype:  PyQt5.QtGui.QOpenGLFramebufferObject
    """

    assert(self.ctx is not None)
    assert(self.ctx.isValid())
    surface = self.ctx.surface()
    if not self.ctx.makeCurrent(surface):
        message = "Could not make surface current, no FBO created!"
        self.logger.fatal(message)
        raise Exception(message)

    format = QtGui.QOpenGLFramebufferObjectFormat()
    format.setAttachment(QtGui.QOpenGLFramebufferObject.CombinedDepthStencil)
    # TODO: Use parameter for samples
    # TODO: Use samples
    format.setSamples(0)
    width = size.width()
    height = size.height()
    buffer = QtGui.QOpenGLFramebufferObject(width, height, format)

    return buffer
◇
```

Fragment referenced in 224.

⟨ *Graphics create IBO* 232a ⟩ ≡

```
def create_ibo(self):
    """Creates a index buffer object (IBO).

    :return: the created index buffer object.
    :rtype:  TODO
    """

    ibo = self.create_buffer(QtGui.QOpenGLBuffer.IndexBuffer)

    return ibo
◇
```

Fragment referenced in 224.

⟨ *Graphics create VAO* 232b ⟩ ≡

```
def create_vao(self):
    """Creates a vertex array object (VAO).

    :return: the created vertex array object.
    :rtype:  TODO
    """

    vao = QtGui.QOpenGLVertexArrayObject()
    vao.create()

    return vao
◇
```

Fragment referenced in 224.

⟨ *Graphics create VBO* 232c ⟩ ≡

```
def create_vbo(self):
    """Creates a vertex buffer object (VBO).

    :return: the created vertex buffer object.
    :rtype:  TODO
    """

    vbo = self.create_buffer(QtGui.QOpenGLBuffer.VertexBuffer)

    return vbo
◇
```

Fragment referenced in 224.

⟨ *Graphics deactivate current render state* 233a ⟩ ≡

```
def deactivate_render_state(self):
    """Deactivates the current render state."""

    self.deactivate_render_targets()
◇
```

Fragment referenced in 224.

⟨ *Graphics deactivate render target* 233b ⟩ ≡

```
def deactivate_render_target(self, rt):
    """Deactivates the given render target.

    :param rt: the render target to deactivate.
    :type  rt: TODO
    """

    if rt is not None:
        if rt.isValid() and rt.isBound():
            rt.release()
        else:
            self.logger.warn(
                "Render target (%s) is fishy, cannot release. Valid: %s",
                rt.handle(), rt.isValid()
            )
◇
```

Fragment referenced in 224.

⟨ *Graphics deactivate all render targets* 233c ⟩ ≡

```
def deactivate_render_targets(self):
    """Deactivates all render targets."""

    rt = self.render_state.render_target
    self.deactivate_render_target(rt)
◇
```

Fragment referenced in 224.

⟨ *Graphics load geometry* 234 ⟩ ≡

```
def load_geometry(self, geometry, vertices, indices=None, colors=None):
    """Loads the given geometry by adding vertices (optionally using
    indices) and colors.

    :param geometry: TODO
    :type geometry: TODO
    :param vertices: TODO
    :type vertices: TODO
    :param indices: TODO
    :type indices: TODO
    :param colors: TODO
    :type colors: TODO
    """

    assert(vertices is not None)
    assert(geometry.is_loading == False)

    program = self.render_state.shader_object.program

    vertex_location  = program.attributeLocation("a_position")
    if geometry.has_texture:
        texture_location = self.get_attribute_location(program,
                                                       "a_texCoords")

    geometry.is_loading    = True
    geometry.used_colors  = 0
    geometry.used_indices  = 0
    geometry.used_vertices = 0

    if indices is not None:
        geometry.is_indexed = True
    else:
        geometry.is_indexed = False
    self.logger.debug("Using indices: %s", geometry.is_indexed)
    self.logger.debug("Using texture: %s", geometry.has_texture)

    if geometry.is_dynamic:
        ⟨ Graphics load dynamic geometry 235 ⟩

    else:
        # TODO: Handle static geometries
        self.logger.warn("Static rendering not yet implemented")

    geometry.is_loading = False
◇
```

Fragment referenced in 224.

⟨ *Graphics load dynamic geometry* 235 ⟩ ≡

```
vertex_array = self.vertices_to_array(geometry, *vertices)
self.logger.debug("Vertices: %s", vertex_array)
vao = self.setup_vao(geometry)
vbo = self.setup_vbo(geometry, vertex_array)

program.enableAttributeArray(vertex_location)
program.setAttributeBuffer(
    vertex_location,
    self.gl.GL_FLOAT,  # Type
    0,                 # Offset
    3,                 # Tuple size
    0                  # Stride
)
if geometry.is_indexed:
    self.logger.debug("Setting up IBO")
    ibo = self.setup_ibo(geometry, indices)
if colors not in [None, []]:
    self.logger.debug("Setting up CBO")
    colors_array = self.colors_to_array(colors)
    cbo = self.setup_cbo(geometry, colors_array)
    colors_location = self.get_attribute_location(program, "a_colors")
    program.enableAttributeArray(colors_location)
    program.setAttributeBuffer(
        colors_location,
        self.gl.GL_FLOAT,  # Type
        0,                 # Offset
        4,                 # Tuple size
        0,                 # Stride
    )
if geometry.has_texture:
    self.logger.debug("Setting up texture buffer")
    texture_coord_location = self.get_attribute_location(program, "a_texCoords")
    program.enableAttributeArray(texture_coord_location)
    program.setAttributeBuffer(
        texture_location,
        self.gl.GL_FLOAT,                 # Type
        3 * vertex_array.dtype.itemsize,  # Offset
        2,                                # Tuple size
        5 * vertex_array.dtype.itemsize,  # Stride
    )
if colors not in [None, []]:
    cbo.release()
if geometry.is_indexed:
    ibo.release()
vbo.release()
vao.release()
◇
```

Fragment referenced in 234.

⟨ *Graphics get attribute location* 236a ⟩ ≡

```
def get_attribute_location(self, program, location):
    """Returns the location of the given attribute within the given shader
    program.


    :param program: TODO
    :type  program: TODO
    :param location: TODO
    :param location: TODO
    """

    program_location = program.attributeLocation(location)
    assert program_location > -1, \
        "Attribute '%s' not found in program '%s'" % (
            location,
            program.objectName()
        )

    return program_location
◇
```

Fragment referenced in 224.

⟨ *Graphics get uniform location* 236b ⟩ ≡

```
def get_uniform_location(self, program, location):
    """Returns the location of the given uniform within the given shader
    program.


    :param program: TODO
    :type  program: TODO
    :param location: TODO
    :param location: TODO
    """

    program_location = program.uniformLocation(location)
    if program_location < 0:
        self.logger.warn("Uniform '%s' not found in program '%s'" %
                         (location, program.objectName()))

    return program_location
◇
```

Fragment referenced in 224.

⟨ *Graphics load shader from template 237* ⟩ ≡

```
def load_shader_template(self, path, tpl_name):
    """Loads a shader template by the given name from the given path in the
    Jinja2 format.

    :param path: the path of the template to load.
    :type  path: str
    :param tpl_name: the name of the template to load.
    :type  tpl_name: str

    :return: the loaded template
    :rtype:  jinja2.Template
    """

    env = jinja2.Environment(
        loader=jinja2.FileSystemLoader(path)
    )
    return env.get_template(tpl_name)
```

◇

Fragment referenced in 224.

⟨ *Graphics render geometry* 238 ⟩ ≡

```
def render_geometry(self, geometry):
    """Renders the given geometry.

    :param geometry: the geometry to render.
    :type geometry:  qde.editor.technical.geometry.Geometry
    """

    assert(geometry is not None)
    assert(not geometry.is_loading)

    if geometry.is_dynamic and geometry.fill_callback is not None:
        geometry.fill_callback(geometry, geometry.fill_callback_params)

    if not geometry.used_vertices:
        self.logger.warn("No used vertices for current geometry")
        return

    # TODO: Do this properly
    # self.gl.glMatrixMode(self.gl.GL_PROJECTION)

    self.activate_render_state()
    program = self.render_state.shader_object.program
    program.bind()
    self.logger.debug("Rendering using shader %s", program.objectName())
    vao = geometry.vertex_array_object
    vao.bind()
    vertex_location = self.get_attribute_location(program, "a_position")
    assert(vertex_location > -1)
    program.enableAttributeArray(vertex_location)

    # TODO: Do this dynamically
    ⟨ Graphics set global uniforms 239a ⟩

    # TODO: Read from render_state / geometry
    if geometry.is_indexed:
        ⟨ Graphics render indexed geometry 239b ⟩

    else:
        ⟨ Graphics render non-indexed geometry 240a ⟩

    vao.release()
    program.disableAttributeArray(vertex_location)
    program.release()
    self.deactivate_render_state()
◇
```

Fragment referenced in 224.

⟨ *Graphics set global uniforms* 239a ⟩ ≡

```
resolution_location = self.get_uniform_location(program, "u_globalResolution")
resolution = QtGui.QVector2D(self.size.width(), self.size.height())
self.set_uniform_value(program, resolution_location, resolution)

if geometry.colour_buffer is None:
    color_location = self.get_uniform_location(program, "u_color")
    color = QtGui.QColor(100, 80, 150, 255)
    self.set_uniform_value(program, color_location, color)

# TODO: Set this using a camera class
mvp_matrix_location = self.get_uniform_location(program, "u_mvpMatrix")
# TODO: mvp_matrix = self.render_state.shader_object.shader_data.data.mvp_matrix
mvp_matrix = QtGui.QMatrix4x4()
mvp_matrix.setToIdentity()
self.set_uniform_value(program, mvp_matrix_location, mvp_matrix)
◇
```

Fragment referenced in 238.

⟨ *Graphics render indexed geometry* 239b ⟩ ≡

```
vao = geometry.vertex_array_object
vao.bind()

if geometry.has_texture:
    texture_coord_location = self.get_attribute_location(program, "a_texCoords")
    program.enableAttributeArray(texture_coord_location)
    self.gl.glActiveTexture(self.gl.GL_TEXTURE0)
    self.gl.glBindTexture(self.gl.GL_TEXTURE_2D, geometry.texture)
    texture_location = self.get_uniform_location(program, "u_texture")
    # TODO: Fix this!
    self.set_uniform_value(program, texture_location, 0)  # self.gl.GL_TEXTURE0)

if geometry.colour_buffer is not None:
    cbo = geometry.colour_buffer
    cbo.bind()

self.logger.debug("Rendering indexed using %d indices", geometry.used_indices)
# self.gl.glDrawArrays(self.gl.GL_TRIANGLES, 0, geometry.used_indices)
# self.gl.glDrawArrays(self.gl.GL_TRIANGLE_STRIP, 0, geometry.used_indices)
self.gl.glDrawElements(
    self.gl.GL_TRIANGLES,
    geometry.used_indices,
    self.gl.GL_UNSIGNED_INT,
    geometry.indices
)
if geometry.has_texture:
    program.disableAttributeArray(texture_coord_location)
◇
```

Fragment referenced in 238.

⟨ *Graphics render non-indexed geometry* 240a ⟩ ≡

```
self.logger.debug("Rendering non-indexed using %d vertices", geometry.used_vertices)
self.gl.glDrawArrays(self.gl.GL_TRIANGLES, 0, geometry.used_vertices)
◇
```

Fragment referenced in 238.

⟨ *Graphics render implicit* 240b ⟩ ≡

```
def render_implicit(self):
    """Renders implicit scenes."""

    self.activate_render_state()
    program = self.render_state.shader_object.program
    program.bind()
◇
```

Fragment referenced in 224.

⟨ *Graphics set clear color* 240c ⟩ ≡

```
def set_clear_color(self, c):
    """Sets the clear color of OpenGL to the given color.

    :param c: the color to set the clear color to.
    :type  c: PyQt5.QtGui.QColor
    """

    self.clear_color = c
    self.gl.glClearColor(c.redF(), c.greenF(), c.blueF(), c.alphaF())
    self.clear()
◇
```

Fragment referenced in 224.

⟨ *Graphics set up IBO* 241 ⟩ ≡

```
def setup_ibo(self, geometry, indices):
    """Sets up a index buffer object (IBO) and assigns it to the given
    geometry.

    :param geometry: the geometry which the generated IBO will be assigned
                     to.
    :type geometry:  qde.editor.technical.geometry.Geometry
    :param indices:  a numpy float32 array holding the ordered indices.
    :type indices:   numpy.array
    :return:         the generated IBO.
    :rtype:          TODO
    """

    indices_array = np.asarray(indices, dtype=np.int16)
    assert len(indices) == np.size(indices_array)

    required_ib_size      = np.size(indices_array) * indices_array.dtype.itemsize
    geometry.index_size   = required_ib_size
    geometry.max_indices  = np.size(indices_array)
    geometry.used_indices = np.size(indices_array)
    assert geometry.used_indices == len(indices)

    if geometry.index_buffer is None:
        ibo = self.create_ibo()
        geometry.index_buffer = ibo
    else:
        ibo = geometry.index_buffer
    assert(ibo.isCreated())
    assert(ibo.bind())

    ibo.allocate(indices_array.tostring(), required_ib_size)
    assert required_ib_size == ibo.size(), \
        "Required: %s, allocated: %s"  % (required_ib_size, ibo.size())

    geometry.indices = indices

    return ibo
◇
```

Fragment referenced in 224.

⟨ *Graphics set up VAO* 242 ⟩ ≡

```
def setup_vao(self, geometry):
    """Sets up a vertex array object (VAO) and assigns it to the given
    geometry.

    :param geometry: the geometry which the generated VAO will be assigned
                     to.
    :type geometry:  qde.editor.technical.geometry.Geometry
    :param vertices: a numpy float32 array holding the ordered vertices.
    :type vertices:  numpy.array

    :return:         the generated VAO.
    :rtype:          TODO
    """

    if geometry.vertex_array_object is None:
        vao = self.create_vao()
        geometry.vertex_array_object = vao
    else:
        vao = geometry.vertex_array_object
    assert(vao.isCreated())
    vao.bind()

    return vao
◇
```

Fragment referenced in 224.

⟨ *Graphics set up VBO* 243 ⟩ ≡

```
def setup_vbo(self, geometry, vertices):
    """Sets up a vertex buffer object (VBO) and assigns it to the given
    geometry.

    :param geometry: the geometry which the generated VBO will be assigned
                     to.
    :type geometry:  qgeom.Geometry
    :param vertices: an numpy float32 array holding the ordered vertices.
    :type vertices:  numpy.array
    :return:         the generated VBO.
    :rtype:          QOpenGLBuffer of type QOpenGLBuffer.VertexBuffer
    """

    if geometry.has_texture:
        required_vb_size   = np.size(vertices) * 5 * vertices.dtype.itemsize
    else:
        required_vb_size   = np.size(vertices) * 3 * vertices.dtype.itemsize
    geometry.vertex_size   = required_vb_size
    geometry.max_vertices  = np.size(vertices) / 3
    geometry.used_vertices = np.size(vertices) / 3

    if geometry.vertex_buffer is None:
        vbo = self.create_vbo()
        geometry.vertex_buffer = vbo
    else:
        vbo = geometry.vertex_buffer
    assert(vbo.isCreated())
    assert(vbo.bind())

    vbo.allocate(required_vb_size)
    assert required_vb_size == vbo.size(), \
        "Required: %s, allocated: %s"  % (required_vb_size, vbo.size())
    vbo.setUsagePattern(QtGui.QOpenGLBuffer.StaticDraw)
    vbo.write(0, vertices.tostring(), len(vertices.tostring()))
    geometry.vertices = vertices

    return vbo
◇
```

Fragment referenced in 224.

⟨ *Graphics set value for uniform* 244 ⟩ ≡

```
def set_uniform_value(self, program, location, value):
    """Sets the given value for the given location within the provided
    shader program.

    :param program: TODO
    :type program: TODO
    :param location: TODO
    :type location: TODO
    :param value: the value to set.
    :type value: TODO
    """

    assert(program is not None)
    assert(program.isLinked())
    assert(location is not None)
    assert(value is not None)

    if location >= 0:
        program.setUniformValue(location, value)
    else:
        self.logger.warn(
            "Could not set %s at %s within %s",
            value,
            location,
            program
        )
```

◇

Fragment referenced in 224.

⟨ *Graphics convert vertices to array* 245a ⟩ ≡

```
def vertices_to_array(self, geometry, *vertices):
    """Returns the given vertices as array.

    :param geometry: TODO
    :type geometry: TODO
    :param vertices: TODO
    :type vertices: TODO

    :return: an array containing the given vertices.
    :rtype:  np.array
    """

    temp = []

    for v in vertices:
        pos = v.position
        temp.append(pos.x())
        temp.append(pos.y())
        temp.append(pos.z())

        if geometry.has_texture and v.uv is not None:
            temp.append(v.uv.x())
            temp.append(v.uv.y())

    return np.array(temp, dtype=np.float32)
◇
```

Fragment referenced in 224.

⟨ *Graphics create instance* 245b ⟩ ≡

```
classmethod
def create(cls, opengl_context, path="data"):
    """Creates an instance of the graphics interface."""

    if cls.__singleton_instance is not None:
        raise Exception("Graphics was already created")

    cls.__singleton_instance = cls(opengl_context, path)
    return cls.__singleton_instance
◇
```

Fragment referenced in 224.

⟨ *Graphics get instance* 246a ⟩ ≡

```
classmethod
def instance(cls):
    """Returns the initialized singleton instance.

    :return: the singleton instance of the initialized graphics abstraction
             object.
    :rtype:  qde.editor.technical.graphics.Graphics
    """
    assert cls.__singleton_instance is not None, \
        "Graphics were not initialized yet"
    return cls.__singleton_instance
◇
```

Fragment referenced in 224.

"../src/qde/editor/technical/geometry.py" 246b≡

```
# -*- coding: utf-8 -*-

"""Module holding geometrical definitions."""

# System imports
from PyQt5 import QtGui

# Project imports
from qde.editor.foundation import type as types


⟨ Geometrical object class 246c ⟩
⟨ Vertex class 247a ⟩
⟨ Geometry class 247b ⟩
⟨ Simple vertex class 248 ⟩
◇
```

⟨ *Geometrical object class* 246c ⟩ ≡

```
class GeometricalObject(object):
    """Class providing a basis for any geometrical object, containing a
    teporary value and a tag, acting as counters."""

    def __init__(self):
        self.temp = 0
        self.tag  = 0
◇
```

Fragment referenced in 246b.

⟨ *Vertex class* 247a ⟩ ≡

```
class Vertex(GeometricalObject):
    """Class providing an abstraction to vertices."""

    def __init__(self):
        self.position = QtGui.QVector3D()
        self.normal   = None
        self.uv       = None
        self.color    = QtGui.QColor()
◇
```

Fragment referenced in 246b.

⟨ *Geometry class* 247b ⟩ ≡

```
class Geometry(GeometricalObject):
    """Class providing an abstraction to geometrical objects."""

    def __init__(self):
        """Constructor."""
        self._type                   = types.GeometryPrimitive.TRIANGLE_LIST
        self.colour_buffer           = None
        self.face_culling_parameters = None
        self.fill_callback           = None
        self.fill_callback_params    = None
        self.index_buffer            = None
        self.index_size              = 0
        self.indices                 = []
        self.is_loading              = False
        self.is_indexed              = False
        self.is_dynamic              = False
        self.max_vertices            = 0
        self.max_indices             = 0
        self.texture                 = None
        self.used_colors             = 0
        self.used_indices            = 0
        self.used_vertices           = 0
        self.vertex_array_object     = None
        self.vertex_buffer           = None
        self.vertex_size             = 0
        self.vertex_type             = types.Vertex.FULL
        self.vertices                = []

    property
    def has_texture(self):
        return self.texture is not None
◇
```

Fragment referenced in 246b.

⟨ *Simple vertex class* 248 ⟩ ≡

```
class SimpleVertex(GeometricalObject):
    """Class representing a simplified vertex, containing only its position and
    texture coordinates (UV)."""

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

        self.position = QtGui.QVector3D()
        self.uv       = QtGui.QVector2D()
◇
```

Fragment referenced in 246b.

"../src/qde/editor/gui/render.py" 249≡

```
# -*- coding: utf-8 -*-

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

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


# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import type as node_types
from qde.editor.technical  import graphics
from qde.editor.technical  import render
from qde.editor.gui_domain import node as node_gui_domain


common.with_logger
class RenderView(QtWidgets.QOpenGLWidget):
    """The render view widget. A widget for rendering nodes."""

    OPENGL_MAJOR        = 4
    OPENGL_MINOR        = 1


    ⟨ Render view constructor 250⟩
    ⟨ Render view initialize OpenGL 251⟩
    ⟨ Render view check for valid render target 252a⟩

    def paintGL(self):
        """Renders the currently set content."""

        self.render_current_node()

    ⟨ Render view render current node 252b⟩

    def render_none(self):
        """Renders nothing except a solid background color."""

        color = QtGui.QColor(QtCore.Qt.cyan)
        self.gfx.set_clear_color(color)

    def render_implicit(self, node):
        """Renders the given node of type implicit."""

        self.logger.debug("Rendering implicit")
        self.render_scene(scene=None, camera=None, time=0.0)

    ⟨ Render view render scene 253a⟩

    def resizeGL(self, w, h):
        """Gets triggered whenever the widget's view port is resized."""

        size = QtCore.QSize(w, h)
        self.gfx.size = size
        self.has_view_changed = True
        # self.update()

    def setup_render_target(self):
        """Sets up a render target."""
```

⟨ *Render view constructor* 250 ⟩ ≡

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

    super(RenderView, self).__init__(parent)

    self.current_node  = None
    self.render_target = None
    self.renderer      = None

    self.has_view_changed = False

    self.surface_format = QtGui.QSurfaceFormat()
    self.surface_format.setProfile(QtGui.QSurfaceFormat.CoreProfile)
    self.surface_format.setVersion(self.OPENGL_MAJOR, self.OPENGL_MINOR)

    self.version_profile = QtGui.QOpenGLVersionProfile(self.surface_format)
    self.version_profile.setProfile(QtGui.QSurfaceFormat.CoreProfile)
    self.version_profile.setVersion(self.OPENGL_MAJOR, self.OPENGL_MINOR)

    self.setFocusPolicy(QtCore.Qt.StrongFocus)
    self.setFormat(self.surface_format)
◇
```

Fragment referenced in 249.

⟨ *Render view initialize OpenGL* 251 ⟩ ≡

```
def initializeGL(self):
    """Initializes OpenGL."""

    try:
        version_functions = self.context().versionFunctions(
            self.version_profile
        )
        if version_functions is None:
            message = (
                'Could not initialize OpenGL with profile {0}'
            ).format(
                self.surface_format.version()
            )
            self.logger.fatal(message)
            raise Exception(message)

    except Exception as e:
            message = (
                'Could not initialize OpenGL with profile {0} ({1})'
            ).format(
                self.surface_format.version(),
                e
            )
            self.logger.fatal(message)
            raise Exception(message)

    self.ctx = self.context()
    self.ctx.version_functions = version_functions
    self.ctx.size = self.frameSize()
    self.gfx = graphics.Graphics.create(self.ctx, self.size())
    self.renderer = render.RayMarchingRenderer(self.ctx)

    # Seems to happen under OS X (10.11) using PyQt5.7 on an Intel
    # Iris 6100 only.
    # The main render target seem to need the ID 1, as otherwise
    # the render target is empty. Well, there _is_ data, but somehow
    # it is not usable, although it has the same properties as when
    # the render target is set up first.
    self.setup_render_target()

    color = QtGui.QColor(QtCore.Qt.black)
    self.gfx.set_clear_color(color)

    self.logger.info('Initialized OpenGL with profile {0} at ({1})'.format(
        self.surface_format.version(),
        self.size()
    ))
    self.logger.info("Size: %s", self.frameSize())

◇
```

Fragment referenced in 249.

⟨ *Render view check for valid render target* 252a ⟩ ≡

```
def has_valid_render_target(self):
    """Returns wether the render target is valid or not.

    :return: the validity of the render target.
    :rtype:  bool
    """

    if self.render_target is None:
        return False

    if self.render_target.size() != self.size():
        return False

    return True
◇
```

Fragment referenced in 249.

⟨ *Render view render current node* 252b ⟩ ≡

```
def render_current_node(self):
    """Renders the currently selected node."""

    # TODO: Process node (time-wise)
    # TODO: Set view changed to True if node has been processed/

    if self.has_view_changed:
        if self.current_node is None:
            self.render_none()
            self.logger.warn("No node set for rendering")
        elif self.current_node.type_ == node_types.NodeType.IMPLICIT:
            self.render_implicit(self.current_node)
        else:
            self.render_none()
            self.logger.warn("Rendered unknown node")

        self.has_view_changed = False
    else:
        self.logger.debug("No change, not rendering")
◇
```

Fragment referenced in 249.

⟨ *Render view render scene* 253a ⟩ ≡

```
def render_scene(self, scene, camera, time):
    """Renders the given scene using the given camera at the given time.

    :param scene: TODO
    :type scene: TODO
    :param camera: TODO
    :type camera: TODO
    :param time: TODO
    :type time: TODO
    """

    assert(time >= 0.0)

    self.renderer.render_scene(scene, camera, None, time)
    # self.renderer.render_scene(scene, camera, self.render_target, time)
    # self.copy_render_target_to_screen()
◇
```

Fragment referenced in 249.

⟨ *Render view handle node selection* 253b ⟩ ≡

```
QtCore.pyqtSlot(node_gui_domain.NodeViewModel)
def on_node_selected(self, node_view_model):
    """Slot that gets triggered whenever a node gets selected within the
    node graph view.

    :param node_view_model: view model of the selected node
    :type  node_view_model: qde.editor.gui_domain.node.NodeViewModel
    """

    self.logger.debug("Set selected node for rendering: %s" % node_view_model)

    # TODO: Check if node is valid
    self.current_node = node_view_model

    # TODO: Remove this and process time wise
    # This must get done in render_current_node
    self.has_view_changed = True
    self.update()
◇
```

Fragment referenced in 249.

⟨ *Render view handle node deselection* 254a ⟩ ≡

```
QtCore.pyqtSlot()
def on_node_deselected(self):
    """Slot that gets triggered whenever a node gets deselected within the
    node graph view.
    """

    self.current_node = None

    # TODO: Remove this and process time wise
    # This must get done in render_current_node
    self.has_view_changed = True
    self.update()
◇
```

Fragment referenced in 249.

"../src/qde/editor/technical/render.py" 254b≡

```
# -*- coding: utf-8 -*-

"""Module providing an abstraction layer for rendering."""

# System imports
from PyQt5 import QtCore
from PyQt5 import QtGui

# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import flag
from qde.editor.foundation import type as types
from qde.editor.technical  import geometry
from qde.editor.technical  import graphics


⟨ Renderer class  255 ⟩
⟨ Ray marching renderer class  256 ⟩
◇
```

⟨ *Renderer class* 255 ⟩ ≡

```
common.with_logger
class Renderer(object):
    """Renderer base class."""

    def __init__(self, default_shader_name="default"):
        """Constructor.

        :param default_shader_name: name of the shader that gets used by default.
        :type  default_shader_name: str
        """

        gfx = graphics.Graphics.instance()
        flags = (flag.Geometry.DYNAMIC.value &
                 flag.Geometry.INDEX_BUFFER_32.value)
        self.default_geometry = gfx.add_geometry(flags, types.Vertex.SIMPLE,
                                                 types.GeometryPrimitive.QUAD_LIST)
        self.quad_geometry = gfx.add_geometry(flags, types.Vertex.SIMPLE,
                                              types.GeometryPrimitive.QUAD_LIST)

        # Set up shader(s)
        self.default_shader = gfx.add_shader(default_shader_name)
        color = QtGui.QColor(QtCore.Qt.black)
        gfx.set_clear_color(color)

    def render_scene(self, scene, camera, render_target, time):
        raise NotImplemented(
            "%s must be implemented in child class." % __name__
        )
◇
```

Fragment referenced in 254b.

⟨ *Ray marching renderer class 256* ⟩ ≡

```
class RayMarchingRenderer(Renderer):
    """Provides a ray marching renderer using sphere tracing."""

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

        :param opengl_context: a valid OpenGL context.
        :type  opengl_context: PyQt5.QtGui.QOpenGLContext
        """

        super(RayMarchingRenderer, self).__init__(
            "sphere_tracer"
        )
        self.ctx = opengl_context

    def render_scene(self, scene, camera, render_target, time):
        """Renders the given scene using the given camera for the given render
        target.

        :param scene: TODO
        :type  scene: TODO
        :param camera: TODO
        :type  camera: TODO
        :param render_target: TODO
        :type  render_target: TODO
        :param time: TODO
        :type  time: TODO
        """

        gfx = graphics.Graphics.instance()
        gfx.render_state.shader_object = self.default_shader

        v1 = geometry.Vertex()
        v1.position = QtGui.QVector3D(-1.0, -1.0,  0.0)
        v2 = geometry.Vertex()
        v2.position = QtGui.QVector3D( 1.0, -1.0,  0.0)
        v3 = geometry.Vertex()
        v3.position = QtGui.QVector3D(-1.0,  1.0,  0.0)
        v4 = geometry.Vertex()
        v4.position = QtGui.QVector3D( 1.0,  1.0,  0.0)
        # vertices = cluster.vertices
        # indices  = cluster.indices
        vertices = [v1, v2, v3, v4]
        indices = [0, 1, 2, 2, 1, 3]
        gfx.load_geometry(self.default_geometry, vertices, indices)
        gfx.render_geometry(self.default_geometry)
◇
```

Fragment referenced in 254b.

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] ——, *The TEXbook*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987, ISBN: 0-201-13448-9.

[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] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80", *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988, ISSN: 0896-8438.

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

[21] 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).

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

[23] 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.

[24] "The JSON Data Interchange Format", ECMA, Tech. Rep. Standard ECMA-404 1st Edition / October 2013, Oct. 2013.

[25] Epic Games, Inc., *Distance Field Ambient Occlusion*, 2017.

[26] ——, *Ray Traced Distance Field Soft Shadows*, 2017.