

QDE

A visual animation system

MTE7102: Projektarbeit 2

Studiengang: Informatik
Autor: Sven Osterwalder¹
Betreuer: Prof. Claude Fuhrer²
Datum: 24.07.2016
Version: 0.1



Licensed under the Creative Commons Attribution-ShareAlike 3.0 License

¹sven.osterwalder@students.bfh.ch

²claudio.fuhrer@bfh.ch

Versionen

Version	Datum	Autor(en)	Änderungen
0.1	24.07.2016	SO	Initiale Erstellung des Dokumentes

Todo list

Add sequencer to glossary	1
Add rendering to glossary	1
Add GPU to glossary	1
check if Konzept is the right word	1
write ray tracing everywhere the same way	3
check if Konzept is the right word	3
add ref. to sw. arch. chapter	3
Explain, that prototype was done before	3
Provide new learning contents	4
Use realistic milestones. Explain, that prototype was built before (sadly)	5

Abstract

Das Fachgebiet “Computergrafik” war schon immer bestrebt eine möglichst realitätsnahe Darstellung von Szenen und Modellen zu erzeugen. Eine Darstellung, welche möglichst nahe an der Realität respektive der menschlichen Wahrnehmung liegt.

Im Laufe der Zeit entstanden verschiedene Ansätze um eine solche Darstellung zu erreichen. Ein Teilgebiet davon sind Beleuchtungsmodelle, welche die Beleuchtung einer Darstellung bzw. einer Szene berechnen.

Ein relativ realistisch wirkendes Beleuchtungsmodell ist Ray Tracing (zu Deutsch Strahlen-Verfolgung), welches auf den physikalischen Grundlagen von Licht und Materialien von Oberflächen basiert.

Diese Verfahren waren lange Zeit zu langsam um damit eine Darstellung zu erreichen. Durch die Weiterentwicklung der Computer, vor allem der Grafikkarten (GPUs), ist Ray Tracing jedoch für die Darstellung von Szenen in Echtzeit wieder interessant geworden.

Diese Projektarbeit stellt die (theoretischen) Grundlagen zur Erzeugung und Darstellung von Szenen und Modellen sowie ein spezielles Ray Tracing Verfahren zur Darstellung von Bildern in Echtzeit vor. Dabei handelt es sich um *Volume Ray Casting* bzw. *Sphere Tracing*.

Um die Hypothese zu stützen, dass die komplexen, realistischer wirkenden Verfahren wieder in den Fokus der Darstellung von Szenen in Echtzeit gerückt sind, wurde ein Prototyp entwickelt, der Sphere Tracing in Echtzeit auf der Grafikkarte (GPU) umsetzt.

Mit dem vorgestellten Verfahren gelingt mittels moderner Grafikkarten eine Darstellung von Szenen und Modellen in Echtzeit.

Inhaltsverzeichnis

Abstract	iv
1. Einleitung	1
2. Administratives	2
2.1. Beteiligte Personen	2
2.2. Aufbau des Dokumentes	2
2.3. Ergebnisse (Deliverables)	2
3. Aufgabenstellung	3
3.1. Motivation	3
3.2. Ziele und Abgrenzung	3
4. Vorgehen	5
4.1. Arbeitsorganisation	5
4.2. Projektphasen	5
4.3. Technologien	7
5. Software-Architektur	9
5.1. Anforderungen	9
5.2. Komponenten	26
5.3. Domänenmodell	28
5.4. Sequenz-Diagramme	31
5.5. Logische Architektur	33
5.6. Klassendiagramme	36
5.7. Prototyp	38
6. Schlusswort	40
6.1. Erweiterungsmöglichkeiten	40
Glossar	42
Literaturverzeichnis	42
Abbildungsverzeichnis	42
Tabellenverzeichnis	43
Auflistungsverzeichnis	44
Anhang	46
A. Meeting minutes	47

1. Einleitung

In [1] wurde mit “Sphere Tracing” ein hoch effizientes Ray-Tracing-Verfahren vorgestellt, welches Objekte (und somit auch Szenen) sowie Operationen mittels impliziten Funktionen darstellt. Dies mag auf den ersten Blick praktisch erscheinen, da die Definition der meisten Objekte und Operationen nur wenige Zeilen lang ist. Dies hat jedoch zur Folge, dass einerseits Programme selbst bei kleinsten Änderungen neu kompiliert werden müssen und, dass andererseits bei komplexen Szenen schnell die Übersicht verloren geht. Möchte man Szenen bzw. Objekte animieren, so erhöht dies die Komplexität erneut.

Diese Projektarbeit stellt daher eine Software-Architektur für ein System zur einfachen Erstellung und Handhabung visueller Szenen in Echtzeit vor.

Das System erlaubt die Erstellung und Handhabung von Szenen mittels einer grafischen Benutzeroberfläche (GUI). Ein Graph erlaubt eine einfache und intuitive Komposition von visuellen Szenen. Ein Sequencer erlaubt die Animation von Elementen, wie z.B. Modellen oder Bitmaps.

Add sequencer

Als Renderingverfahren kommt Sphere Tracing zum Einsatz, ein hoch-optimiertes Ray-Tracing-Verfahren, welches die Darstellung von Szenen in Echtzeit per GPU erlaubt.

Add rendering

Die Machbarkeit des Konzeptes wird durch einen Prototypen aufgezeigt, welcher die Komposition sowie Darstellung von einfachen Szenen in Echtzeit erlaubt.

Add GPU to

check if Kon

2. Administratives

Einige administrative Aspekte der Projektarbeit werden angesprochen, obwohl sie für das Verständnis der Resultate nicht notwendig sind.

Im gesamten Dokument wird nur die männliche Form verwendet, womit aber beide Geschlechter gemeint sind.

2.1. Beteiligte Personen

Autor Sven Osterwalder¹
Betreuer Prof. Claude Fuhrer²

Begleitet den Studenten bei der Projektarbeit

2.2. Aufbau des Dokumentes

Die vorliegende Arbeit ist aufgebaut wie folgt:

- Einleitung zur Projektarbeit
- Beschreibung der Aufgabenstellung
- Vorgehen des Autors im Hinblick auf die gestellten Aufgaben
- Lösung der gestellten Aufgaben
- Verwendete Technologien

2.3. Ergebnisse (Deliverables)

Nachfolgend sind die abzugebenden Objekte aufgeführt:

- **Abschlussdokument**
Das Abschlussdokument beinhaltet die Ausarbeitung einer Software-Architektur eines Systems zur einfachen Erstellung und Handhabung visueller Szenen in Echtzeit.
- **Prototyp**
Der Prototyp entstand im Rahmen der Bearbeitung der Thematik. Er zeigt die Machbarkeit (von Teilen) der angedachten Software-Architektur auf.

¹sven.osterwalder@students.bfh.ch

²claudio.fuhrer@bfh.ch

3. Aufgabenstellung

3.1. Motivation

Das in der vorhergehenden Projektarbeit [1] vorgestellte Renderingverfahren, Sphere Tracing, optimiert das Raytracing-Verfahren so weit, dass eine Darstellung von Szenen mit hoher Qualität in Echtzeit möglich ist.

write ray tra
way

Das Verfahren nutzt implizite Funktionen um Objekte (und somit auch Szenen) zu definieren. Selbiges gilt auch für Operationen. Dies mag auf den ersten Blick praktisch erscheinen, da die Definitionen der meisten Objekte und Operationen nur wenige Zeilen lang sind. Dies hat jedoch zur Folge, dass einerseits Programme selbst bei kleinsten Änderungen neu kompiliert werden müssen und, dass andererseits bei komplexen Szenen schnell die Übersicht verloren geht. Möchte man Szenen bzw. Objekte animieren, so erhöht dies die Komplexität erneut.

Diese Projektarbeit stellt daher eine Software-Architektur für ein System zur einfachen Erstellung und Handhabung visueller Szenen in Echtzeit vor.

Die Machbarkeit des Konzeptes wird durch einen Prototypen aufgezeigt, welcher die Komposition sowie Darstellung von einfachen Szenen in Echtzeit erlaubt.

check if Kon

3.1.1. Demoszene

In den 1980er-Jahren entwickelte sich “unter Anhängern der Computerszene ... während der Blütezeit der 8-Bit-Systeme” [[wikipedia_foundation_demoszene_2015](#)] eine Bewegung namens Demoszene. “Ihre Mitglieder, die häufig Demoszener oder einfach Szener genannt werden, erzeugen mit Computerprogrammen auf Rechnern so genannte Demos – Digitale Kunst, meist in Form von musikalisch unterlegten Echtzeit-Animationen.” [[wikipedia_foundation_demoszene_2015](#)]

Es handelt sich bei der Demoszene um ein sehr aktives und kreatives Umfeld, in welchem die Technologie ständig an die Grenzen ihrer Möglichkeiten gebracht wird. In diesem Umfeld entstehen regelmässig neue Ideen zur Erzeugung von noch realistischer wirkenden Bildern. Dabei findet eine wechselseitige Beeinflussung zwischen dem akademischen Umfeld und der Demoszene statt.

So wurde auch das in [1] vorgestellte *Sphere Tracing* Verfahren relativ früh von in der Demoszene aktiven Personen aufgegriffen und behandelt.

3.2. Ziele und Abgrenzung

Diese Projektarbeit besteht aus zwei Teilen. Der Beschreibung der Software-Architektur sowie der Umsetzung eines Prototypen. Dieser bildet einen kleinen Teil der Software-Architektur ab.

add ref. to s

Bei dem entwickelten Prototypen handelt es sich um eine Machbarkeitsstudie. Diese zeigt, dass die Komposition sowie Darstellung von einfachen Szenen in Echtzeit mit dem angedachten System möglich ist.

Explain, tha
re

Diese Projektarbeit dient als Vorarbeit und Grundlage für das MSE-Modul *MTE7103* — “Master Thesis” (Folgemodul und Abschlussarbeit).

3.2.1. Vorgängige Aktivitäten

Die vorhergehende Projektarbeit [1] bildet die Grundlage für diese Projektarbeit.

3.2.2. Neue Lerninhalte

Zusätzlich zu den formalen Lerninhalten hatte die Arbeit für den Autor die folgenden neuen Lerninhalte:

Provide new

- TODO

4. Vorgehen

4.1. Arbeitsorganisation

4.1.1. Treffen

Besprechungen mit dem Betreuer der Arbeit halfen, die gesteckten Ziele zu erreichen und Fehlentwicklungen zu vermeiden. Der Betreuer unterstützte den Autor dabei mit Vorschlägen. Insgesamt fanden drei Treffen statt. Sie wurden in Form eines Protokolles festgehalten. Das Protokoll findet sich unter Anhang A.

4.2. Projektphasen

4.2.1. Meilensteine

. Um bei der Arbeit ein möglichst strukturiertes Vorgehen einzuhalten, wurden folgende Projektphasen gewählt:

- Start der Projektarbeit
- Erarbeitung und Festhalten der Anforderungen
- Erarbeitung der theoretischen Grundlagen
- Erstellung der abschliessenden Dokumentation
- Erstellung eines Prototypen

Die Phasen *Erarbeitung der theoretischen Grundlagen*, *Erstellung der abschliessenden Dokumentation* sowie *Erstellung eines Prototypen* liefen parallel ab. Erkenntnisse einer Phase flossen jeweils in die anderen Phasen ein.

Use realistic
prototype wa

4.2.2. Zeitplan / Projektphasen

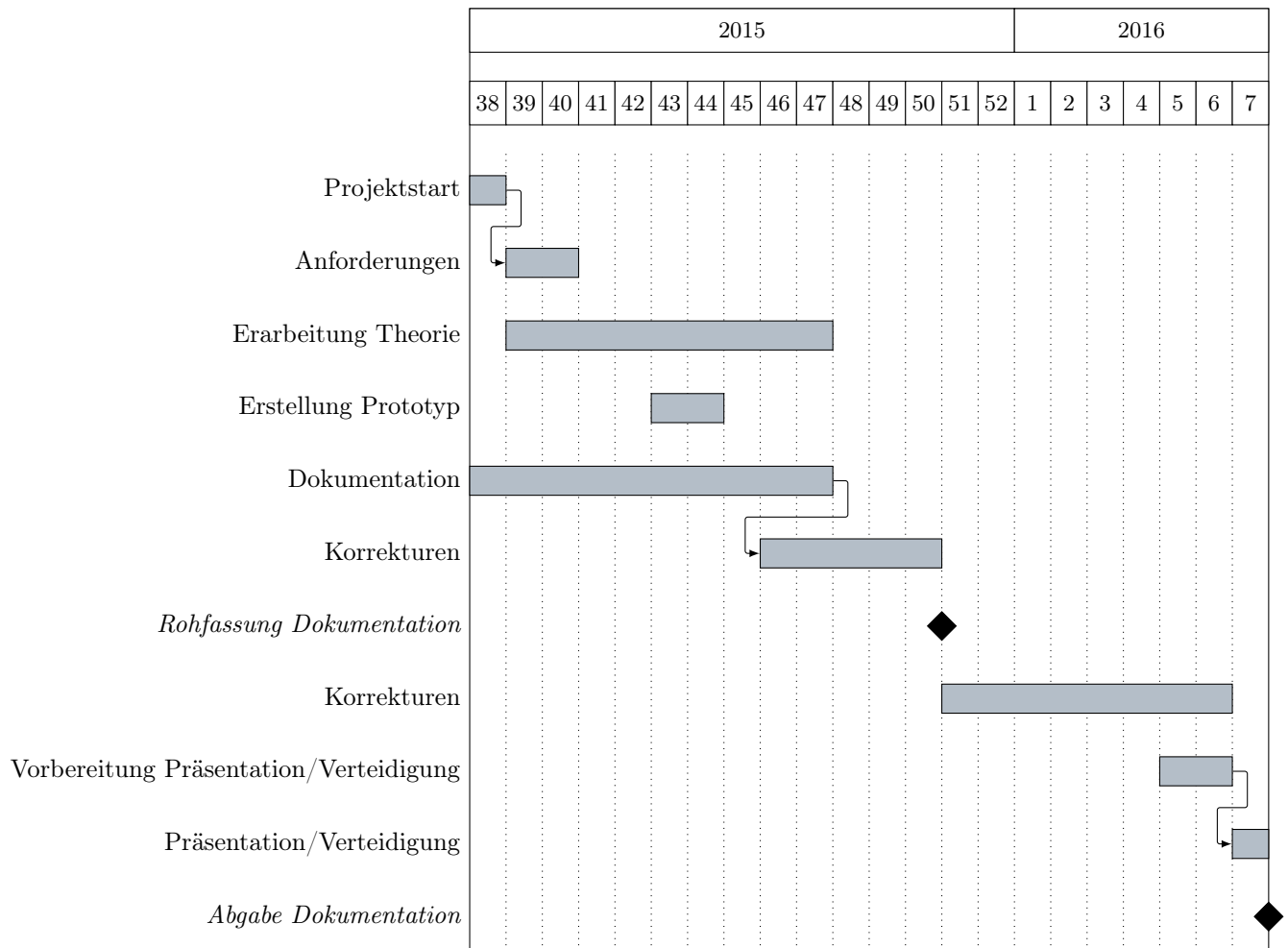


Abbildung 4.1.: Zeitplan; Der Titel stellt Jahreszahlen, der Untertitel Kalenderwochen dar

Projektstart

In der ersten Phase wurden die Zwischenziele der Arbeit identifiziert und skizziert. Um Einzelheiten der Aufgabe zu verstehen, wurde das notwendige Vorwissen über Algorithmen für globale Beleuchtung erarbeitet. Weiter wurde die Grundlage dieser Dokumentation erstellt.

Anforderungen

In dieser Phase wurde das Ziel dieser Projektarbeit festgelegt. Ausgehend vom Ziel wurden die dazu erforderlichen Projektphasen festgelegt.

Erarbeitung theoretische Grundlagen

Diese Phase bedeutete Publikationen zu lesen und durch zuarbeiten, was immer zu vielen neuen Denkanstößen weiteren Recherchen führte. Die Erkenntnisse dieser Phase flossen stetig in die Dokumentation ein, ergänzten und beeinflussten diese.

Dokumentation

Die vorliegende Arbeit entspricht der Dokumentation. Sie wurde während der gesamten Projektarbeit stetig erweitert und diente zur Reflexion von fertiggestellten Teilen.

4.3. Technologien

4.3.1. Tools und Software

Dokumentation und Prototyp

L^AT_EX Eine Makro-Sammlung für das T_EX-System. Wurde zur Erstellung dieser Dokumentation eingesetzt. Diese Dokumentation wurde mittels L^AT_EX geschrieben.

Make Build-Automations-Werkzeug, wurde zur Erstellung dieses Dokumentes eingesetzt.

CMake Build-Automations-Werkzeug, wurde zur Erstellung des Prototypen eingesetzt.

zotero Ein freies, quelloffenes Literaturverwaltungsprogramm zum Sammeln, Verwalten und Zitieren unterschiedlicher Online- und Offline-Quellen [2].

VIM Vi IMproved. Ein freier, quelloffener Texteditor zur Textbearbeitung. Er wurde zum Verfassen der Dokumentation sowie zur Entwicklung des Prototypen eingesetzt.

LLVM Low Level Virtual Machine. Eine Compiler-Architektur zum Kompilieren von Applikationen. Wurde zur Kompilation des Prototypen eingesetzt.

clang Ein C-Sprachen-Frontend für LLVM. Wurde zur Kompilation des Prototypen eingesetzt.

Papyrus Ein quelloffenes Werkzeug zur Erstellung von UML- und SysML-Modellen [the_eclipse_foundation_papyrus].

Pencil Ein quelloffenes Werkzeug zur Erstellung von Prototypen von grafischen Benutzeroberflächen [evolus_pencil_2010].

Arbeitsorganisation

Git Freie Software zur verteilten Versionsverwaltung, wurde für die Entwicklung dieser Dokumentation sowie des Prototypen verwendet. Die Projektarbeit findet sich unter GitHub¹.

GitHub Eine freie Hosting-Plattform für Git mit Weboberfläche.

4.3.2. Standards und Richtlinien

Programmcode

Der Programmcode des Prototypen, welcher in C++ geschrieben wurde, folgt den offiziellen Richtlinien für C++ von Google ².

Pseudocode

Da der Autor, bedingt durch seine früheren, beruflichen Tätigkeiten, in der Programmiersprache *Python* relativ bewandert ist, wird diese als Sprache zur Beschreibung von Pseudocode verwendet. Dabei wird kein Augenmerk auf die formale Korrektheit, geschweige denn auf die Lauffähigkeit des Pseudocodes gelegt.

¹ <https://www.github.com/sosterwalder/mte7101-project1>

² <https://google.github.io/styleguide/cppguide.html>

Projekt-Struktur

Um die Übersicht zu wahren und den Verwaltungsaufwand minimal zu halten, wurde eine entsprechende Projekt-Struktur gewählt. Diese ist in Auflistung 4.1 ersichtlich.

Projekt-Struktur

```
19 bin/      -- Compiled, binary file of the prototype
20 build/    -- Temporary directory for build output
21 doc/      -- Top folder containing all documentation
22   abstract/ -- A short abstract of the project
23   doc/      -- The actual documentation
24     img/    -- Images used for the documentation
25     inc/    -- LaTeX files used for inclusion to maintain
26               readability and managabilty
27   static/   -- Static files used for inclusion, e.g. the
28               bibliography, versioning of the document
29               and so on
30   attachment/ -- Attachments for the documentation,
31               e.g. the minutes of the held meetings
32   uml/      -- Source files as well as output files for all
33               UML related documentation data. Mostly Eclipse
34               Papyrus compatible files
35 inc/      -- External source files for inclusion, needed by the
36               prototype(s)
37 lib/      -- External libraries for linking against when building
38               the prototype(s)
39 resources/ -- Various resources needed for building the binary,
```

Auflistung 4.1: Projekt-Struktur.

5. Software-Architektur

Als Grundlage zur Entwicklung der Software-Architektur dient *Applying UML and Patterns* von Larman.

Da [3] auf dem Unified Process (UP) [4] basiert, wird vor allem dieser angewendet. Dies hat ein iteratives Arbeiten zur Folge, da sich der UP auf agile Ansätze wie Extreme Programming (XP) und Scrum fokussiert [3, S. 18].

Der Leser findet in dieser Projektarbeit keine fertige Architektur, welche sich direkt umsetzen lässt. Vielmehr dient diese als Grundlage für die darauffolgende Arbeit, MTE7013 — “Master Thesis”.

Die Entwicklung der Architektur ist ein iterativer Prozess, welcher unter Fortschreiten dieser sowie der darauffolgenden Projektarbeit immer wieder fortgesetzt wird. Die Architektur wird somit kontinuierlich verändert und verbessert. Verbesserung ist im Sinne der Anpassung an die stetig neu gewonnenen Erkenntnisse gemeint.

5.1. Anforderungen

5.1.1. Vision

Der Autor dieser Projektarbeit stellt sich eine Software zur Verwaltung und Darstellung von Echtzeit-Animationen vor. Die Software soll es Anwendern erlauben Echtzeit-Animationen in intuitiver Weise zu erstellen. Sie soll zudem modular gehalten sein, so dass spätere Änderungen, wie z.B. zusätzliche Arten von Rendering, ohne Weiteres adaptierbar sind. Des Weiteren soll eine erstellte Echtzeit-Animation mit geringem Aufwand exportiert werden können. Ein solcher Export soll dann — losgelöst vom Editor — von einem Betrachter über eine ausführbare Datei wiedergegeben werden können. Die Software besteht also eigentlich aus zwei Applikationen: Dem *Editor*, zum Erstellen und Verwalten von Echtzeit-Animationen, sowie dem *Player*, zum Betrachten von Echtzeit-Animationen.

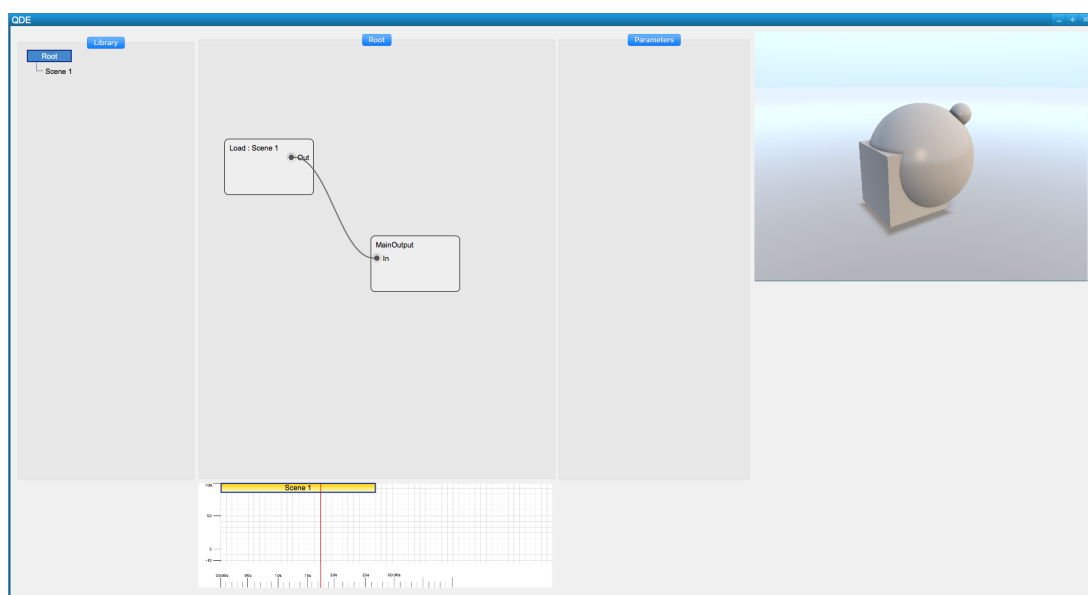


Abbildung 5.1.: Mögliches Aussehen des Editors. ¹

5.1.2. Akteure (actors)

Ein Akteur ist ein Objekt mit einem Verhalten (informal gesprochen), wie zum Beispiel eine Person (definiert durch eine Rolle), ein Computersystem oder eine Organisation [3, S.63].

Nachfolgend finden sich alle Akteure der hier spezifizierten Software-Architektur. Dabei wird zwischen primären und sekundären Akteuren unterschieden.

Primäre Akteure

- **Anwender**
Ein *Anwender* erstellt Echtzeit-Animationen mit der Software. Er hat also eine aktive Rolle.
- **Betrachter**
Ein *Betrachter* nutzt die Software um eine durch den *Anwender* erstellte Echtzeit-Animation anzusehen. Er nimmt also eine passive Rolle ein.

Sekundäre Akteure

- **Entwickler**
Ein *Entwickler* erweitert die Software um neue Elemente (wie z.B. neue Shader).

5.1.3. Use Cases

Bei Use Cases handelt es sich um Anforderungen, genauer gesagt um funktionelle bzw. Anforderungen an das Verhalten eines Systems [3, S. 61 bis 63]. Sie sagen also aus, was ein System tut beziehungsweise tun soll. Die nachfolgenden Use Cases sind keinesfalls vollständig — dann wäre der Sinn des UP bzw. eines iterativen Vorgehens verfehlt. Sie entwickeln sich viel mehr über die Zeit mit dem Fortschreiten der Umsetzung (welche vorallem in der darauffolgenden Projektarbeit statt findet).

Die Use Cases 1 bis und mit 5 sind bewusst sehr grob gehalten. Sie veranschaulichen den Gesamtprozess. Use Cases 6 bis 10 zeigen den Prozess zur Erstellung einer neuen Szene sowie einer Animation mehr im Detail.

Einleitend folgt eine Tabelle zur Erklärung der einzelnen Begriffe in den nachfolgenden Use Cases.

Tabelle 5.1.: Erklärung der Begrifflichkeiten der Use Cases, angelehnt an [3, S. 67].

Bereich	Der Bereich des Use Cases. Dies ist entweder der Editor oder aber der Player (siehe hierzu auch 5.1.1).
Stufe (level)	“Ziel des Benutzers” oder “Unterfunktion”.
Primärer Akteur	Primärer Akteur des Use Cases: Anwender, Betrachter oder Entwickler. Siehe auch 5.1.2.
Stakeholder und Interessen	Interessenten des Use Cases und deren Ziele.
Erfolgsszenario	Typisches, gewünschtes Szenario des Uses Cases, bei welchem keinerlei Fehler oder Nebenwirkungen auftreten.
Erweiterungen	Zusätzliche Szenarien, erfolgreich oder fehlerhaft.
Zusätzliche Anforderungen	Non-funktionelle Anforderungen, welche in Relation zu dem Use Case stehen.

Use Case UC1: Betrachten einer Echtzeit-Animation

¹Eigene Darstellung mittels Pencil.

Tabelle 5.2.: Use Case UC1: Betrachten einer Echtzeit-Animation.

Bereich	Player
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Betrachter
Stakeholder und Interessen	Betrachter: Möchte eine zuvor erstellte Echtzeit-Animation ansehen. Anwender: Testen einer erstellten Echtzeit-Animation. Entwickler: Testen einer erstellten Echtzeit-Animation.
Erfolgsszenario	<ol style="list-style-type: none"> 1. Der Betrachter startet die Player-Applikation und wählt die gewünschten Optionen, wie Auflösung und Bit-Tiefe. 2. Der Betrachter startet die Animation. 3. Der Player spielt die Animation bis zum Ende. 4. Der Player wird automatisch geschlossen.
Erweiterungen	<ol style="list-style-type: none"> (a) Vorzeitiger Abbruch durch den Betrachter <ol style="list-style-type: none"> (i) Der Betrachter bricht eine laufende Animation ab. (ii) Der Player wird sofort geschlossen. (b) Absturz/Ausfall des Systems <ol style="list-style-type: none"> (i) Das System stürzt an einem beliebigen Punkt ab. (ii) Neustart des Players. (iii) Der Player beginnt die Animation von vorne.
Zusätzliche Anforderungen	—

Use Case UC2: Erstellen einer Echtzeit-Animation

Tabelle 5.3.: Use Case UC2: Erstellen einer Echtzeit-Animation.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Anwender
Stakeholder und Interessen	Anwender: Möchte eine neue Echtzeit-Animation erstellen. Betrachter: Möchte eine erstellte Echtzeit-Animation ansehen.
Vorbedingungen	Der Editor sichert eine Animation alle n -Minuten in einer temporären Sicherungsdatei.

Erfolgsszenario

1. Der Anwender startet die Editor-Applikation.
2. Der Anwender fügt Elemente zu einer Szene zusammen.
3. Der Anwender legt Start und Ende einer Animation fest.
4. Der Anwender speichert die getätigte Arbeit.
5. Der Anwender exportiert die erstellte Animation.
6. Der Anwender schliesst den Editor.

Erweiterungen

- (a) Schliessen des Editors bei gemachten Änderungen ohne zu speichern
 - (i) Der Anwender schliesst den Editor bei gemachten Änderungen ohne diese zu speichern.
 - (ii) Der Editor weist den Anwender auf die nicht gespeicherten Änderungen hin und bietet die Möglichkeit diese zu speichern, diese nicht zu speichern oder das Schliessen abzubrechen.
- (b) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.

Zusätzliche Anforderungen

—

Use Case UC3: Bearbeiten einer bestehenden Animation

Tabelle 5.4.: Use Case UC3: Bearbeiten einer bestehenden Animation

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Anwender
Stakeholder und Interessen	<p>Anwender: Möchte eine zuvor erstellte Echtzeit-Animation verändern.</p> <p>Betrachter: Möchte eine geänderte Echtzeit-Animation ansehen.</p> <p>Entwickler: Testen einer geänderten Echtzeit-Animation.</p>

Erfolgsszenario

1. Der Anwender startet die Editor-Applikation.
 2. Der Anwender öffnet eine zuvor gespeicherte Echtzeit-Animation.
 3. Der Editor lädt die gewählte Echtzeit-Animation.
 4. Der Anwender nimmt eine oder mehrere Änderungen vor.
 5. Der Anwender speichert die getätigte Arbeit.
 6. Der Anwender exportiert die erstellte Animation.
 7. Der Anwender schliesst den Editor.
-

Erweiterungen

- (a) Schliessen des Editors bei gemachten Änderungen ohne zu speichern
 - (i) Der Anwender schliesst den Editor bei gemachten Änderungen ohne diese zu speichern.
 - (ii) Der Editor weist den Anwender auf die nicht gespeicherten Änderungen hin und bietet die Möglichkeit diese zu speichern, diese nicht zu speichern oder das Schliessen abubrechen.
- (b) Eine zuvor gespeicherte Echtzeit-Animation kann nicht geladen werden.
 - (i) Der Anwender öffnet eine zuvor gespeicherte Echtzeit-Animation.
 - (ii) Der Editor weist den Anwender auf den Umstand hin, dass die Echtzeit-Animation nicht geöffnet werden kann.
- (c) Benötigte zusätzliche Dateien können nicht gefunden werden.
 - (i) Der Anwender öffnet eine zuvor gespeicherte Echtzeit-Animation bei welcher benötigte Dateien (wie zum Beispiel Bitmaps) fehlen.
 - (ii) Der Editor weist den Anwender auf den Umstand hin, öffnet die Echtzeit-Animation dennoch. Er ersetzt die fehlenden Dateien mit Platzhaltern. Ein fehlerfreier Ablauf der Echtzeit-Animation ist nicht gewährleistet.
- (d) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.

Zusätzliche Anforderungen	—
---------------------------	---

Use Case UC4: Exportieren einer Animation

Tabelle 5.5.: Use Case UC4: Exportieren einer Animation.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Anwender

Stakeholder und Interessen

Anwender: Möchte eine erstellte Echtzeit-Animation für die Verwendung im Player exportieren.

Betrachter: Möchte eine Echtzeit-Animation ansehen.

Entwickler: Testen einer Echtzeit-Animation im Player.

Erfolgsszenario

1. Der Anwender startet die Editor-Applikation.
 2. Der Anwender öffnet eine zuvor gespeicherte Echtzeit-Animation oder erstellt eine neue Echtzeit-Animation.
 3. Der Anwender speichert die getätigte Arbeit.
 4. Der Anwender exportiert die erstellte Animation.
 5. Der Editor exportiert die erstellte Animation mit allen benötigten Abhängigkeiten in ein definiertes (Unter-) Verzeichnis.
 6. Der Anwender schliesst den Editor.
-

Erweiterungen

- (a) Schliessen des Editors bei gemachten Änderungen ohne zu speichern
 - (i) Der Anwender schliesst den Editor bei gemachten Änderungen ohne diese zu speichern.
 - (ii) Der Editor weist den Anwender auf die nicht gespeicherten Änderungen hin und bietet die Möglichkeit diese zu speichern, diese nicht zu speichern oder das Schliessen abubrechen.
- (b) Eine zuvor gespeicherte Echtzeit-Animation kann nicht geladen werden.
 - (i) Der Anwender öffnet eine zuvor gespeicherte Echtzeit-Animation.
 - (ii) Der Editor weist den Anwender auf den Umstand hin, dass die Echtzeit-Animation nicht geöffnet werden kann.
- (c) Benötigte zusätzliche Dateien können nicht gefunden werden.
 - (i) Der Anwender öffnet eine zuvor gespeicherte bzw. erstellt eine neue Echtzeit-Animation bei welcher benötigte Dateien (wie zum Beispiel Bitmaps) fehlen.
 - (ii) Der Editor weist den Anwender auf den Umstand hin, exportiert die Echtzeit-Animation dennoch. Er ersetzt die fehlenden Dateien mit Platzhaltern. Ein fehlerfreier Ablauf der Echtzeit-Animation ist nicht gewährleistet.
- (d) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.

Zusätzliche Anforderungen	—
---------------------------	---

Use Case UC5: Bereitstellen neuer Editor-Elemente

Tabelle 5.6.: Use Case UC5: Bereitstellen neuer Editor-Elemente.

Bereich	Player
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Entwickler

Stakeholder und Interessen

Entwickler: Möchte dem Anwender neue Möglichkeiten zur visuellen Gestaltung bieten. Möchte dem Betrachter neue, visuell ansprechende Elemente bieten.
Anwender: Möchte neue Funktionen zur Erstellung von Echtzeit-Animation nutzen können.
Betrachter: Möchte visuell ansprechende Echtzeit-Animationen mit neuen Effekten ansehen.

Erfolgsszenario

1. Der Entwickler entwickelt neuen Inhalt in Form eines Knotens für den Graphen (dies kann zum Beispiel ein prozedurales Mesh oder ein dedizierter Shader sein).
2. Der Entwickler startet den Editor.
3. Der neue Knoten steht im Editor automatisch zur Verfügung und kann genutzt werden.
4. Der Entwickler schliesst den Editor.

Erweiterungen

- (a) Erfassen neuer Knoten direkt im Editor
 - (i) Der Entwickler startet den Editor.
 - (ii) Der Entwickler öffnet die Bibliothek mit allen Knoten-Typen.
 - (iii) Der Entwickler fügt der Bibliothek einen neuen Knoten hinzu.
 - (iv) Der Entwickler füllt den Knoten mit den entsprechenden Details und speichert diesen.
 - (v) Der neu erstellte Knoten wird persistiert.
 - (vi) Der neu erstellte Knoten erscheint in der Bibliothek und ist per sofort auswählbar.
 - (vii) Der Entwickler schliesst den Editor.
- (b) Fehlerhafter Inhalt des Knotens
 - (i) Der Entwickler fügt dem Knoten keinen oder fehlerhaften Inhalt hinzu.
 - (ii) Der Editor weist den Entwickler auf den Umstand hin. Der Knoten kann nicht verwendet werden.
- (c) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.

Zusätzliche Anforderungen

Use Case UC6: Erstellen einer neuen Szene

Tabelle 5.7.: Use Case UC6: Erstellen einer neuen Szene.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Anwender
Stakeholder und Interessen	Anwender: Möchte eine neue Szene einer Echtzeit-Animation erstellen. Betrachter: Möchte eine erstellte Szene ansehen.
Vorbedingungen	<ul style="list-style-type: none"> • Die Editor-Applikation ist gestartet.
Erfolgsszenario	<ol style="list-style-type: none"> 1. Der Anwender startet die Editor-Applikation. 2. Der Anwender klickt mit der rechten Maustaste die Root-Szene in der Bibliothek an. 3. Der Anwender wählt im Kontext-Menü den Menüpunkt zum Erstellen einer neuen Szene. 4. Der Editor erstellt mittels dem Szenegraphen eine neue Szene und fügt diese der Liste von Szenen hinzu. 5. Die Szene wird entsprechend im Szenegraphen als Unterknoten des Root-Knotens dargestellt.
Erweiterungen	<ol style="list-style-type: none"> (a) Schliessen des Editors bei gemachten Änderungen ohne zu speichern <ol style="list-style-type: none"> (i) Der Anwender schliesst den Editor bei gemachten Änderungen ohne diese zu speichern. (ii) Der Editor weist den Anwender auf die nicht gespeicherten Änderungen hin und bietet die Möglichkeit diese zu speichern, diese nicht zu speichern oder das Schliessen abzubrechen. (b) Absturz/Ausfall des Systems <ol style="list-style-type: none"> (i) Das System stürzt an einem beliebigen Punkt ab. (ii) Neustart des Editors. (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.
Zusätzliche Anforderungen	—

Use Case UC7: Hinzufügen eines neuen Knotens im Graphen einer Szene

Tabelle 5.8.: Use Case UC7: Hinzufügen eines neuen Knotens im Graphen einer Szene.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Anwender
Stakeholder und Interessen	Anwender: Möchte neuen Inhalt zu einer Szene einer Echtzeit-Animation hinzufügen. Betrachter: Möchte neu erstellten Inhalt ansehen. Entwickler: Möchte, dass seine erstellten Knoten-Typen produktiv eingesetzt werden können.
Vorbedingungen	<ul style="list-style-type: none"> • Die Editor-Applikation ist gestartet. • Es sind Knoten-Typen zum Hinzufügen vorhanden. • Eine beliebige Szene ist angewählt.
Erfolgsszenario	<ol style="list-style-type: none"> 1. Der Anwender klickt mit der rechten Maustaste auf eine leere Fläche der Szene. 2. Das Kontext-Menü zum Hinzufügen neuer Knoten öffnet sich. 3. Der Anwender wählt im Kontext-Menü den gewünschten Knoten-Typen aus. 4. Der Editor erstellt anhand der aktuellen Szene einen neuen Knoten im Szenegraphen. 5. Der Editor fügt den neu erstellten Knoten zur Ausgabe des Szenegraphen hinzu. 6. Der Editor sendet via Szenegraph das Signal, dass ein neuer Knoten hinzugefügt wurde. 7. Der Editor merkt sich, dass sein Fenster verändert wurde. 8. Der Editor aktualisiert die Anzahl der Knoten in den einzelnen Szenen.

Erweiterungen

- (a) Schliessen des Editors bei gemachten Änderungen ohne zu speichern
 - (i) Der Anwender schliesst den Editor bei gemachten Änderungen ohne diese zu speichern.
 - (ii) Der Editor weist den Anwender auf die nicht gespeicherten Änderungen hin und bietet die Möglichkeit diese zu speichern, diese nicht zu speichern oder das Schliessen abubrechen.
- (b) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.

Zusätzliche Anforderungen

—

Use Case UC8: Verbinden eines Knotens im Graphen einer Szene

Tabelle 5.9.: Use Case UC8: Verbinden eines Knotens im Graphen einer Szene.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Anwender
Stakeholder und Interessen	Anwender: Möchte (Teil-) Inhalt einer Szene einer Echtzeit-Animation sichtbar machen. Betrachter: Möchte neu erstellten Inhalt ansehen. Entwickler: Möchte, dass seine erstellten Knoten-Typen produktiv eingesetzt werden können.
Vorbedingungen	<ul style="list-style-type: none">• Die Editor-Applikation ist gestartet.• Es sind Knoten-Typen zum Hinzufügen vorhanden.• Eine beliebige Szene ist ausgewählt.• Die Szene verfügt bereits über Knoten.

Erfolgsszenario

1. Der Anwender klickt mit der linken Maustaste auf die Ausgangs-Buchse des gewünschten Knotens und hält die linke Maustaste gedrückt.
2. Der Editor erstellt eine Verbindungslinie mit Ursprung in der Ausgangs-Buchse des Knotens und der aktuellen Position des Mauszeigers als Ziel.
3. Der Anwender verschiebt den Mauszeiger über die Eingangs-Buchse eines kompatiblen Knotens (zum Beispiel einen Szene-Knoten zum Hauptausgangs-Knoten) und lässt die linke Maustaste los.
4. Der Editor erstellt eine neue Verbindung zwischen dem dem Ausgangs- und dem Zielknoten.
5. Der Editor fügt die neu erstellte Verbindung zur Ausgabe des Szenegraphen bzw. der Szene hinzu.
6. Der Editor sendet via Szenegraph das Signal, dass eine neue Verbindung hinzugefügt wurde.
7. Der Editor merkt sich, dass sein Fenster verändert wurde.

Erweiterungen

- (a) Schliessen des Editors bei gemachten Änderungen ohne zu speichern
 - (i) Der Anwender schliesst den Editor bei gemachten Änderungen ohne diese zu speichern.
 - (ii) Der Editor weist den Anwender auf die nicht gespeicherten Änderungen hin und bietet die Möglichkeit diese zu speichern, diese nicht zu speichern oder das Schliessen abubrechen.
- (b) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.

Zusätzliche Anforderungen

Use Case UC9: Hinzufügen eines Schlüsselbildes eines Parameters

Tabelle 5.10.: Use Case UC9: Hinzufügen eines Schlüsselbildes eines Parameters eines Knotens.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Akteur	Anwender
Stakeholder und Interessen	Anwender: Möchte einen Parameter eines Knotens einer Szene animieren. Betrachter: Möchte animierten Inhalt sehen.
Vorbedingungen	<ul style="list-style-type: none"> • Die Editor-Applikation ist gestartet. • Es sind Knoten-Typen zum Hinzufügen vorhanden. • Eine beliebige Szene ist angewählt. • Die Szene verfügt bereits über Knoten.
Erfolgsszenario	<ol style="list-style-type: none"> 1. Der Anwender klickt mit der linken Maustaste auf den Knoten, dessen Parameter er animieren möchte. 2. Der Knoten wird im Graphen als markiert dargestellt. 3. Die Parameter des Knoten werden im Parameter-Fenster dargestellt. 4. Der Anwender verschiebt den Marker der Zeitachse auf die gewünschte Position. 5. Der Anwender klickt im Parameter-Fenster bei dem gewünschten Parameter auf die Schaltfläche zur Erstellung eines Schlüsselbildes. 6. Der Editor erstellt ein neues Schlüsselbild an der gewählten Stelle für den gewählten Parameter. Existieren frühere oder spätere Schlüsselbilder, wird zwischen diesen linear interpoliert. 7. Der Editor fügt die neu erstellte Animation zur Ausgabe der Szene hinzu. 8. Der Editor sendet via Szenegraph das Signal, dass ein neues Schlüsselbild hinzugefügt wurde. 9. Der Editor merkt sich, dass sein Fenster verändert wurde.

Erweiterungen

- (a) Schliessen des Editors bei gemachten Änderungen ohne zu speichern
 - (i) Der Anwender schliesst den Editor bei gemachten Änderungen ohne diese zu speichern.
 - (ii) Der Editor weist den Anwender auf die nicht gespeicherten Änderungen hin und bietet die Möglichkeit diese zu speichern, diese nicht zu speichern oder das Schliessen abubrechen.
- (b) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her.

Zusätzliche Anforderungen

—

Use Case UC10: Auswerten und Darstellen eines Knotens

Tabelle 5.11.: Use Case UC10: Auswerten und Darstellen eines Knotens.

Bereich	Editor und Player
Stufe (level)	Ziel des Benutzers und Unterfunktion
Primärer Akteur	Anwender
Stakeholder und Interessen	Anwender: Möchte das (visuelle) Ergebnis eines Knotens sehen. Betrachter: Möchte animierten Inhalt sehen. Entwickler: Testen von Knoten und Unterknoten.
Vorbedingungen Editor	<ul style="list-style-type: none"> • Die Editor-Applikation ist gestartet. • Es sind Knoten-Typen zum Hinzufügen vorhanden. • Eine beliebige Szene ist ausgewählt. • Die Szene verfügt bereits über Knoten.

Vorbedingungen Player

- Die Player-Applikation ist gestartet.
- Es ist ein Demo-Skript mitsamt Inhalt zum Ausführen vorhanden.
- Das Demo-Skript verfügt über mindestens eine Szene.
- Die Szene verfügt über Knoten.
- Die Szene ist mit dem Haupt-Ausgabeknoten verbunden.

Erfolgsszenario Editor

1. Der Anwender klickt mit der linken Maustaste auf den Knoten, dessen Ausgabe er sehen möchte.
2. Der Knoten wird im Graphen als markiert dargestellt.
3. Die Parameter des Knoten werden im Parameter-Fenster dargestellt.
4. Der Editor evaluiert den Graphen des Knoten rekursiv zum aktuellen Zeitpunkt der Zeitachse.
5. Der Editor stellt die berechnete Ausgabe im Rendering-Ansichtsfenster dar.

Erfolgsszenario Player

1. Der Player lädt das Demoskript mitsamt all seinen Ressourcen und evaluiert. Dabei wird zuerst der Haupt-Ausgabeknoten evaluiert und dann rekursiv traversiert.
 2. Ist eine Animation vorhanden, so wird diese nun gestartet und der Graph des Haupt-Ausgabeknotens rekursiv zum aktuellen Zeitpunkt der Animation evaluiert.
 3. Ist keine Animation vorhanden, so wird immer der Zeitpunkt 0 evaluiert.
 4. Der Player stellt die berechnete Ausgabe im Rendering-Ansichtsfenster dar.
-

Erweiterungen

- (a) Absturz/Ausfall des Systems
 - (i) Das System stürzt an einem beliebigen Punkt ab.
 - (ii) Neustart des Editors/Players.
 - (iii) Der Editor stellt den ihm zuletzt bekannten Punkt der Animation wieder her. Der Player beginnt die Animation von vorne — sofern eine solche vorhanden ist.

Zusätzliche Anforderungen

5.1.4. Zusätzliche Anforderungen

Software

Durch die vorhergehende Projektarbeit — MTE7101 (siehe [1]) — sowie persönlicher Erfahrungen bei diversen Projekten, sieht der Autor die Software gemäss Tabelle 5.12 zur Umsetzung vor. Alle genannten Komponenten — ausser Qt — beziehen sich auf den *Player*, als auch auf den *Editor*. Der Player benötigt kein Qt, da er kein Frontend hat und möglichst schlank gehalten werden soll.

Tabelle 5.12.: Mögliche Software/Technologien

Komponente	Version	Beschreibung	Verweise
C++	11	Objektorientierte Programmiersprache	2
OpenGL	4.5	Plattformunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafikanwendungen [5]	3
Qt	5.7	“Qt ist eine C++-Klassenbibliothek für die plattformübergreifende Programmierung grafischer Benutzeroberflächen. Außerdem bietet Qt umfangreiche Funktionen zur Internationalisierung sowie Datenbankfunktionen und XML-Unterstützung an und ist für eine große Zahl an Betriebssystemen bzw. Grafikplattformen, wie X11 (Unix-Derivate), OS X, Windows, iOS und Android erhältlich.” [6].	4
GLFW	3.1.2	OpenGL-Bibliothek, welche die Erstellung und Verwaltung von Fenstern sowie OpenGL-Kontexte vereinfacht [7].	5
GLEW	1.13	OpenGL Extension Wrangler. Bibliothek zum Abfragen und Laden von OpenGL-Erweiterungen (Extensions) [8].	6
GLM	0.9.7.6	Header-only C++ Mathematik-Bibliothek, basierend auf der Spezifikation der OpenGL Shader-Sprache GLSL. Sie bietet eine Vielzahl an Datentypen wie etwa Vektoren, Matrizen oder Quaternionen.	7
CMake	3.3.2	Software zur Verwaltung von Build-Prozessen von Software	8
LLVM	3.8.1	Ansammlung von modularen und wiederverwendbaren Compilern und Toolchains.	9
Clang	3.8.1	Compiler-Frontend für LLVM.	10

² http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

³ <https://www.opengl.org/registry/doc/glspec45.core.pdf>

⁴ <https://www.qt.io>

⁵ <http://www.glfw.org>

⁶ <http://glew.sourceforge.net>

⁷ <https://glm.g-truc.net>

⁸ <https://www.cmake.org>

⁹ <http://llvm.org>

¹⁰ <http://clang.llvm.org>

5.2. Komponenten

Ausgehend von den Anforderungen (5.1) können einzelne Komponenten der Applikation abgeleitet werden. Einzelne Teile davon wurden schon durch die Vision (??) definiert beziehungsweise aus dieser gewonnen.

Dieser Prozess entspricht nicht direkt dem Vorgehen gemäss [3] beziehungsweise dem UP, der Autor dieser Projektarbeit ist jedoch der Ansicht, dass dieser Abschnitt eine Brücke zwischen Anforderungen und der (Software-) Modellierung bildet. Zudem bietet dieser Abschnitt eine relativ bildliche Beschreibung, was dem Verständnis des Gesamtkonzeptes sicher zuträglich ist. Am ehesten entspricht dieser Abschnitt den Komponenten-Diagrammen in [3, S. 653 bis 654].

Die Applikation besteht aus zwei Applikationen: Einem *Player*, welcher dem Abspielen von Echtzeit-Animationen dient, sowie einem *Editor*, welcher der Erstellung und Verwaltung von Echtzeit-Animationen dient.

5.2.1. Player


Der *Player* liest die vom *Editor* exportierten Echtzeit-Animationen. Er bietet vor dem Abspielen die Auswahl der Auflösung, des Seitenverhältnisses, Antialiasing und ob die Animation im Vollbild-Modus abgespielt werden soll.

5.2.2. Editor

Der *Editor* erlaubt das Erstellen und Bearbeiten von Echtzeit-Animationen. Diese können schliesslich inklusive den dazugehörigen Dateien, wie zum Beispiel Bitmaps oder Modellen, exportiert werden.

Abbildung 5.2 zeigt die einzelnen Komponenten des Editors. Nachfolgend findet sich eine Beschreibung dieser.

Bibliothek

Das Element  in Abbildung 5.2 zeigt die (Szenen-) Bibliothek. Diese beinhaltet alle Szenen einer Echtzeit-Animation. Es können neue Szenen angelegt und auch bestehende Szenen gelöscht werden. Wird ein neues Projekt erstellt, so verfügt dieses immer über die "Root"-Szene. Diese beinhaltet den Haupt-Ausgabeknoten des Graphen (5.2.2), welcher schliesslich zum Abspielen evaluiert wird, und kann nicht gelöscht werden. Wird eine Szene mit der Maus angewählt, so wird deren Inhalt im Graphen (5.2.2) dargestellt.

¹¹ <http://www.boost.org>

¹²Eigene Darstellung mittels Pencil.

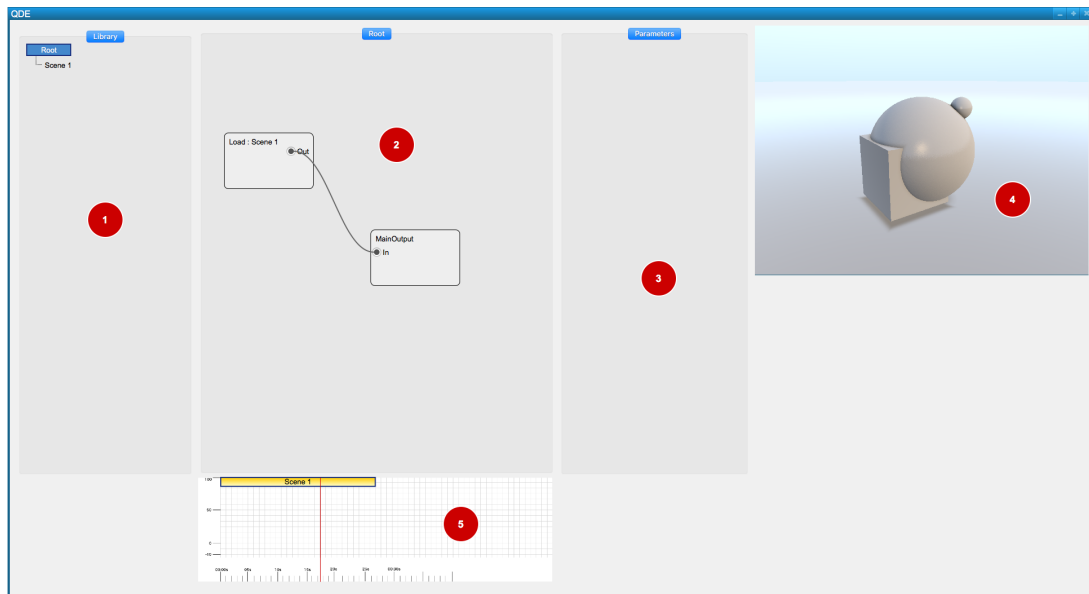


Abbildung 5.2.: Einzelne Komponenten des Editors ¹²

Graph

Das Element 2 in Abbildung 5.2 zeigt den Graphen einer Szene. Dieser beinhaltet sämtliche Knoten einer Szene. Mittels Kontextmenü können neue Knoten eingefügt und bestehende Knoten gelöscht werden. Wird ein Knoten angewählt, so wird dieser einerseits im Rendering-Ansichtsfenster (5.2.2) dargestellt, andererseits werden dessen Eigenschaften im Parameter-Fenster (5.2.2) angezeigt.

Folgende Typen von Knoten sind geplant:

- Scene
- TimelineClip
- Model
- Camera
- Light
- Material
- Operator
- Effect


Parameter

Das Element 3 in Abbildung 5.2 zeigt die Parameter des aktuell gewählten Knoten im Graphen (5.2.2). Neben jedem Parameter befindet sich eine Schaltfläche zum Setzen von Schlüsselbildern (Keyframes) in der Zeitachse (Timeline, 5.2.2). Wird die Schaltfläche betätigt, so wird bei dem aktuell ausgewählten Zeitpunkt der Zeitachse ein Schlüsselbild gesetzt.

Rendering

Das Element 4 in Abbildung 5.2 zeigt das Rendering-Ansichtsfenster. Dieses stellt den Inhalt des aktuell gewählten Knotens dar. Die Art des Knotens ist dabei nicht beschränkt, es kann dies eine Szene, aber zum Beispiel auch ein einzelnes Modell sein. Es wird immer der gesamte vorhergehende (Teil-) Baum des Knotens evaluiert.

Zeitachse

Die Zeitachse wird mit  in Abbildung 5.3 dargestellt. Sie bildet das zeitliche Geschehen einer Echtzeit-Animation ab. Alle Knoten vom Typ Timeline-Clip werden am oberen Rand des Fensters in deren zeitlicher Reihenfolge abgebildet. Wird im Graph (5.2.2) ein Knoten mit animierten Parametern (5.2.2) angewählt, so sind diese ersichtlich. Vertikal wird der Wertebereich, horizontal die Zeitachse in Sekunden dargestellt. Ein vertikal verlaufender, roter Marker zeigt die aktuelle zeitliche Position der Echtzeit-Animation an.

Die untenstehende Abbildung 5.3 zeigt ein Beispiel, wie eine typische Szene mit animierten Parametern aussehen könnte.

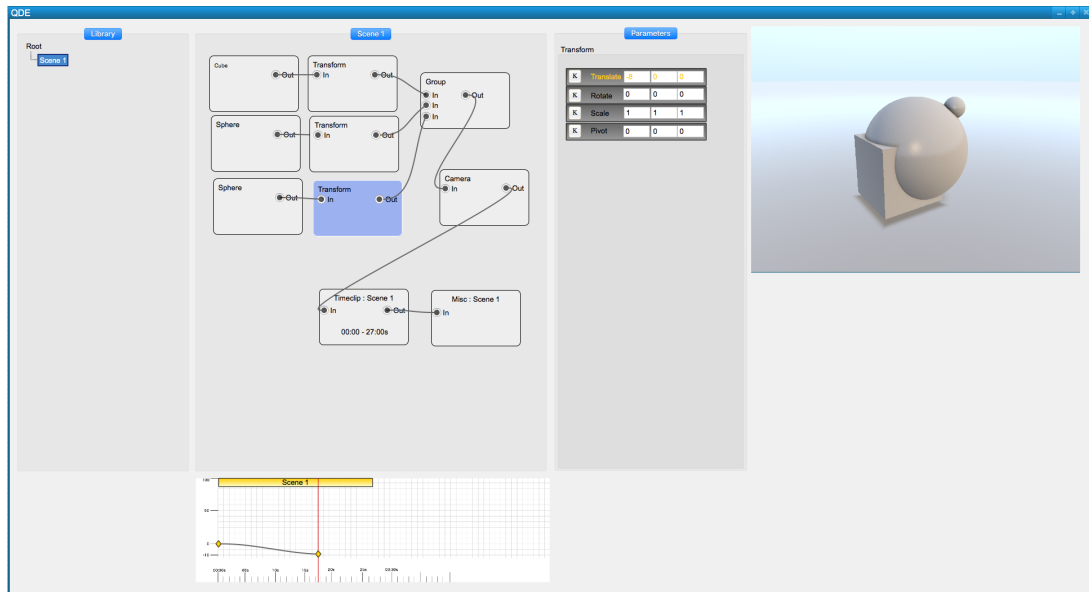


Abbildung 5.3.: Beispiel-Szene innerhalb des Editors ¹³

5.3. Domänenmodell

Der wesentliche Schritt der objektorientierten Analyse ist die Zerlegung einer Domäne in essentielle Konzepte oder Objekte [3, S. 134].

In UML wird ein Domänenmodell typischerweise als eine Menge von Klassendiagrammen ohne Operationen dargestellt. Es liefert eine konzeptuelle Perspektive und kann folgende Elemente beinhalten [3, S. 134]:

- Objekte der Domäne oder konzeptuelle Klassen
- Relationen zwischen konzeptuellen Klassen
- Attribute der konzeptuellen Klassen

Larman weist darauf hin, dass nicht versucht werden sollte von Beginn weg ein möglichst genaues, vollständiges oder “korrektes” Domänenmodell zu erstellen. Ein solcher Ansatz führt zu “analysis paralysis” und sollte daher vermieden werden, da dies wenig bis keinen Mehrwert bringt [3, S. 133].

Da angedacht ist, dass die Software aus zwei Applikationen, dem *Player* sowie dem *Editor*, besteht, wird für jede Applikation ein Domänenmodell erstellt.

¹³Eigene Darstellung mittels Pencil.

5.3.1. Player

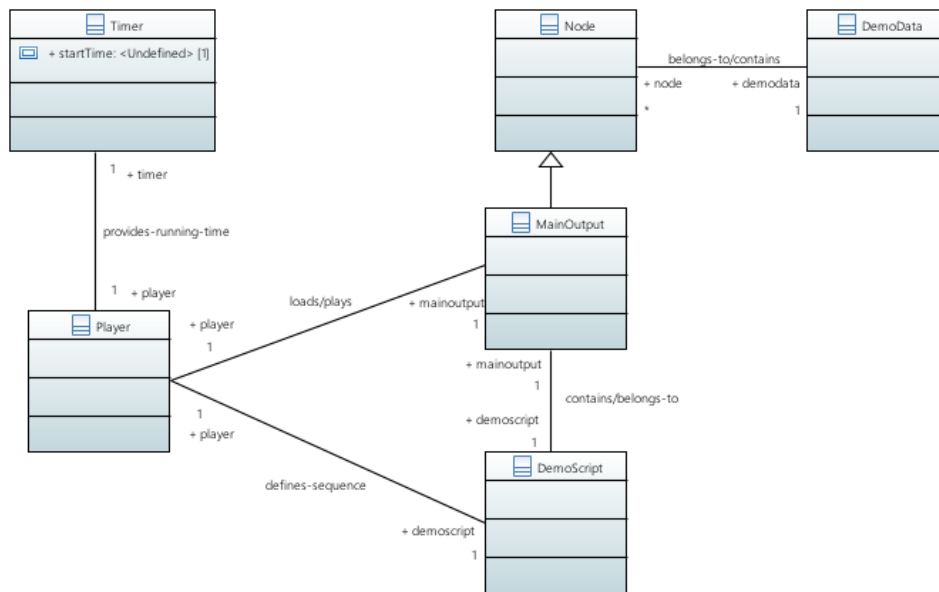


Abbildung 5.4.: Domänenmodell der Player-Applikation¹⁴

Abbildung 5.4 zeigt das Domänenmodell der Player-Applikation. Das Modell ist bewusst minimal gehalten und zeigt nur die nötigsten Komponenten. Im Zentrum steht das Player-Objekt. Dieses liest ein DemoScript, welches eine exportiert Echtzeit-Animation des Editors darstellt. Ausgehend vom DemoScript kann schliesslich der Hauptknoten des Graphen gefunden evaluiert werden. Ausgehend von diesem wird so die gesamte Echtzeit-Animation aufgebaut und wiedergeben.

Viele essentielle Konzepte beziehungsweise Objekte fehlen in diesem Modell bewusst. Diese werden während den Iterationen der darauffolgenden Projektarbeit erarbeitet. Ein Teil davon findet sich im Domänenmodell des Editors (5.3.2) sowie im Klassendiagramm des Prototypen (??).

¹⁴Eigene Darstellung mittels Papyrus.

5.3.2. Editor

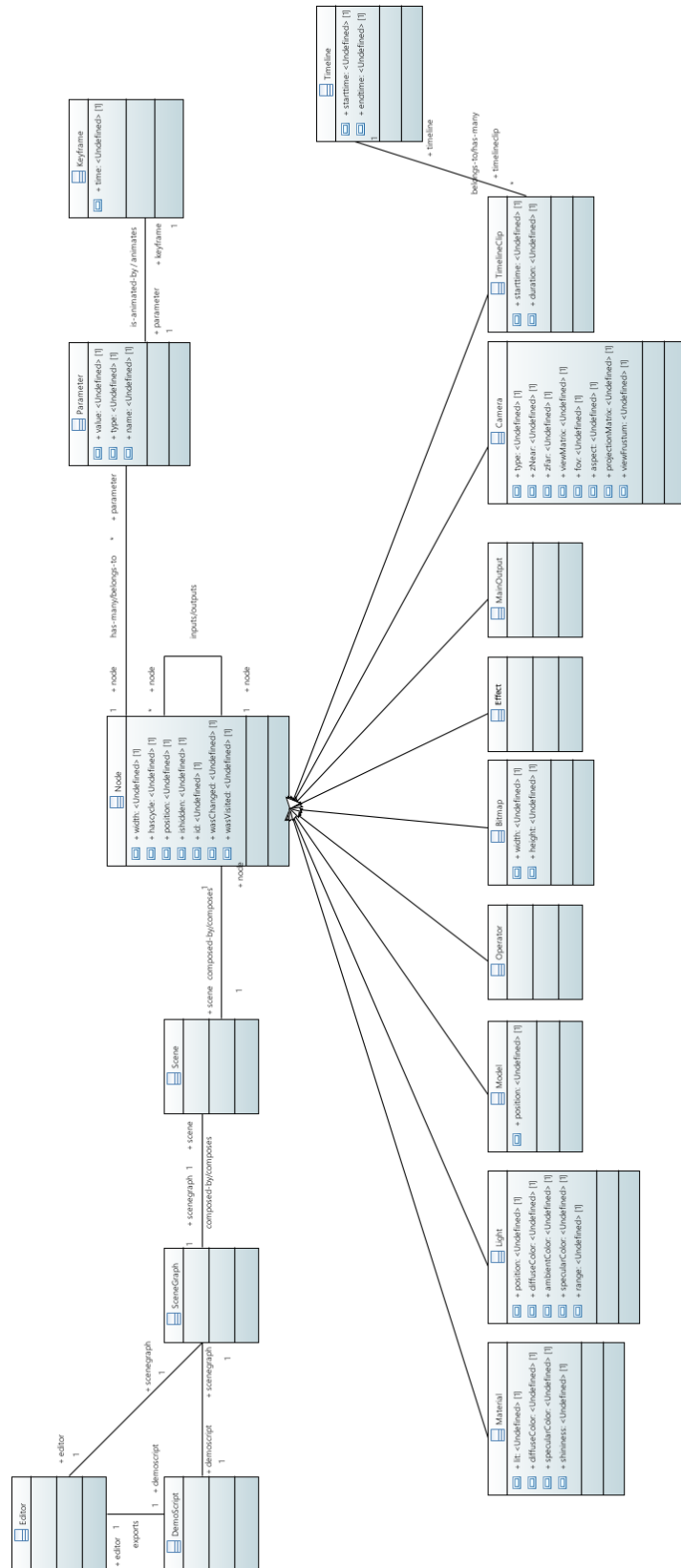


Abbildung 5.5.: Domänenmodell der Editor-Applikation¹⁵

Abbildung 5.5 zeigt das Domänenmodell der Editor-Applikation. Analog dem vorherigen Modell steht hier das Editor-Objekt im Zentrum. Dieses bildet die Schnittstelle zwischen der grafischen Benutzeroberfläche (GUI) und der Applikationslogik.

Die Komponenten der grafischen Benutzeroberfläche wurden in diesem Modell bewusst weggelassen, da dies das Modell nur unnötig vergrössern würde. Die eigentliche Logik findet sich in den essentiellen Konzepten beziehungsweise in den hier gezeigten Objekten. Die grafische Benutzeroberfläche bildet diese nur ab respektive Kopien davon.

Im mittleren Teil ist mit dem Node-Objekt und dessen Spezialisierung die gesamte Struktur des Graphen angedeutet. Dies werden schliesslich die Objekte sein, welche ein Anwender nutzen kann um Echtzeit-Animationen zu erstellen.

Im rechten Teil des Modelles stellen die Objekte Parameter, Keyframe, TimelineClip und Timeline Elemente zur Animation dar. Pro Parameter kann zu einem bestimmten Zeitpunkt der Zeitachse ein Schlüsselbild (Keyframe) gesetzt werden. Timeline stellt die gesamte Zeitachse und TimelineClip einzelne Elemente dieser dar. Es handelt sich bei Letzteren um Szenen.

Auch hier fehlen wiederum etliche Details, wie zum Beispiel das gesamte Rendering inklusive den Shadern. Ein Teil davon ist wiederum im Klassendiagramm des Prototypen (??) ersichtlich.

Wie Eingangs erwähnt, ist es nicht das Ziel ein vollständiges oder “korrektes” Domänenmodell zu erstellen. Es geht viel mehr um einen ersten Anstoss zur eigentlichen Implementierung. Alle weiteren Details werden während den Iterationen der darauffolgenden Projektarbeit erarbeitet.

5.4. Sequenz-Diagramme

Gemäss [3] stellen Sequenz-Diagramme Ereignisse, welche externe Akteure auslösen bzw. generieren, deren Ablauf sowie Ereignisse zwischen Systemen eines spezifischen Szenarios eines Use Cases dar.

Da Sequenz-Diagramme schnell eine gewisse Grösse und auch Komplexität annehmen wird in dieser Projektarbeit darauf verzichtet ein Sequenz-Diagramm für alle Use Cases zu erstellen. Gerade bei “UC2: Erstellen einer Echtzeit-Animation” würde dies ansonsten grosse Ausmasse annehmen. Bei Bedarf können diese bei den einzelnen Iterationen bzw. Phasen (Elaboration, Construction und Transition) der darauffolgenden Projektarbeit noch erarbeitet werden. An dieser Stelle wird daher nur das Sequenz-Diagramm für den ersten Use Case, “UC1: Betrachten einer Echtzeit-Animation” dargestellt. Dies deckt bereits einen Teil des Editors mit ab.

¹⁵Eigene Darstellung mittels Papyrus.

Der Player öffnet einen Setup-Dialog, mittels welchem der Betrachter die gewünschte Auflösung sowie Bit-Tiefe wählt.

Der Player entpackt und lädt dann das Demoskript. Darauffolgend wird die Engine und die Grafik-Schnittstelle (und somit auch das Hauptfenster des Players) initialisiert. Danach importiert der Player das Demoskript.

Ausgehend von diesem findet er den Hauptknoten des Graphen, DemoNode. Auf diesem, beziehungsweise auf dessen Interface oder Hauptklasse — Node — ruft er dann die Process-Methode zum Zeitpunkt 0 auf. Dies hat zur Folge, dass der komplette Szenegraph aufgebaut und (vor-) evaluiert wird. Details dieses Schrittes respektive dieser Schritte sind im Sequenz-Diagramm mittels gelben Notizen annotiert.

Der Player startet darauf den Timer um die Zeit messen zu können. Darauffolgend wird die Hauptschleife des Players gestartet. Es handelt sich dabei um eine While-Schleife, welche als Abbruchbedingung entweder den Input des Betrachters (in Form der Escape-Taste beispielsweise) oder keine gültige Sequenz mehr hat.

In der Haupt-Schleife wird jeweils der aktuelle Zeit-Stempel geholt und danach wird dieser als Argument für die Process-Methode des Hauptknotens des Graphen verwendet. Diese ist nicht nochmal im Detail aufgeführt, da der Ablauf derselbe wie beim vorherigen Aufruf ist. Schliesslich wird das berechnete Bild zum aktuellen Zeitpunkt mittels Renderer dargestellt.

Bricht der Betrachter die Haupt-Schleife mittels Taste ab oder ist keine gültige Sequenz mehr vorhanden, beendet sich der Player.

5.5. Logische Architektur

Die logische Architektur zeigt das Gesamtbild der Software-Klassen in Form von Paketen (bzw. Namespaces), Subsystemen und Layern [3, S. 199]. Bei Layern handelt es sich um eine grobe Gruppierung von Klassen, Paketen oder Subsystemen, welche zusammenhängen [3, S. 199].

Eine strikt abgestufte (strict layered) Architektur sieht vor, dass ein Layer nur Services beziehungsweise Schnittstellen der darunterliegenden Schicht aufruft [3, S. 200].

Da dies in der Praxis je nach Fall der Anwendung jedoch schwer umzusetzen ist, wird häufig auch von einer locker abgestuften (relaxed layered) Architektur gesprochen [3, S. 200]. Diese ermöglicht es, dass höhere Stufen ohne Weiteres mit wesentlich tieferen Stufen kommunizieren. Eine solche Architektur wird hier angestrebt.

Die Verwendung dieses Design-Musters reduziert Kupplung (coupling) sowie Abhängigkeiten, erhöht Kohäsion und Klarheit [3, S. 204].

Um dies auch hinsichtlich der grafischen Benutzeroberfläche zu gewährleisten, wurde zudem das Prinzip der Trennung von Modellen und Ansichten (model-view separation principle) angewendet. Dieses sagt aus, dass Modelle beziehungsweise Objekte einer Domäne kein direktes Wissen über Objekte der grafischen Benutzeroberfläche haben sollen [3, S. 209].

Die Trennung von Modellen und Ansichten führt zu oder unterstützt kohäsive (geschlossene) Modelle, welche auf die Prozesse der Domäne fokussiert sind anstatt auf die grafische Benutzeroberfläche [3, S. 210].

Da diese Trennung doch sehr strikt ist, kann das Observer-Muster als weniger strikte Version angesehen werden [3, S. 210]. Ein Domänen-Objekt sendet Nachrichten zu reinen Schnittstellen von Objekten der grafischen Benutzeroberfläche. Somit weiss das Domänen-Objekt nicht, dass es sich bei dem angesprochenen Objekt um ein Objekt der grafischen Benutzeroberfläche handelt [3, S. 210]. Dieser Schritt wurde in dieser Version jedoch nicht umgesetzt. Es wird sich in der darauffolgenden Projektarbeit zeigen, ob dies nötig ist.

Als zusätzliche Abstraktion wurden noch Controller eingeführt. Controller stellen Workflow-Objekte der Applikationsschicht dar [3, S. 209].

Die logische Architektur umfasst folgende Elemente:

- **UI**
Alle Elemente der grafischen Benutzeroberfläche.
- **Application**
Controller / Workflow-Objekte.
- **Domain**
Modelle / Applikationslogik.
- **Technical services**
Technische Infrastruktur wie Grafik, Erzeugung von Fenstern etc.
- **Foundation**
Grundlegende Elemente, low-level technical services wie Timer, Array oder andere Datenklassen.

Sowohl für den Player als auch für den Editor ist dieselbe logische Architektur vorgesehen. Ein grosser Teil der Pakete überschneidet sich bei beiden Komponenten, da beide dieselbe (Daten-) Struktur nutzen. Beim Paket-Diagramm des Players wurde dies jedoch der Übersicht halber, sowie zur Vermeidung von unnötiger Redundanz weggelassen. Auch hier gilt wiederum, dass essentielle Konzepte beziehungsweise Objekte fehlen. Diese werden während den Iterationen der darauffolgenden Projektarbeit erarbeitet. Es sind jedoch bereits mehr Elemente angedeutet, als in den vorherigen Diagrammen.

Abbildung 5.7 zeigt das Paket-Diagramm des Players, Abbildung 5.8 dies des Editors. Blau-graue Elemente stellen dabei selbst entwickelte Pakete, violette Elemente externe Pakete von Drittpersonen dar.

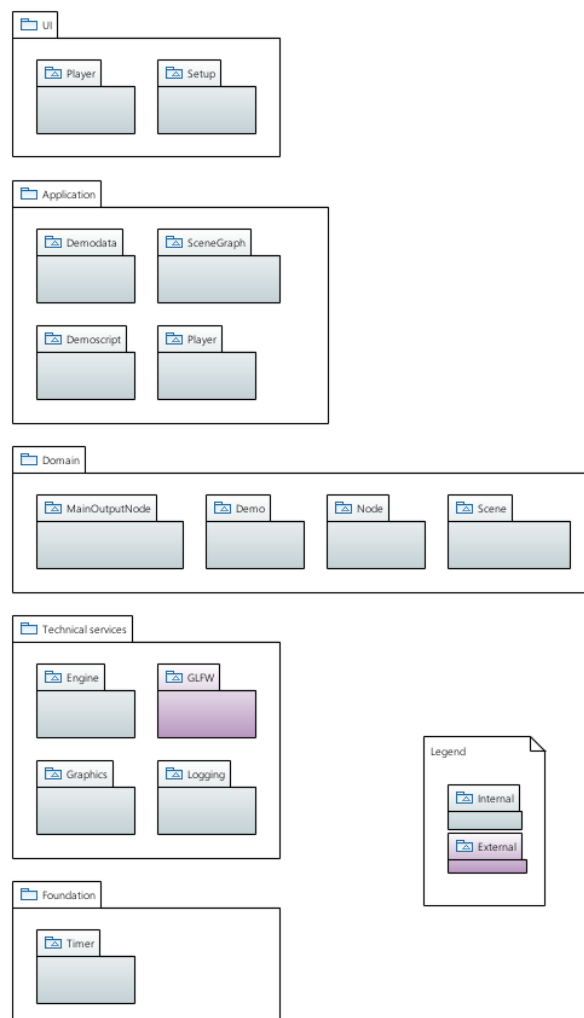


Abbildung 5.7.: Paket-Diagramm der Player-Applikation¹⁷

5.6. Klassendiagramme

Klassendiagramme sind Teil des Design-Modelles und illustrieren Klassen, Interfaces und deren Beziehungen [3, S. 249 bis 251]. Sie sind also eine grafische Darstellung der statischen Sicht einer Software **rumbaugh_unified_2004** Ein Klassendiagramm beinhaltet diverse konkrete Elemente, welche auf das Verhalten der Software bezogen sind, wie zum Beispiel Methoden. Deren Dynamik wird allerdings in anderen Diagrammen, wie dem Statechart-Diagramm oder dem Kommunikations-Diagramm festgehalten **rumbaugh_unified_2004**

Die Notation ist dieselbe oder zumindest sehr ähnlich wie bei dem Domänenmodell. Letzteres zeigt aber mehr eine konzeptuelle Sicht. Domänenmodelle können als Klassendiagramme aus einer konzeptuellen Sicht bezeichnet werden [3, S. 249]. Klassendiagramme selbst repräsentieren also mehr eine Software-Sicht und daher einen Teil der Implementation.

Wie bereits in den vorherigen Modell, fehlen auch hier einige essentielle Konzepte beziehungsweise Objekte bewusst. Diese werden während den Iterationen der darauffolgenden Projektarbeit erarbeitet. Ein Teil ist im Player angedeutet, ein anderer Teil im Editor. Würde man das gesamte Klassendiagramm abbilden wollen, würde dies schnell unübersichtlich. Ein weiterer Teil findet sich im Klassendiagramm des Prototypen (??). Ein detaillierteres Bild der Klassen wird bei den einzelnen Iterationen bzw. Phasen (Elaboration, Construction und Transition) im Rahmen der darauffolgenden Projektarbeit noch im Detail erarbeitet. Gegebenenfalls ist es dort dann jedoch sinnvoller nur Ausschnitte pro Paket zu zeigen und die Interaktion zwischen den Schichten mittels Interfaces darzustellen.

Abbildung 5.9 zeigt das Klassendiagramm des Players, Abbildung 5.10 dies des Editors.

Die verschiedenen Pakete respektive Layer wurden farblich unterschieden. Die farblichen Konventionen sind die folgenden:



UI: Alle Elemente der grafischen Benutzeroberfläche.



Application: Controller / Workflow-Objekte.



Domain: Modelle / Applikationslogik.



Technical services: Technische Infrastruktur wie Grafik, Erzeugung von Fenstern etc.



Foundation: Grundlegende Elemente, low-level technical services wie Timer, Array oder andere Datenklassen.

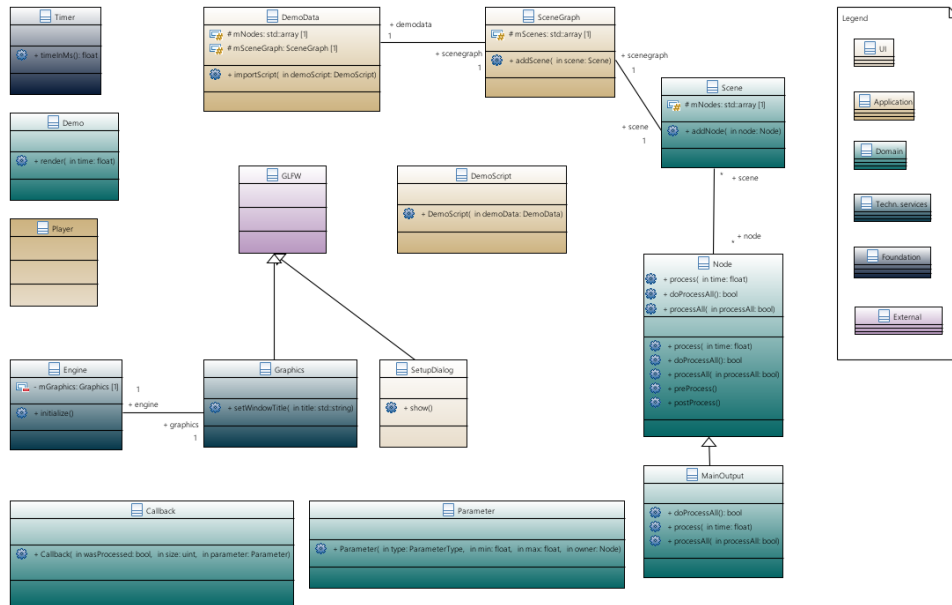


Abbildung 5.9.: Klassendiagramm der Player-Applikation¹⁹

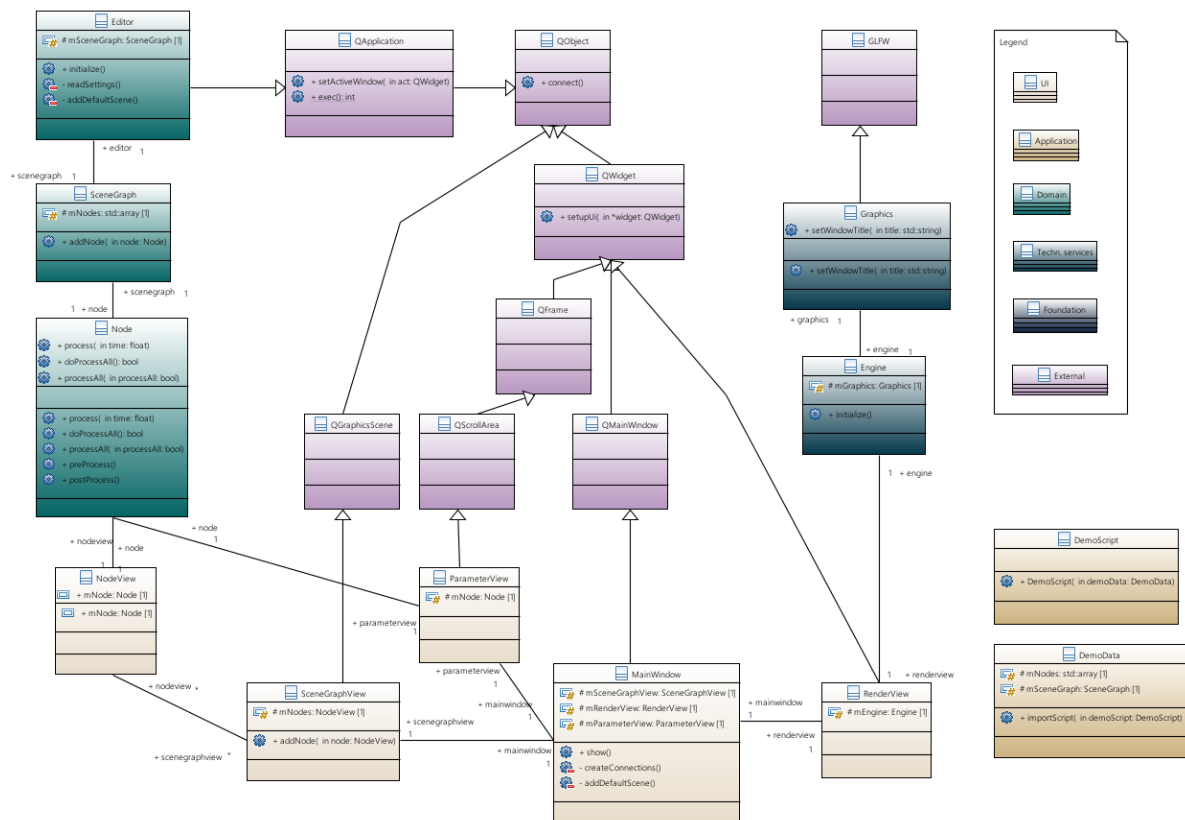


Abbildung 5.10.: Klassendiagramm der Editor-Applikation²⁰

¹⁹Eigene Darstellung mittels Papyrus.

²⁰Eigene Darstellung mittels Papyrus.

5.7. Prototyp

Nach dem Finden und Ausarbeiten der Vision (5.1.1) sowie der Identifikation der wichtigsten Komponenten (5.2), wurde ein Prototyp von Teilen der geplanten Software umgesetzt. Dies diente auch der Identifikation der zusätzlichen Anforderungen (5.1.4).

Die Entwicklung des Prototypen war ein iterativer Prozess. Es wurden Teil-Ziele definiert, welche dann Etappenweise erarbeitet wurden.

Für den Prototypen wurde nicht explizit ein Domänenmodell festgehalten. Dieses wurde anhand der Vision und der Komponenten erstellt. Überlegungen dazu flossen direkt in das Domänenmodell der Software-Architektur ein (5.3).

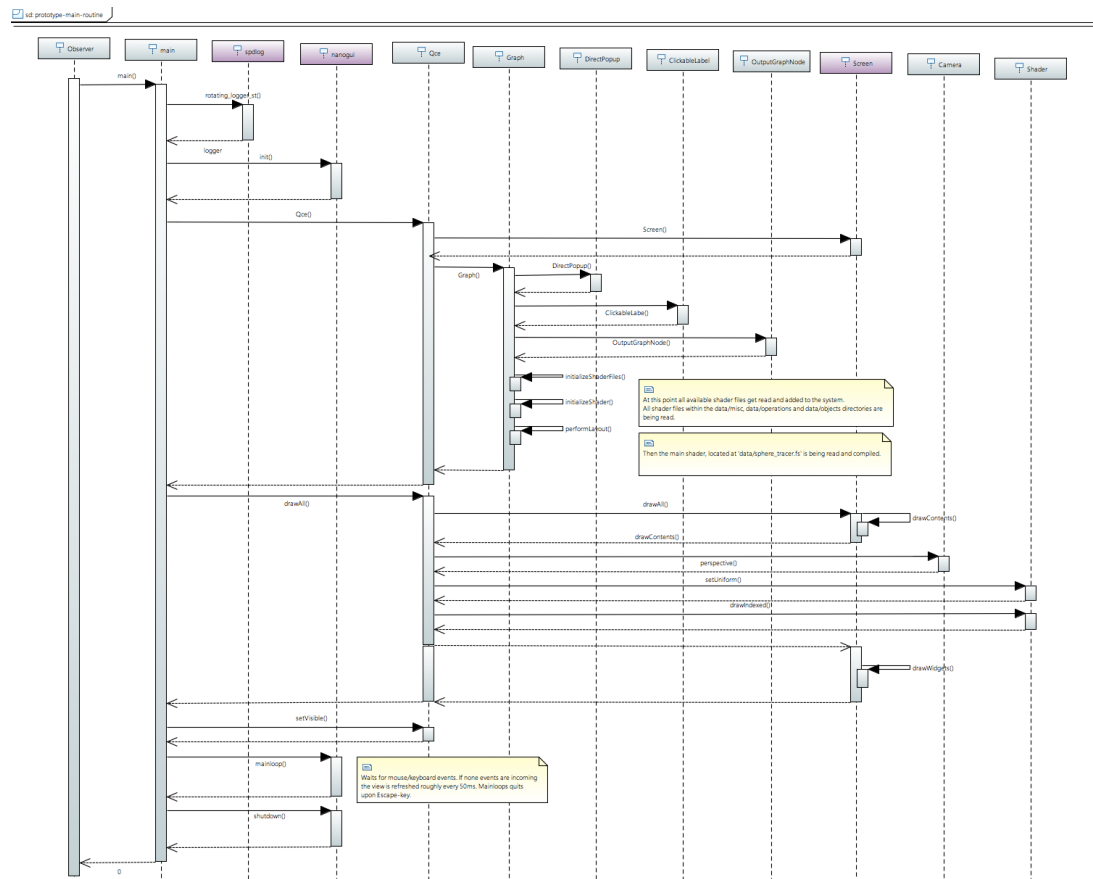


Abbildung 5.11.: Sequenz-Diagramm des Haupt-Ablaufes der Prototyp-Applikation²¹

In Abbildung 5.12 findet sich das Klassendiagramm des Prototypen. Blau-graue Elemente stellen dabei selbst entwickelte Pakete, violette Elemente externe Pakete von Drittpersonen dar. Es wird bewusst nicht das vollständige Klassendiagramm mit allen Elementen abgebildet, da dies nach Meinung des Autors zu unübersichtlich würde. Die gesamte Struktur ist dem Programmcode des Prototypen zu entnehmen, welcher dieser Projektarbeit beiliegt.

²¹Eigene Darstellung mittels Papyrus.

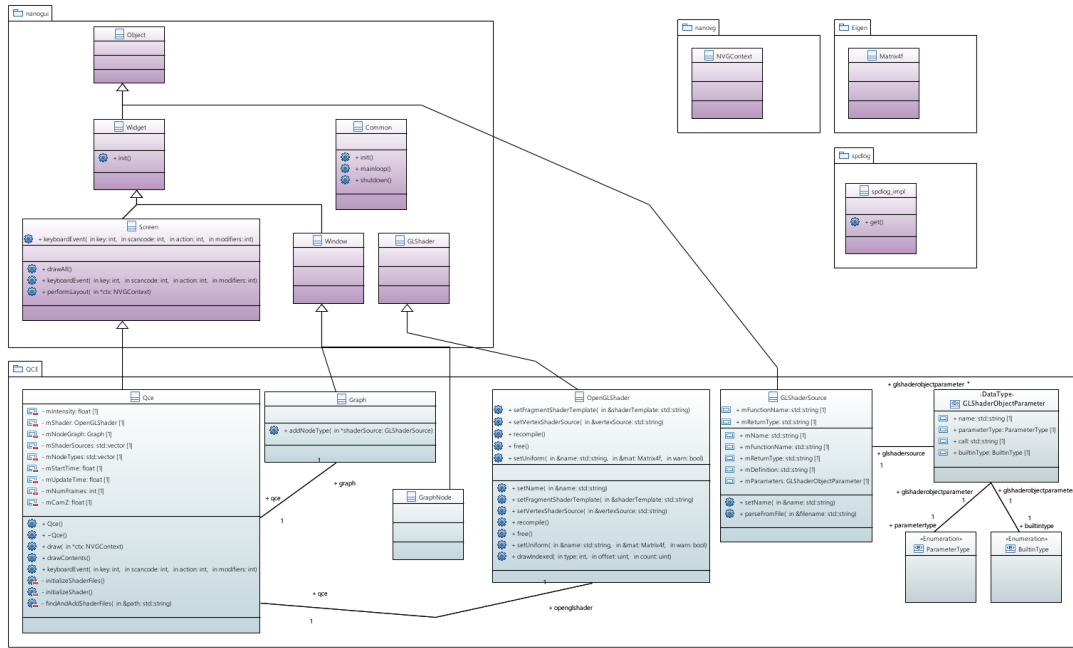


Abbildung 5.12.: Klassendiagramm der Prototyp-Applikation²²

²²Eigene Darstellung mittels Papyrus.

6. Schlusswort

In dieser Projektarbeit wurde Sphere Tracing vorgestellt. Es ist eine optimierte Variante des als Ray Tracing bekannten Verfahrens. Sphere Tracing erlaubt die Darstellung von Szenen in der Qualität von Ray Tracing, aber in Echtzeit.

Zuerst wurden *lokale*, dann *globale Beleuchtungsmodelle* inklusive der *Renderinggleichung* vorgestellt. Weiter wurde das *Ray Tracing* Verfahren ausgehend von der ursprünglichen Idee des Ray Tracings, ein als *Ray Casting* bekanntes Verfahren, vorgestellt. Dabei wurde auch die zugrundeliegende Physik vereinfacht aufgezeigt.

Für die praktische Anwendung der Beleuchtungsmodelle wurden die klassischen Modelle zur Schattierung *Flat-Shading*, *Gouraud-Shading* sowie *Phong-Shading* vorgestellt.

Um die eigentliche Szene darstellen zu können, wurden *Oberflächen* im Allgemeinen eingeführt. Dabei wurde gezeigt, dass zur Modellierung von Oberflächen hauptsächlich zwei Techniken verwendet werden: Die parametrische und die implizite Modellierung bzw. Darstellung. Es wurden dann *implizite Oberflächen* im Speziellen behandelt. Mittels *Distanzfunktionen* wurde eine Grundlage zur Berechnung von Distanzen gezeigt, welche der späteren Modellierung und Darstellung dient. Ausgehend von den Distanzfunktionen wurden schliesslich *Distanzfelder* beschrieben, eine Art Datenstruktur basierend auf den Distanzfunktionen.

Mit *Ray Marching* und *Sphere Tracing* wurden zwei Methoden zur Darstellung von impliziten Oberflächen aufgezeigt. Weiter wurden *Operationen für implizite Oberflächen*, wie beispielsweise die Vereinigung oder Subtraktion, sowie Funktionen zur Modellierung von *geometrischen Primitiven* vorgestellt.

Schliesslich wurde gezeigt, wie die vorgestellten Grundlagen für das Rendering von impliziten Oberflächen genutzt werden können.

Im letzten Abschnitt wurde der umgesetzte *Prototyp* vorgestellt, wobei einzelne Parameter des Prototyps (*Distanz*, *Präzision*, die *Anzahl Schritte* und der *Skalierungsfaktor für Schatten*) anhand einer Beispielszene verglichen wurden.

6.1. Erweiterungsmöglichkeiten

Da diese Projektarbeit den begrenzten Zeitrahmen eines Semesters hat, konnten nicht alle von mir gewünschten Themata bearbeitet werden.

Bei Sphere Tracing handelt es sich um ein relativ junges Verfahren, welches ein grosses Potential für Forschung und Entwicklung enthält. Angesichts des grossen Themengebietes kann man viel Zeit investieren.

Während der Recherche sind einige Thematiken hinzugekommen, sie werden aber nicht bearbeitet. Sie zeigen auf, in welche Richtung die Weiterführung von Projektarbeit und Prototyp gehen kann.

6.1.1. Umgebungs-Verdeckung (Ambient Occlusion)

“Umgebungsverdeckung (englisch Ambient Occlusion, AO) ist eine Shading-Methode, die in der 3D-Computergrafik verwendet wird, um mit relativ kurzer Renderzeit eine realistische Verschattung von Szenen zu erreichen. Das Ergebnis ist zwar nicht physikalisch korrekt, reicht jedoch in seinem Realismus oft aus, um auf rechenintensive globale Beleuchtung verzichten zu können.” [wikipedia_the_free_encyclopedia_umgeb

evans_fast_2006 liefert in seiner Arbeit **evans_fast_2006** interessante Ansätze, wie die Umgebungsverdeckung auf Distanzfelder angewendet werden kann. Dies kann mit relativ wenig Aufwand auch für Sphere Tracing angewendet werden [**evans_fast_2006**].

6.1.2. Kantenglättung

Bei der Darstellung von Szenen mittels Sphere Tracing kommt es an Kanten zu harten Übergängen und Artefakten — es tritt das so genannte “Aliasing” auf.

Hart schlägt in „Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces“ bereits Verfahren zur Kantenglättung vor. Diese verkomplizieren jedoch den Rendering-Prozess und haben Schwächen im Umgang mit angenäherten Obergrenzen von impliziten Oberflächen [10].

lottes_fxaa_2009 schlägt in seiner Arbeit **lottes_fxaa_2009** eine einfache Lösung dieser Problematiken vor. Es handelt sich um einen Bild-Filter, welcher typischerweise in der Nachbearbeitung eines Bildes zum Einsatz kommt. Der Filter wird auf das gerenderte Bild (bzw. auf einem Bild-Puffer) angewendet. Es handelt sich um ein Verfahren, welches auf der Detektion von Kanten basiert [**lottes_fxaa_2009**].

6.1.3. Realistische Materialien

In ?? werden Verfahren zur Berechnung der physikalischen Eigenschaften von Oberflächen und damit von Materialien aufgezeigt. Dabei handelt es sich aber nur um Näherungen unter Annahme von perfekten Bedingungen, was von der Realität weit entfernt ist.

Eine allgemeinere und daher realistischerer Form ist die so genannte BRDF: “Eine bidirektionale Reflektanzverteilungsfunktion (engl. Bidirectional Reflectance Distribution Function, BRDF) stellt eine Funktion für das Reflexionsverhalten von Oberflächen eines Materials unter beliebigen Einfallswinkeln dar. Sie liefert für jeden auf dem Material auftreffenden Lichtstrahl mit gegebenem Eintrittswinkel den Quotienten aus Strahlungsdichte und Bestrahlungsstärke für jeden austretenden Lichtstrahl. BRDFs werden unter anderem in der realistischen 3D-Computergrafik verwendet, wo sie einen Teil der fundamentalen Rendergleichung darstellen und dazu dienen, Oberflächen möglichst realistisch und physikalisch korrekt darzustellen. Eine Verallgemeinerung der BRDF auf Texturen stellt die BTF (Bidirectional Texturing Function) dar.” [**wikipedia_the_free_encyclopedia_bidirektionale_2014**]

burley_physicall-based_2012 liefert mit **burley_physicall-based_2012** eine gute Übersicht, wie solche Verfahren umgesetzt werden können [**burley_physicall-based_2012**]. **bagher_accurate_2012** bieten in **bagher_accurate_2012** eine Fülle an Materialien und Eigenschaften [**bagher_accurate_2012**].

6.1.4. Optimierungsverfahren

Zur Beschleunigung des Sphere Tracing Verfahrens existieren diverse Möglichkeiten. So stellen z.B. **keinert_enhanced_2014** in **keinert_enhanced_2014** diverse Methoden zur Optimierung und Beschleunigung des Renderings sowie Methoden zum Finden von Schnittpunkten vor [**keinert_enhanced_2014**]. **seven_rendering_2012** stellt in **seven_rendering_2012** eine Methode vor, wie Sphere Tracing mittels Aussenden von Kegelförmigen (Primär-) Strahlen optimiert werden kann [**seven_rendering_2012**].

Literatur

- [1] S. Osterwalder, *Volume ray casting - basics & principles*, 14. Feb. 2016.
- [2] Wikipedia Foundation, *Zotero*. Aug. 2015, Published: Website Abgerufen am 27. September 2015.
- [3] 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.
- [4] I. Jacobson, G. Booch und J. Rumbaugh, *The Unified Software Development Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN: 978-0-201-57169-1.
- [5] Wikipedia, the free encyclopedia, *OpenGL*. Okt. 2015, Page Version ID: 146904140.
- [6] Wikipedia Foundation, *Qt (bibliothek)*, in *Wikipedia*, Page Version ID: 156236968, Wikipedia, 17. Juli 2016.
- [7] Wikipedia, the free encyclopedia, *GLFW*. Okt. 2015, Page Version ID: 687716464.
- [8] —, *OPENGL Extension Wrangler Library*. Sep. 2015, Page Version ID: 680716273.
- [9] —, *Boost (C++-Bibliothek)*. Sep. 2015, Page Version ID: 146313866.
- [10] J. C. Hart, „Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces“, *The Visual Computer*, Bd. 12, S. 527–545, 1994.

Abbildungsverzeichnis

4.1. Zeitplan; Der Titel stellt Jahreszahlen, der Untertitel Kalenderwochen dar	6
5.1. Mögliches Aussehen des Editors. ¹	9
5.2. Einzelne Komponenten des Editors ²	27
5.3. Beispiel-Szene innerhalb des Editors ³	28
5.4. Domänenmodell der Player-Applikation ⁴	29
5.5. Domänenmodell der Editor-Applikation ⁵	30
5.6. Sequenz-Diagramm des Use Cases UC1 ⁶	32
5.7. Paket-Diagramm der Player-Applikation ⁷	34
5.8. Paket-Diagramm der Editor-Applikation ⁸	35
5.9. Klassendiagramm der Player-Applikation ⁹	37
5.10. Klassendiagramm der Editor-Applikation ¹⁰	37
5.11. Sequenz-Diagramm des Haupt-Ablaufes der Prototyp-Applikation ¹¹	38
5.12. Klassendiagramm der Prototyp-Applikation ¹²	39

Tabellenverzeichnis

5.1. Erklärung der Begrifflichkeiten der Use Cases, angelehnt an [3, S. 67].	10
5.2. Use Case UC1: Betrachten einer Echtzeit-Animation.	11
5.3. Use Case UC2: Erstellen einer Echtzeit-Animation.	11
5.4. Use Case UC3: Bearbeiten einer bestehenden Animation	12
5.5. Use Case UC4: Exportieren einer Animation.	14
5.6. Use Case UC5: Bereitstellen neuer Editor-Elemente.	16
5.7. Use Case UC6: Erstellen einer neuen Szene.	18
5.8. Use Case UC7: Hinzufügen eines neuen Knotens im Graphen einer Szene.	19
5.9. Use Case UC8: Verbinden eines Knotens im Graphen einer Szene.	20
5.10. Use Case UC9: Hinzufügen eines Schlüsselsbildes eines Parameters eines Knotens.	21
5.11. Use Case UC10: Auswerten und Darstellen eines Knotens.	23
5.12. Mögliche Software/Technologien	25

Auflistungsverzeichnis

4.1. Projekt-Struktur.	8
--------------------------------	---

Anhang

A. Meeting minutes