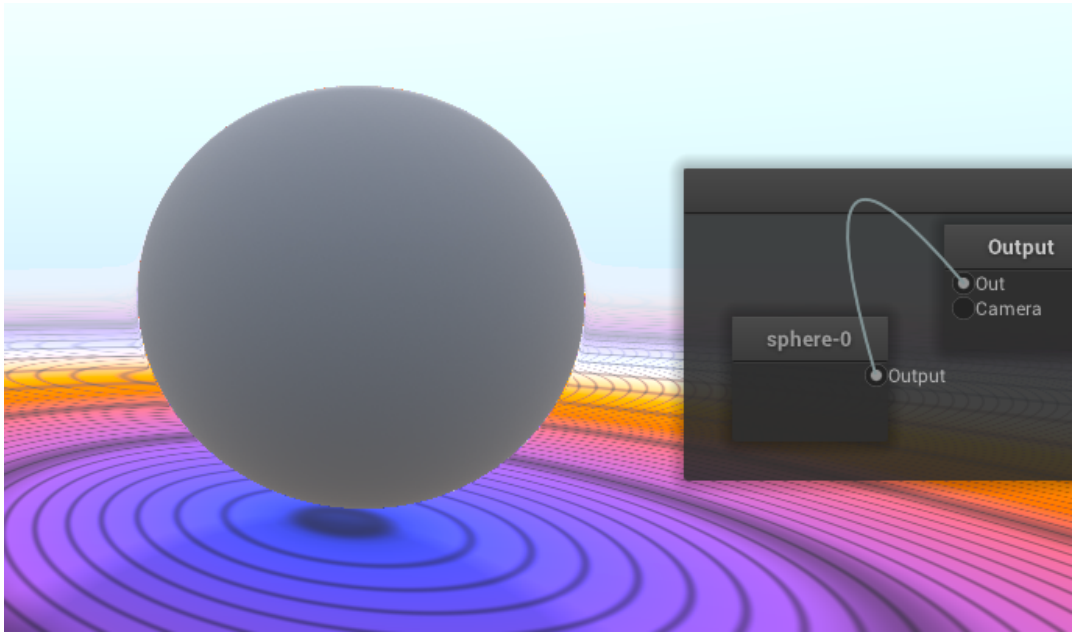


QDE — a visual animation system.

Software-Architektur

MTE7102: Projektarbeit 2

Studiengang:	Informatik
Autor:	Sven Osterwalder[1]
Betreuer:	Prof. Claude Fuhrer[2]
Datum:	05.08.2016
Version:	1.0

[1] sven.osterwalder@students.bfh.ch
[2] claude.fuhrer@bfh.ch

Versionen

Version	Datum	Autor(en)	Änderungen
0.1	29.04.2016	SO	Initiale Erstellung des Dokumentes
0.2	11.05.2016	SO	Erarbeitung diverser Diagramme
0.3	10.06.2016	SO	Erweiterte Gliederung und Struktur
0.4	28.07.2016	SO	Hinzufügen der administrativen Aspekte, des Vorgehens und des Abstracts
0.5	04.08.2016	SO	Software-Architektur, Prototyp und Rendering. Schlusswort, Überarbeitung Abstract.
1.0	04.08.2016	SO	Software-Architektur, Prototyp und Rendering. Schlusswort, Überarbeitung Abstract.
1.0	05.08.2016	SO	Überarbeitung des gesamten Dokumentes hinsichtlich Rechtschreibung sowie Formulierung. Nachführen der Meeting Minutes. Nachführen der Versionshistorie.

Die hier aufgeführten Versionen entsprechen nicht dem Zeitpunkt des Hinzufügens einzelner Elemente, sie stellen vielmehr dar wann die Elemente fertiggestellt wurden.

Abstract

Diese Projektarbeit stellt 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 Sequenzer erlaubt die Animation von Elementen, wie z.B. Modellen oder Bitmaps.

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

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

Inhaltsverzeichnis

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 beziehungsweise 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 Sequenzer erlaubt die Animation von Elementen, wie z.B. Modellen oder Bitmaps.

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

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

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 Osterwalder[1]	
Betreuer	Prof. Claude Fuhrer[2]	*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.
- **Protoyp**
  Der Prototyp entstand im Rahmen der Bearbeitung der Thematik. Er zeigt die Machbarkeit (von Teilen) der angedachten Software-Architektur auf.

[1] sven.osterwalder@students.bfh.ch
[2] claude.fuhrer@bfh.ch

3. Aufgabenstellung

3.1. Motivation

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

Das Verfahren nutzt die parametrische Darstellung impliziter Oberflächen um Objekte (und somit auch Szenen) zu definieren. Eine implizite Oberfläche ist entweder eine Kontur einer Funktion mit Wert 0 oder aber eine Funktion $f(\boldsymbol{x}) = \mathbb{R}^3 \rightarrow \mathbb{R}$ [1, S. 29]. Jedem Punkt einer Menge $\boldsymbol{p} \in \mathbb{R}^3$ wird ein skalarer Wert $s \in \mathbb{R}$ zugewiesen. Dabei besteht die Oberfläche aus der Menge von Punkten $\boldsymbol{x} \equiv (x, y, z) \in \mathbb{R}^3$ [1, S. 29]. Bei Operationen, welche auf implizite Oberflächen angewendet werden, handelt es sich um Transformationen. Dabei wird der Raum, in dem sich eine implizite Oberfläche befindet, invertiert. Bei Transformationen handelt es sich um vom Vorzeichen abhängige Distanzfunktionen (*signed distance functions*) [1, S. 37]. Mehr Details zu impliziten Oberflächen und Operationen finden sich unter [1, S. 29ff] sowie [1, S. 37ff].

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 beziehungsweise 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.

3.1.1. Demoszene

In den 1980er-Jahren entwickelte sich "unter Anhängern der Computerszene ... während der Blütezeit der 8-Bit-Systeme" [2] 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." [2]

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 (siehe Kapitel 5) sowie der Umsetzung eines Prototypen. Dieser bildet einen Teil der Software-Architektur ab.

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.

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. Ansonsten fanden keine vorgängigen Aktivitäten statt.

3.2.2. Neue Lerninhalte

Zusätzlich zu den formalen Lerninhalten hatte die Arbeit für den Autor keine wirklich neuen Lerninhalte, die Themen waren dem Autor bereits aus seinem Bachelor-Studium bekannt. Folgende Lerninhalte konnten jedoch wesentlich vertieft werden:

- Software-Architektur
- Software-Entwurfs-Muster
- UML
- Teilweise praktische Umsetzung der vorgestellten Architektur

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 vier 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
- Erstellung eines Prototypen
- Erstellung der abschliessenden Dokumentation aufgrund der gewonnen Erkenntnisse

Die Phasen *Erstellung eines Prototypen* sowie *Erstellung der abschliessenden Dokumentation* liefen parallel ab. Erkenntnisse einer Phase flossen jeweils in die andere Phase ein.

4.2.2. Zeitplan / Projektphasen

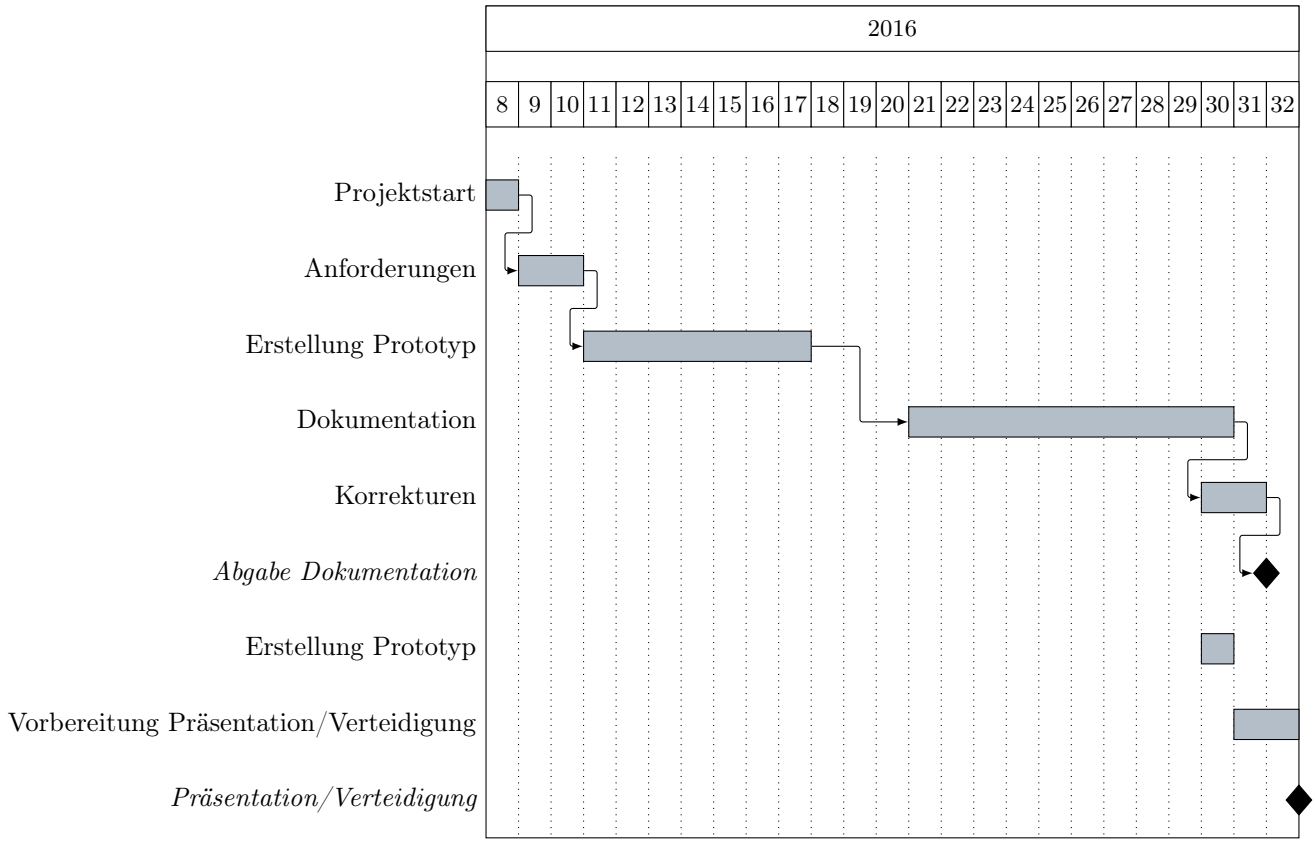


Abbildung 4.1.: Zeitplan; Der Titel stellt Jahreszahlen, der Untertitel Kalenderwochen dar

Anforderungen

In dieser Phase wurde das Ziel dieser Projektarbeit festgelegt. Ausgehend vom Ziel wurden die dazu erforderlichen Projektphasen festgelegt. Bei diesen handelt es sich um die hier beschriebenen Projektphasen.

Erstellung Prototyp

Ausgehend von den Anforderungen und vor allem der Vision, wurde ein Prototyp erstellt um die Machbarkeit aufzuzeigen und um Erkenntnisse zu gewinnen.

Dokumentation

Die vorliegende Arbeit entspricht der Dokumentation. Sie sollte während der gesamten Projektarbeit stetig erweitert werden. Allerdings wurde zu viel Fokus auf den Prototypen gelegt und somit wurde die Dokumentation eher erst spät richtig angegangen. Sie diente zur Reflexion von fertiggestellten Teilen.

4.3. Standards und Richtlinien

4.3.1. Programmcode

Der Programmcode des Prototypen, welcher in C++ geschrieben wurde, folgt den offiziellen Richtlinien für C++ von Google [1].

4.3.2. Diagramme

Alle Diagramme folgen dem UML 2 Standard [2].

4.3.3. 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.

[1] https://google.github.io/styleguide/cppguide.html
[2] http://www.omg.org/spec/UML/

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 zeigt das mögliche Aussehen des Editors und dessen Komponenten. Diese werden in Unterabschnitt 5.2.2 erläutert.

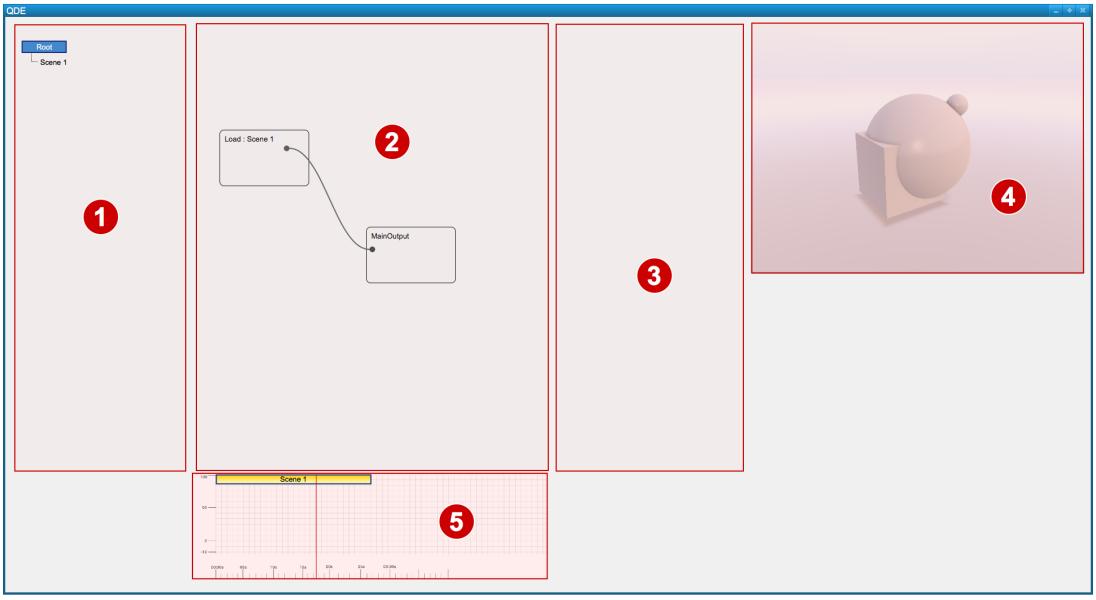


Abbildung 5.1.: Mögliches Aussehen des Editors.

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 vor allem 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.

Tabelle 5.1.: Übersicht der Use Cases.

Use Case 1	Zeigt, wie ein Betrachter mittels Player eine Echtzeit-Animation betrachtet.
Use Case 2	Zeigt, wie ein Anwender mittels Editor eine Echtzeit-Animation erstellt.
Use Case 3	Zeigt, wie ein Anwender mittels Editor eine bestehende Echtzeit-Animation bearbeitet.
Use Case 4	Zeigt, wie ein Anwender mittels Editor eine Echtzeit-Animation für die Verwendung im Player exportiert.
Use Case 5	Zeigt, wie ein Entwickler neue Elemente, wie zum Beispiel Objekte oder Operationen, für den Editor bereitstellen kann.
Use Case 6	Zeigt im Detail, wie ein Anwender mittels Editor eine neue Szene erstellt.
Use Case 7	Zeigt im Detail, wie ein Anwender mittels Editor einen neuen Knoten zu einer bestehenden Szene hinzufügt.
Use Case 8	Zeigt im Detail, wie ein Anwender Knoten einer Szene mittels Graphen des Editors verbindet.
Use Case 9	Zeigt im Detail, wie ein Anwender ein Schlüsselbild für einen Parameter eines Knotens einer Szene des Editors erstellt.
Use Case 10	Zeigt im Detail, wie die Auswertung und Darstellung eines Knotens, sowohl im Editor als auch im Player, geschieht.

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

Tabelle 5.2.: 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 Unterabschnitt 5.1.1).
Stufe (level)	"Ziel des Benutzers" oder "Unterfunktion".
Primärer Aktor	Primärer Aktor des Use Cases: Anwender, Betrachter oder Entwickler. Siehe auch Unterabschnitt 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.
Fehlerfall	Zusätzliche, fehlerhafte Szenarien.
Erweiterungen	Zusätzliche Szenarien.
Zusätzliche Anforderungen	Non-funktionelle Anforderungen, welche in Relation zu dem Use Case stehen.

Use Case UC1: Betrachten einer Echtzeit-Animation

Tabelle 5.3.: Use Case UC1: Betrachten einer Echtzeit-Animation.

Bereich	Player
Stufe (level)	Ziel des Benutzers
Primärer Aktor	Betrachter
Stakeholder und Interessen	Betrachter: Möchte eine zuvor erstellte Echtzeit-Animation ansehen. Anwender: Testen einer erstellen Echtzeit-Animation. Entwickler: Testen einer erstellten Echtzeit-Animation.

Erfolgsszenario	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	—
Fehlerfall	(a) Vorzeitiger Abbruch durch den Betrachter (i) Der Betrachter bricht eine laufende Animation ab. (ii) Der Player wird sofort geschlossen. (b) Absturz/Ausfall des Systems (i) Das System stürzt an einem beliebigen Punkt ab. (ii) Neustart des Players. (iii) Der Player beginnt die Animation von vorne. (c) Auflösungen und Bit-Tiefen können nicht ausgelesen werden (i) Der Player meldet dies per Dialog-Fenster. (ii) Neustart des Players. (iii) Tritt der Fehler erneut auf, wird der Player mit entsprechendem Hinweis geschlossen.
Zusätzliche Anforderungen	—

Use Case UC2: Erstellen einer Echtzeit-Animation

Tabelle 5.4.: Use Case UC2: Erstellen einer Echtzeit-Animation.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Aktor	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.

Fehlerfall	(a) 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. (b) Es sind keine Elemente zum Hinzufügen vorhanden (i) Das Fenster zum Hinzufügen von Elementen ist leer. (c) Die getätigte Arbeit kann nicht gespeichert werden (i) Beim Speichern tritt ein Fehler auf. (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert. (iii) Die Datei wird nicht gespeichert. (iv) Die Datei gilt nach wie vor als geändert. (v) Die Datei muss zu einem späteren Zeitpunkt gespeichert werden. (d) Die erstellte Animation kann nicht exportiert werden (i) Beim Exportieren tritt ein Fehler auf. (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert. (iii) Die Datei wird nicht exportiert. (iv) Die Datei muss zu einem späteren Zeitpunkt exportiert werden.
Zusätzliche Anforderungen	Betreffend dem Exportieren ist das zu verwendende Format zum jetzigen Zeitpunkt noch unklar. In Frage käme zum Beispiel JSON[1] oder X3D[2].

Use Case UC3: Bearbeiten einer bestehenden Animation

Tabelle 5.5.: Use Case UC3: Bearbeiten einer bestehenden Animation

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

[2]http://www.json.org/
[2]http://www.web3d.org/x3d

Stakeholder und Interessen	Anwender: Möchte eine zuvor erstellte Echtzeit-Animation verändern. Betrachter: Möchte eine geänderte Echtzeit-Animation ansehen. Entwickler: Testen einer geänderten Echtzeit-Animation.
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 abzubrechen.

Fehlerfall

(a) 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.

(b) 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.

(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.

(d) Die getätigte Arbeit kann nicht gespeichert werden
  (i) Beim Speichern tritt ein Fehler auf.
  (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.
  (iii) Die Datei wird nicht gespeichert.
  (iv) Die Datei gilt nach wie vor als geändert.
  (v) Die Datei muss zu einem späteren Zeitpunkt gespeichert werden.

(e) Die erstellte Animation kann nicht exportiert werden
  (i) Beim Exportieren tritt ein Fehler auf.
  (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.
  (iii) Die Datei wird nicht exportiert.
  (iv) Die Datei muss zu einem späteren Zeitpunkt exportiert werden.

Zusätzliche Anforderungen	—

Use Case UC4: Exportieren einer Animation

Tabelle 5.6.: Use Case UC4: Exportieren einer Animation.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Aktor	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 benötigen 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 abzubrechen.

Fehlerfall

(a) 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.

(b) 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.

(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.

(d) Die getätigte Arbeit kann nicht gespeichert werden
  (i) Beim Speichern tritt ein Fehler auf.
  (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.
  (iii) Die Datei wird nicht gespeichert.
  (iv) Die Datei gilt nach wie vor als geändert.
  (v) Die Datei muss zu einem späteren Zeitpunkt gespeichert werden.

(e) Die erstellte Animation kann nicht exportiert werden
  (i) Beim Exportieren tritt ein Fehler auf.
  (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.
  (iii) Die Datei wird nicht exportiert.
  (iv) Die Datei muss zu einem späteren Zeitpunkt exportiert werden.

Zusätzliche Anforderungen	—

Use Case UC5: Bereitstellen neuer Editor-Elemente

Tabelle 5.7.: Use Case UC5: Bereitstellen neuer Editor-Elemente.

Bereich	Player
Stufe (level)	Ziel des Benutzers
Primärer Aktor	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.

Fehlerfall	(a) 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. (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. (c) Der Knoten kann nicht persistiert werden (i) Beim Speichern tritt ein Fehler auf. (ii) Der Entwickler wird via Dialog-Fenster über den Fehler informiert. (iii) Die Datei wird nicht gespeichert. (iv) Die Datei gilt nach wie vor als nicht persistiert. (v) Die Datei muss zu einem späteren Zeitpunkt gespeichert werden.
Zusätzliche Anforderungen	—

Use Case UC6: Erstellen einer neuen Szene

Tabelle 5.8.: Use Case UC6: Erstellen einer neuen Szene.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Aktor	Anwender
Stakeholder und Interessen	Anwender: Möchte eine neue Szene einer Echtzeit-Animation erstellen. Betrachter: Möchte eine erstellte Szene ansehen.
Vorbedingungen	• Die Editor-Applikation ist gestartet.

Erfolgsszenario

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 neue 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

(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.

Fehlerfall	1. 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. 2. Die Root-Szene ist nicht in der Bibliothek vorhanden (i) Die Root-Szene ist nicht in der Bibliothek vorhanden. (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert. (iii) Der Editor wird geschlossen. 3. Es erscheint kein Kontext-Menü (i) Der Anwender wird via Dialog-Fenster über den Fehler informiert. (ii) Der Editor wird geschlossen. 4. Es kann keine neue Szene erstellt werden (i) Beim Erstellen der neuen Szene tritt ein Fehler auf. (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.
Zusätzliche Anforderungen	—

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

Tabelle 5.9.: Use Case UC7: Hinzufügen eines neuen Knotens im Graphen einer Szene.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Aktor	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	• Die Editor-Applikation ist gestartet. • Es sind Knoten-Typen zum Hinzufügen vorhanden. • Eine beliebige Szene ist angewählt.

Erfolgsszenario

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 abzubrechen.

Fehlerfall

(a) 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.

(b) Es sind keine Knotentypen zum Hinzufügen vorhanden
   (i) Das Fenster zum Hinzufügen von Knoten-Typen ist leer.

(c) Es kann kein neuer Knoten erstellt werden
   (i) Beim Erstellen des neuen Knotens tritt ein Fehler auf.
   (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.

Zusätzliche Anforderungen	—

Use Case UC8: Verbinden eines Knotens im Graphen einer Szene

Tabelle 5.10.: Use Case UC8: Verbinden eines Knotens im Graphen einer Szene.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Aktor	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	• Die Editor-Applikation ist gestartet. • Es sind Knoten-Typen zum Hinzufügen vorhanden. • Eine beliebige Szene ist angewählt. • 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 abzubrechen.
Fehlerfall	(a) 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. (b) Es kann keine neue Verbindungslinie erstellt werden (i) Beim Erstellen der neuen Verbindungslinie tritt ein Fehler auf. (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert. (c) Es sind keine Ein- beziehungsweise Ausgangs-Buchsen vorhanden (i) Beim Laden eines Knotens tritt ein Fehler auf. (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.
Zusätzliche Anforderungen	—

Use Case UC9: Hinzufügen eines Schlüsselbildes eines Parameters

Tabelle 5.11.: Use Case UC9: Hinzufügen eines Schlüsselsbildes eines Parameters eines Knotens.

Bereich	Editor
Stufe (level)	Ziel des Benutzers
Primärer Aktor	Anwender
Stakeholder und Interessen	Anwender: Möchte einen Parameter eines Knotens einer Szene animieren. Betrachter: Möchte animierten Inhalt sehen.

Vorbedingungen

- Die Editor-Applikation ist gestartet.
- Es sind Knoten-Typen zum Hinzufügen vorhanden.
- Eine beliebige Szene ist angewählt.
- Die Szene verfügt bereits über Knoten.

Erfolgsszenario

1. Der Anwender klickt mit der linken Maustaste auf den Knoten, wessen 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 Schlüsselsbildes.
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 des der Szene hinzu.
8. Der Editor sendet via Szenegraph das Signal, dass eine 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 abzubrechen.

Fehlerfall	(a) 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. (b) Es kann kein neues Schlüsselbild erzeugt werden (i) Beim Erstellen des neuen Schlüsselbildes tritt ein Fehler auf. (ii) Der Anwender wird via Dialog-Fenster über den Fehler informiert.
Zusätzliche Anforderungen	—

Use Case UC10: Auswerten und Darstellen eines Knotens

Tabelle 5.12.: Use Case UC10: Auswerten und Darstellen eines Knotens.

Bereich	Editor und Player
Stufe (level)	Ziel des Benutzers und Unterfunktion
Primärer Aktor	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	• Die Editor-Applikation ist gestartet. • Es sind Knoten-Typen zum Hinzufügen vorhanden. • Eine beliebige Szene ist angewählt. • 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 —

Fehlerfall Editor

(a) 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.

(b) Rendering ist nicht möglich
  (i) Der Anwender wird via Dialog-Fenster über den Fehler informiert.

(c) Ein Knoten kann nicht evaluiert werden
  (i) Der Anwender wird via Dialog-Fenster über den Fehler informiert.

Fehlerfall Player	(a) Absturz/Ausfall des Systems (i) Das System stürzt an einem beliebigen Punkt ab. (ii) Neustart des Players. (iii) Der Player beginnt die Animation von vorne — sofern eine solche vorhanden ist. (b) Das Demoskript kann nicht gefunden werden (i) Der Betrachter wird via Dialog-Fenster über den Fehler informiert. (ii) Der Player wird geschlossen. (c) Das Demoskript kann nicht evaluiert werden oder ist fehlerhaft (i) Der Betrachter wird via Dialog-Fenster über den Fehler informiert. (ii) Der Player wird geschlossen. (d) Rendering ist nicht möglich (i) Der Betrachter wird via Dialog-Fenster über den Fehler informiert. (ii) Der Player wird geschlossen. (e) Ein oder mehrere Knoten können nicht evaluiert werden (i) Der Betrachter wird via Dialog-Fenster über den Fehler informiert. (ii) Der Player wird geschlossen.
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 5.13 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 5.13.: Mögliche Software/Technologien

Komponente	Beschreibung	Verweise
C++	Objektorientierte Programmiersprache	[3]
OpenGL	Plattformunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafikanwendungen [5]	[4]

[3] `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`
[4] `https://www.opengl.org/registry/doc/glspec45.core.pdf`

Qt	"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].	[5]
GLFW	OpenGL-Bibliothek, welche die Erstellung und Verwaltung von Fenstern sowie OpenGL-Kontexte vereinfacht [7].	[6]
GLEW	OpenGL Extension Wrangler. Bibliothek zum Abfragen und Laden von OpenGL-Erweiterungen (Extensions) [8].	[7]
GLM	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.	[8]
CMake	Software zur Verwaltung von Build-Prozessen von Software	[9]
LLVM	Ansammlung von modularen und wiederverwendbaren Compilern und Toolchains.	[10]
Clang	Compiler-Frontend für LLVM.	[11]
Boost	Freie Bibliothek bestehend aus einer Vielzahl von Bibliotheken, die den unterschiedlichsten Aufgaben von Algorithmen auf Graphen über Metaprogrammierung bis hin zu Speicherverwaltung dienen [9].	[12]

5.2. Komponenten

Ausgehend von den Anforderungen (siehe Abschnitt 5.1) können einzelne Komponenten der Applikation abgeleitet werden. Einzelne Teile davon wurden schon durch die Vision (siehe Unterabschnitt 5.1.1) 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.

[5] `https://www.qt.io`
[6] `http://www.glfw.org`
[7] `http://glew.sourceforge.net`
[8] `https://glm.g-truc.net`
[9] `https://www.cmake.org`
[10] `http://llvm.org`
[11] `http://clang.llvm.org`
[12] `http://www.boost.org`

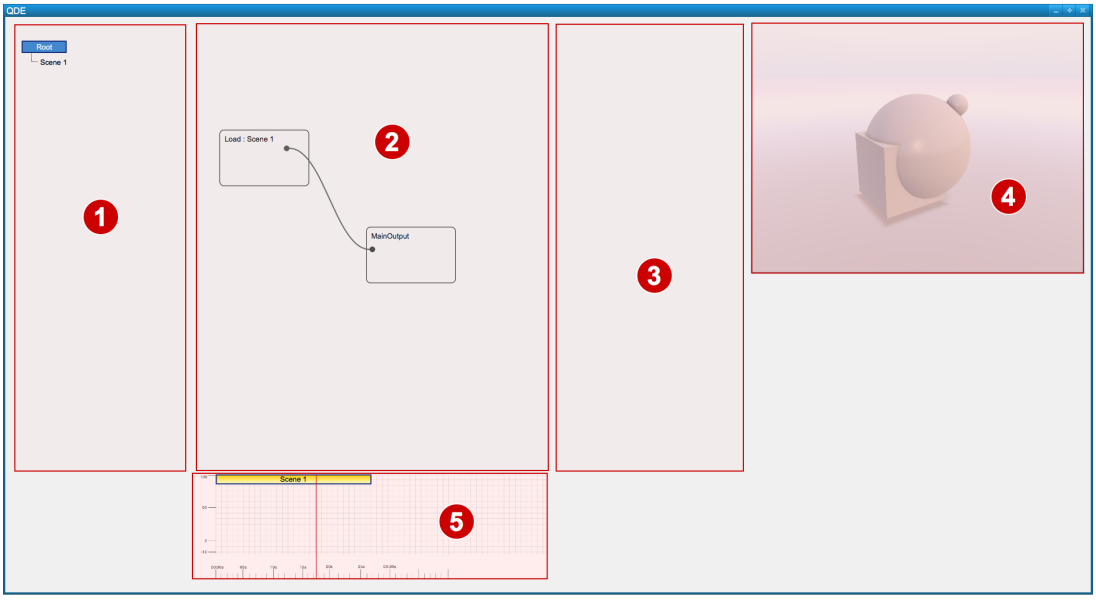


Abbildung 5.2.: Einzelne Komponenten des Editors

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

Bibliothek

Das Element 1 in Abbildung 5.2 zeigt die (Szenen-) Bibliothek. Diese beinhaltet alle Szenen einer Echtzeit-Animation. Eine Szene ist dabei eine Menge von Knoten und Kanten. 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, welcher schliesslich zum Abspielen evaluiert wird, und kann nicht gelöscht werden. Wird eine Szene mit der Maus angewählt, so wird deren Inhalt im Graphen dargestellt.

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 dargestellt, andererseits werden dessen Eigenschaften im Parameter-Fenster angezeigt.

Folgende Typen von Knoten sind geplant:

- **Scene**
  Eine Menge von Knoten und Kanten.
- **TimelineClip**
  Eine Szene, welche in der Zeitachse platziert werden kann.
- **Model**
  Ein Objekt, wie zum Beispiel ein Würfel oder eine Kugel.
- **Camera**
  Eine Kamera, welche die Sicht auf eine Szene erlaubt.
- **Light**
  Ein Licht, welches in einer Szene platziert wird und welches für Beleuchtung dieser sorgt.

- **Material**
  Definiert Material-Eigenschaften eines Objektes, zum Beispiel ob dessen Oberfläche matt oder reflektierend ist.
- **Operator**
  Bietet eine Operation wie zum Beispiel die Gruppierung von zwei Modellen oder die Verdrehung eines Modelles.
- **Effect**
  Ein Nachbearbeitungs-Effekt (postprocessing-effect) wie zum Beispiel Bewegungsunschärfe oder Bloom.

Parameter

Das Element 3 in Abbildung 5.2 zeigt die Parameter des aktuell gewählten Knoten im Graphen. Neben jedem Parameter befindet sich eine Schaltfläche zum Setzen von Schlüsselbildern (Keyframes) in der Zeitachse (Timeline). 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 5 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 ein Knoten mit animierten Parametern 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.

Abbildung 5.3 zeigt ein Beispiel, wie eine typische Szene mit animierten Parametern aussehen könnte. Der Übersicht halber werden nur der Graph, die Parameter sowie die Zeitachse dargestellt.

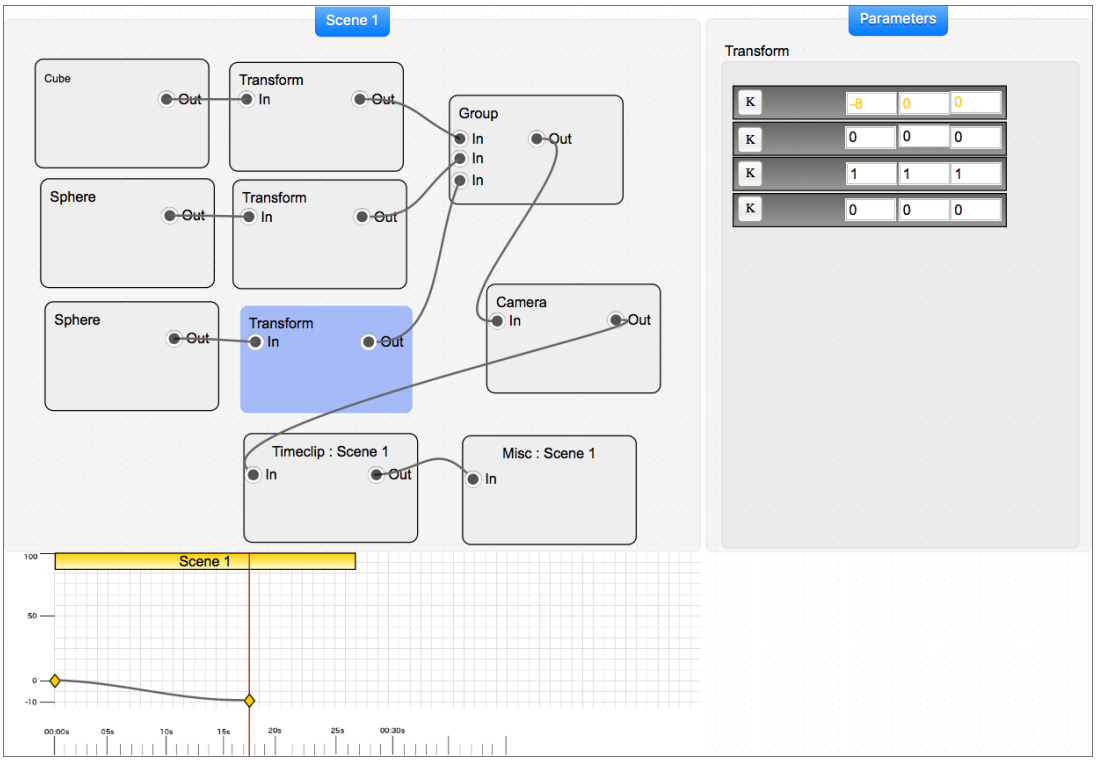


Abbildung 5.3.: Graph, Parameter und Zeitachse einer Beispiel-Szene innerhalb des Editors.

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 bring [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.

5.3.1. Player

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 exportierte 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. DemoData dient der Verwaltung von DemoScript, also zum Export von diesem.

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 (siehe Unterabschnitt 5.3.2) sowie im Klassendiagramm des Prototypen (siehe Kapitel 6).

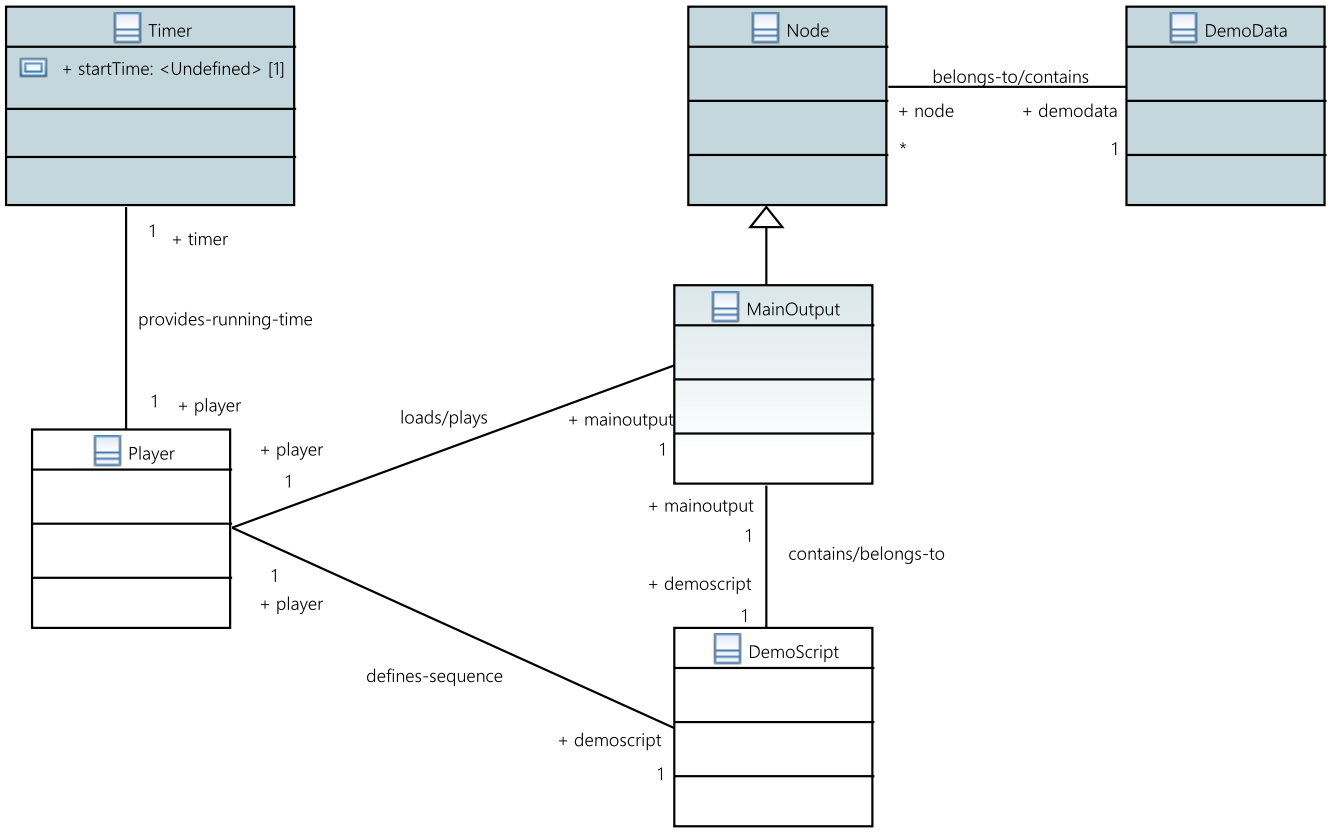


Abbildung 5.4.: Domänenmodell der Player-Applikation

5.3.2. Editor

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.

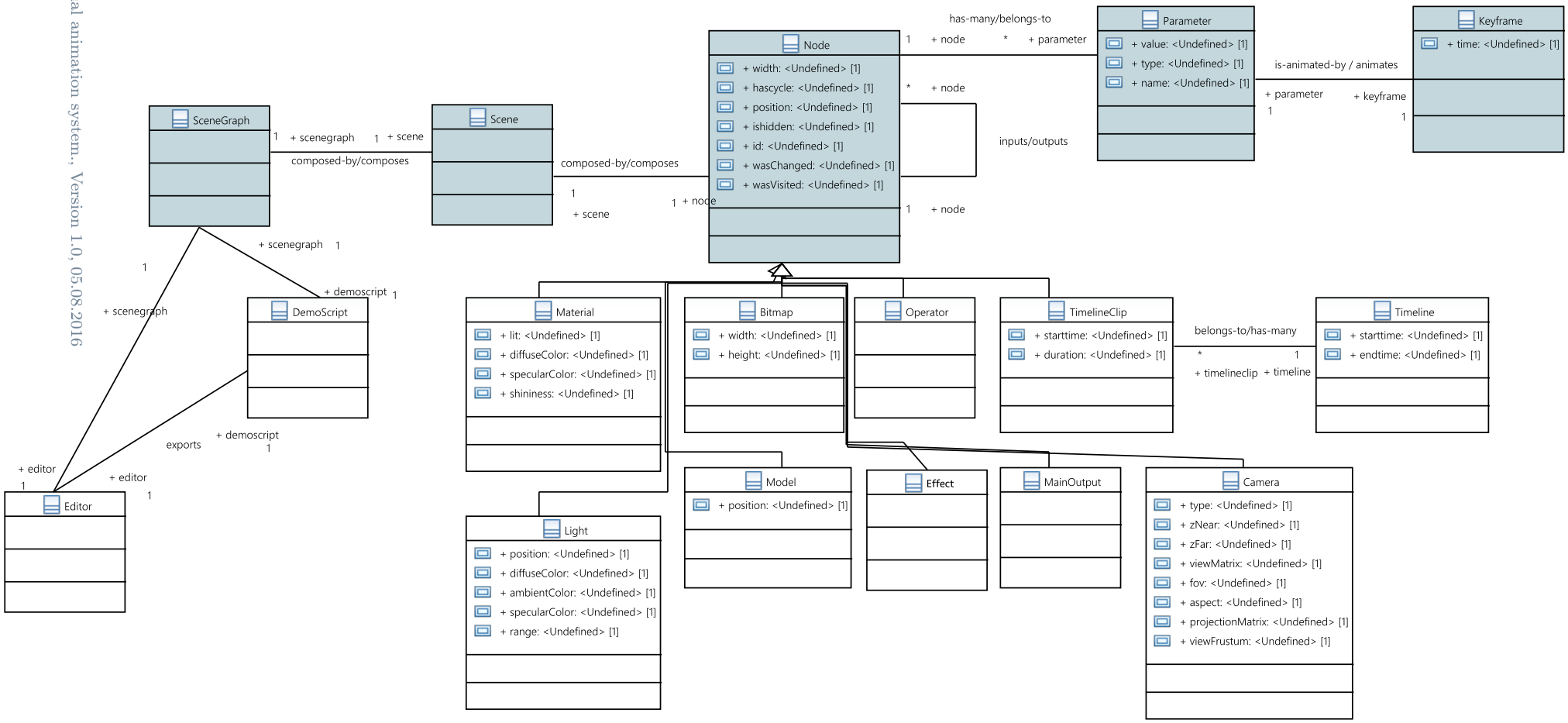
Die Komponenten der grafischen Benutzeroberfläche wurden in diesem Modell bewusst weggelassen, da dies das Modell nur unnötig vergrössern würde. Die eigentliche Logik findet sich in den essentiellen Konzepten beziehungsweise in den hier gezeigten Objekten. Die grafische Benutzeroberfläche bildet diese 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 (siehe Kapitel 6) 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.


SceneGraph
+ scenegraph
composed-by/composes
Scene
composed-by/composes
+ scene
Node
+ width: <Undefined> [1]
+ hascycle: <Undefined> [1]
+ position: <Undefined> [1]
+ ishidden: <Undefined> [1]
+ id: <Undefined> [1]
+ wasChanged: <Undefined> [1]
+ wasVisited: <Undefined> [1]
+ node
has-many/belongs-to
+ parameter
Parameter
+ value: <Undefined> [1]
+ type: <Undefined> [1]
+ name: <Undefined> [1]
is-animated-by / animates
+ parameter
+ keyframe
Keyframe
+ time: <Undefined> [1]
inputs/outputs
+ scenegraph
+ democript
DemoScript
+ editor
Editor
exports
Material
+ lit: <Undefined> [1]
+ diffuseColor: <Undefined> [1]
+ specularColor: <Undefined> [1]
+ shininess: <Undefined> [1]
Bitmap
+ width: <Undefined> [1]
+ height: <Undefined> [1]
Operator
TimelineClip
+ starttime: <Undefined> [1]
+ duration: <Undefined> [1]
belongs-to/has-many
+ timelineclip
+ timeline
Timeline
+ starttime: <Undefined> [1]
+ endtime: <Undefined> [1]
Light
+ position: <Undefined> [1]
+ diffuseColor: <Undefined> [1]
+ ambientColor: <Undefined> [1]
+ specularColor: <Undefined> [1]
+ range: <Undefined> [1]
Model
+ position: <Undefined> [1]
Effect
MainOutput
Camera
+ type: <Undefined> [1]
+ zNear: <Undefined> [1]
+ zFar: <Undefined> [1]
+ viewMatrix: <Undefined> [1]
+ fov: <Undefined> [1]
+ aspect: <Undefined> [1]
+ projectionMatrix: <Undefined> [1]
+ viewFrustum: <Undefined> [1]

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.

Das Sequenz-Diagramm in Abbildung 5.6 stellt das Erfolgsszenario dar. Die Erweiterungen werden bewusst nicht dargestellt.
Nachfolgend wird der Ablauf des Erfolgsszenarios beschrieben.

1. Der Betrachter startet die Player-Applikation. Voraussetzung ist hierbei ein gültiges Demoskript mitsamt den benötigen Ressourcen der exportieren Echtzeit-Animation. Idealerweise beinhaltet ein Export nur zwei Dateien: Die Player-Applikation sowie eine binär beziehungsweise eine komprimierte Datei mit dem Demoskript und allen benötigten Ressourcen.
2. 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.
3. 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.
4. Der Player startet darauf den Timer um die Zeit messen zu können. Darauffolgend wird die Haupt-Schleife 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.
5. 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 berechnet Bild zum aktuellen Zeitpunkt mittels Renderer dargestellt.
6. Bricht der Betrachter die Haupt-Schleife mittels Taste ab oder ist keine gültige Sequenz mehr vorhanden, beendet sich der Player.

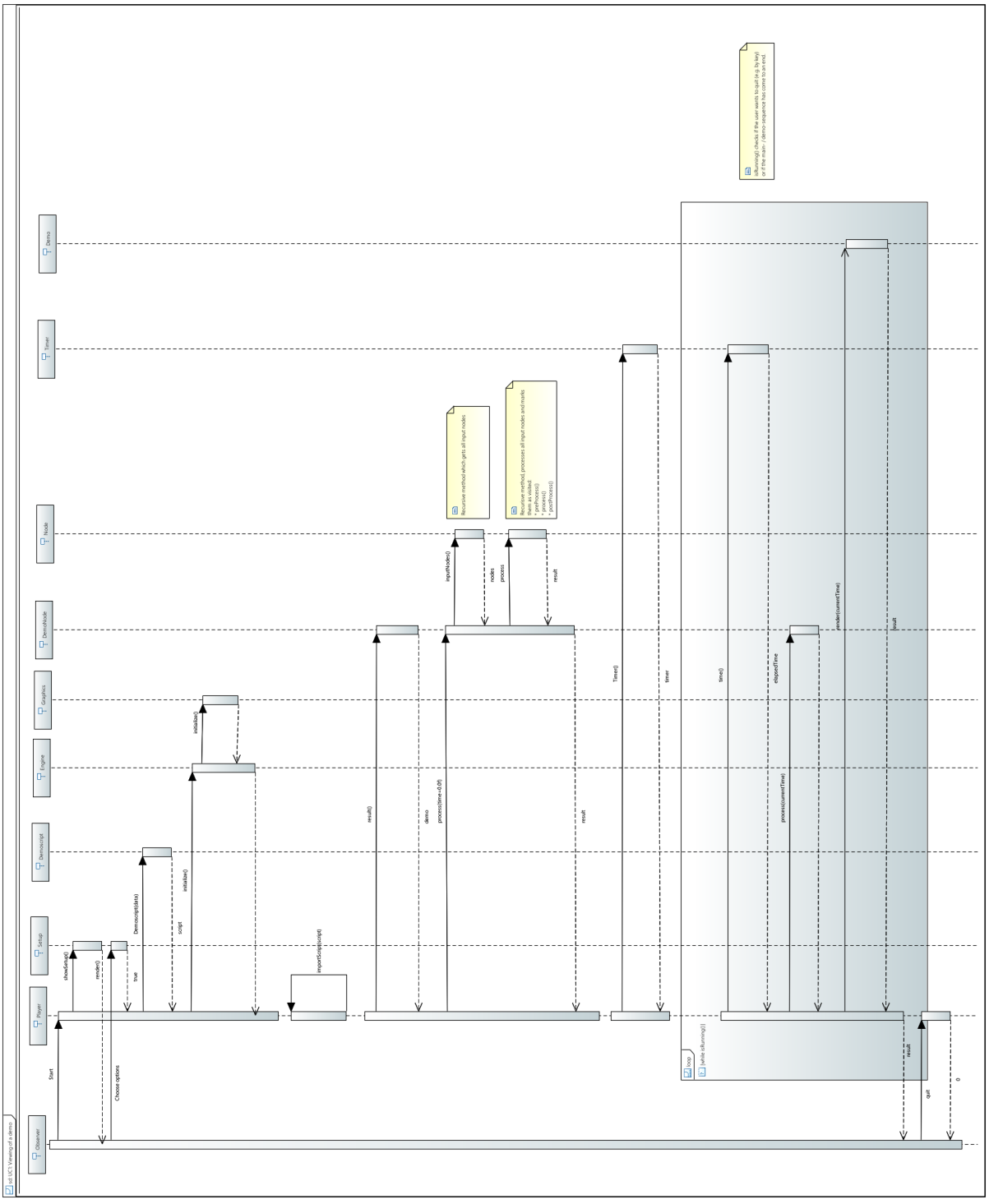


Abbildung 5.6.: Sequenz-Diagramm des Use Cases UC1

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 die verschiedenen Schichten als Übersicht. Abbildung 5.8 zeigt das Paket-Diagramm des Players, Abbildung 5.9 dies des Editors. Blau-graue Elemente stellen dabei selbst entwickelte Pakete, violette Elemente externe Pakete von Drittpersonen dar.

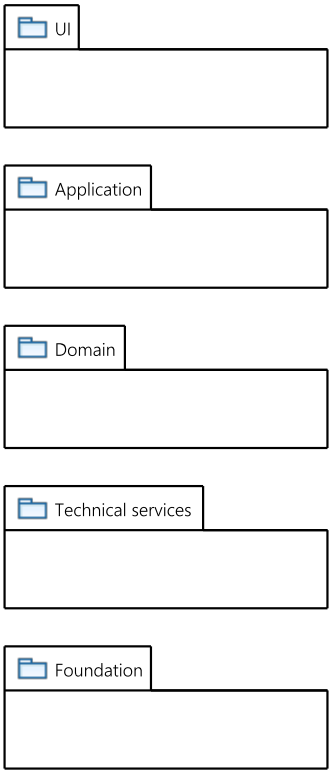


Abbildung 5.7.: Paket-Diagramm der Player-Applikation

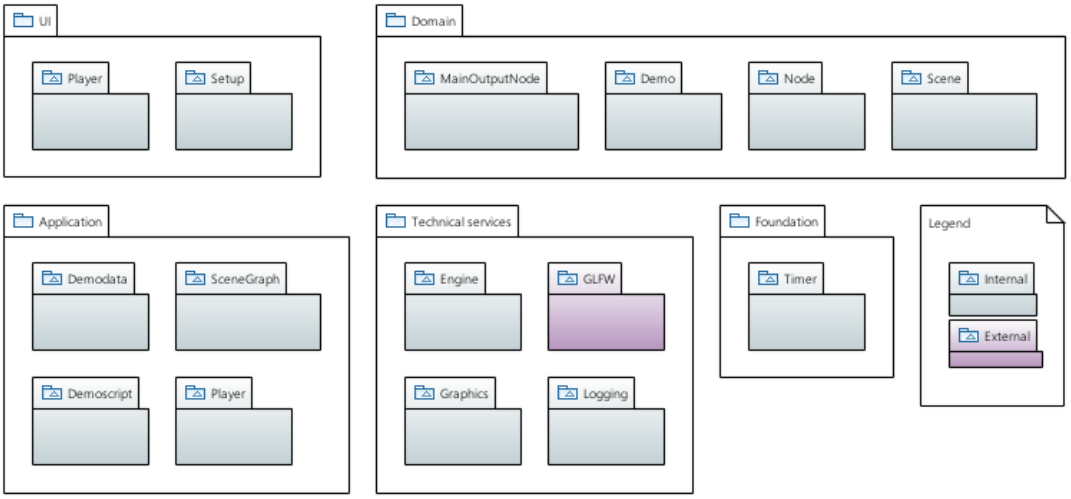


Abbildung 5.8.: Paket-Diagramm der Player-Applikation

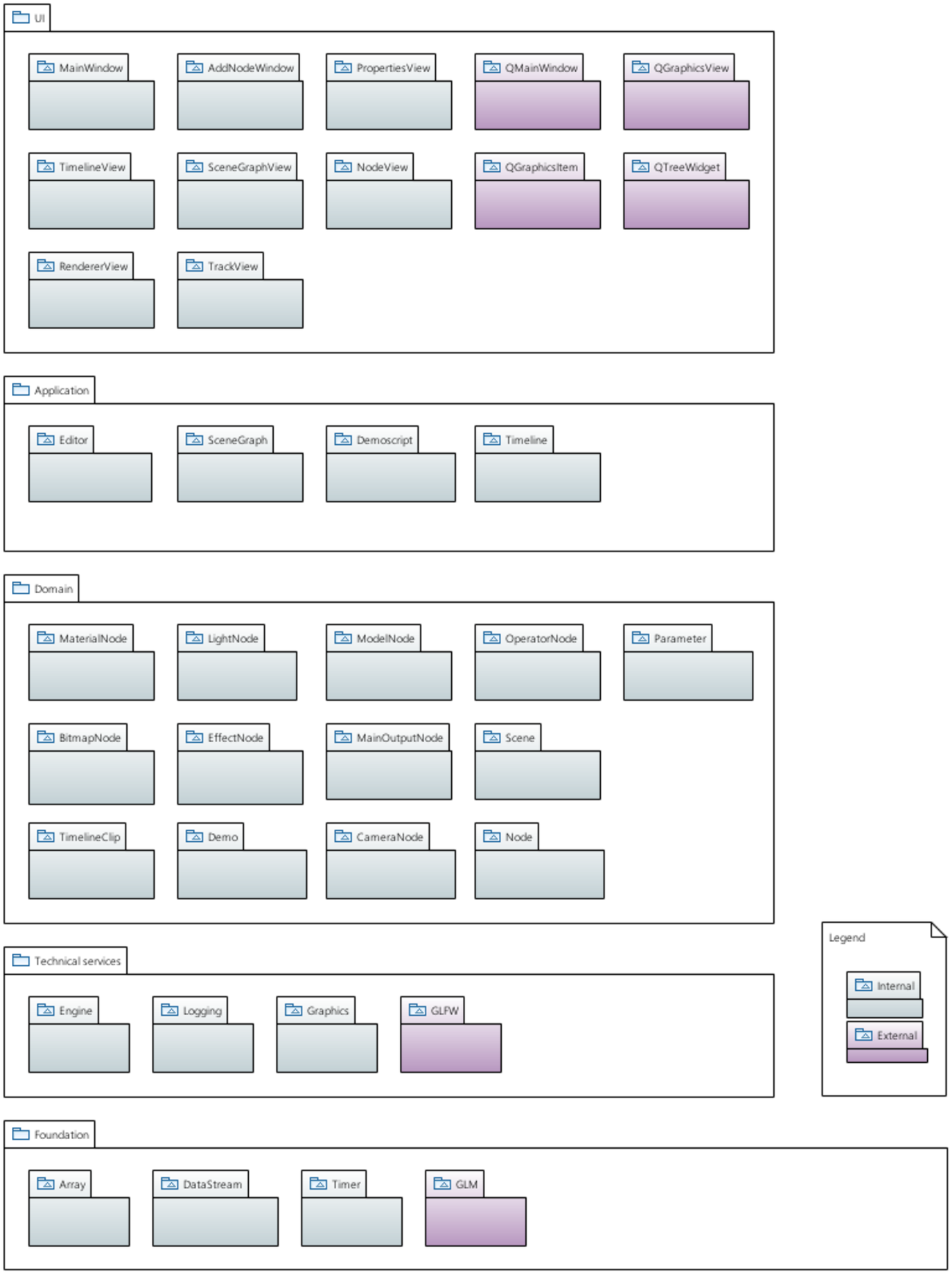


Abbildung 5.9.: Paket-Diagramm der Editor-Applikation

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 [10, S. 217]. 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 [10, S. 217].

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 (siehe Kapitel 6). 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 5.10 zeigt das Klassendiagramm des Players, Abbildung 5.11 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.

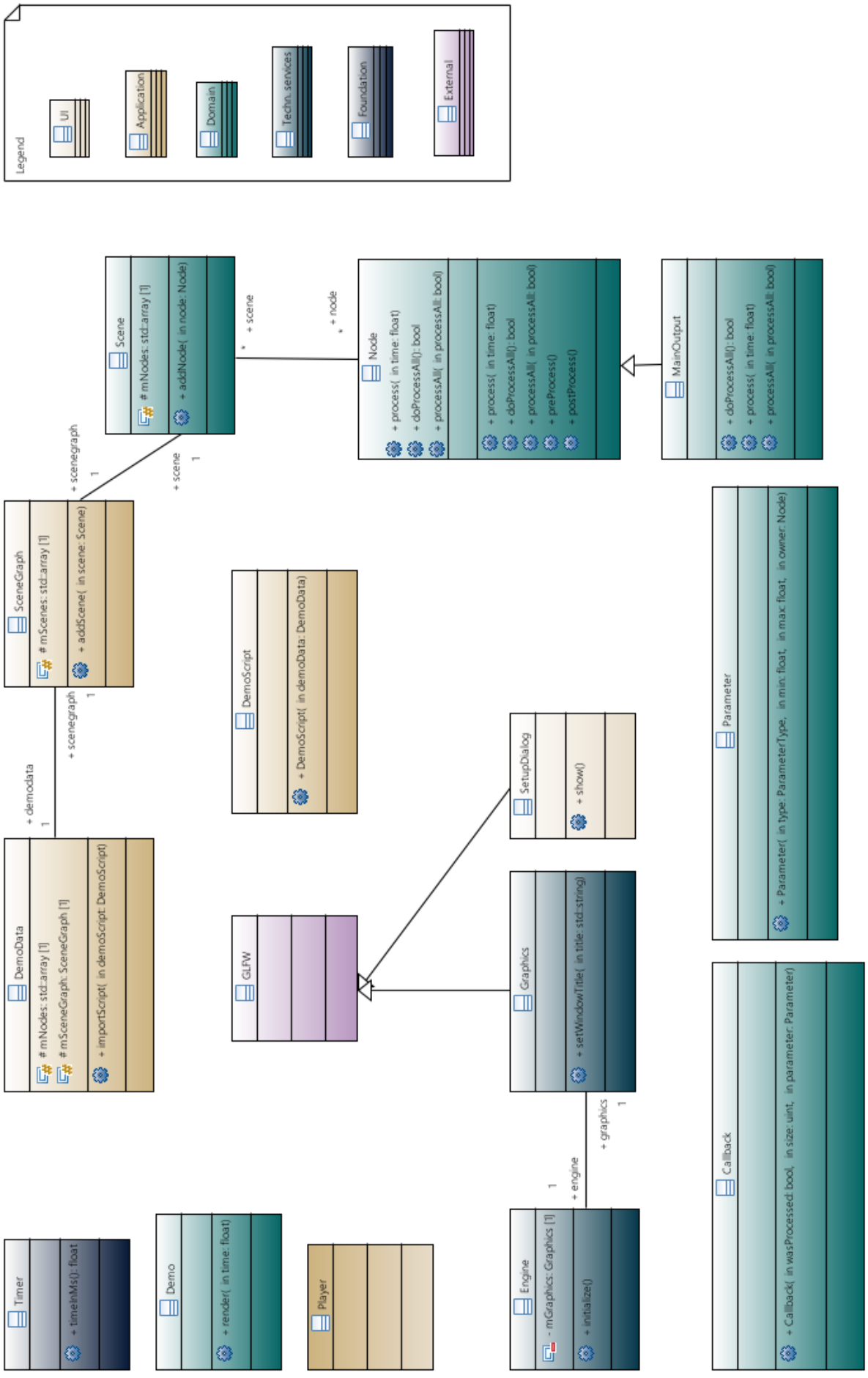


Abbildung 5.10.: Klassendiagramm der Player-Applikation

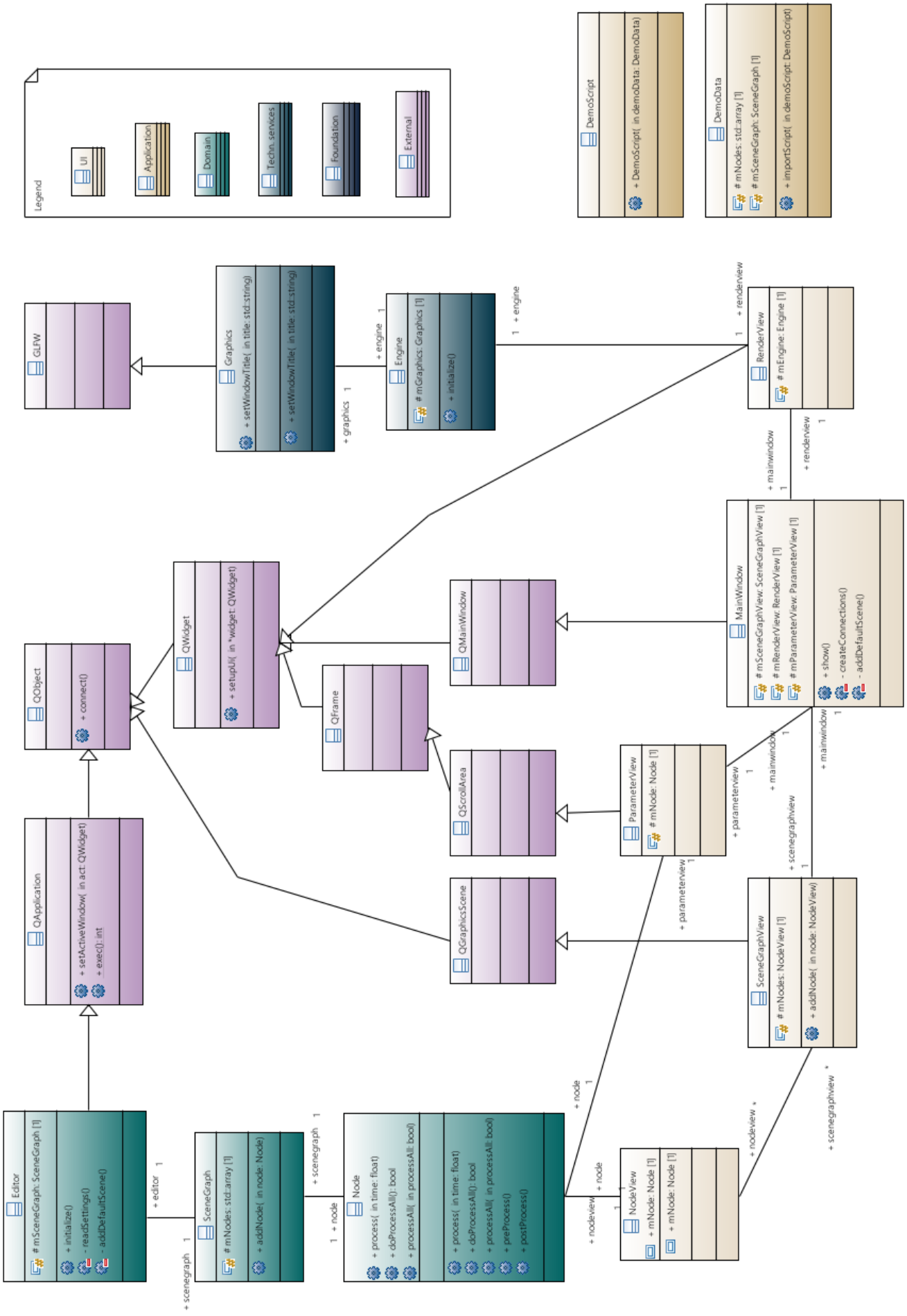


Abbildung 5.11.: Klassendiagramm der Editor-Applikation

6. Prototyp

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

Der Prototyp erlaubt die Modellierung einer einfachen Szene, bestehend aus Primitiven, anhand der Graph-Komponente. Die Primitiven befinden sich in externen (Shader-) Dateien, welche zur Laufzeit zusammen mit einem Haupt-Shader geladen und zur Verfügung gestellt werden. Die nachfolgende Auflistung zeigt die möglichen Unterverzeichnisse und Datei-Typen.

- **data/objects/*.fs.xml**
  Sämtliche Objekt-Definitionen, wie zum Beispiel Kugel oder Würfel.
- **data/operations/*.fs.xml**
  Sämtliche Operationen, wie zum Beispiel die Vereinigung von Objekten.
- **data/misc/*.fs.xml**
  Diverse andere Definitionen, wie zum Beispiel Kameras.
- **data/sphere_tracer.fs**
  Die Haupt-Shader-Datei, welche um Objekte, Operationen oder andere Definitionen ergänzt wird.

Durch die Modellierung wird der Haupt-Shader um Teile ergänzt und neu kompiliert, dies geschieht alles zur Laufzeit in Echtzeit. Die Darstellung findet schliesslich mittels dem als Sphere-Tracing bekannten Ray-Tracing-Verfahren statt. Auflistung 6.1 und Auflistung 6.2 zeigen Beispiel von Shader-Defintionen.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<function>
    <name>sphere</name>
    <return-type>float</return-type>
    <parameters>
        <parameter>
            <name>position</name>
            <builtin>vec3</builtin>
            <type>property</type>
            <call>position - {}</call>
        </parameter>
        <parameter>
            <name>radius</name>
            <builtin>float</builtin>
            <type>property</type>
            <call>{}</call>
        </parameter>
    </parameters>
    <source>
        // Returns the signed distance to a sphere with given radius for the
        // given position.
        float sphere(vec3 position, float radius)
        {
            return length(position) - radius;
        }
    </source>
</function>
```

Auflistung 6.1: Objekt-Definition einer Kugel in XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<function>
    <name>opUnion</name>
    <return-type>float</return-type>
    <parameters>
        <parameter>
            <name>a</name>
            <builtin>float</builtin>
            <type>input</type>
            <call>{}</call>
        </parameter>
        <parameter>
            <name>b</name>
            <builtin>float</builtin>
            <type>input</type>
            <call>{}</call>
        </parameter>
    </parameters>
    <source>
        // Returns the signed distance for a merge of given signed
        // distance a and signed distance b.
        float opUnion(float a, float b)
        {
            return min(a, b);
        }
    </source>
</function>
```

Auflistung 6.2: Definition des Vereinigungs-Operators in XML.

6.1. Vorgehen

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

Das erste Teil-Ziel war das Neu-Kompilieren von Shadern während der Laufzeit, so dass Shader während der Laufzeit erweitert, neu geladen und dargestellt werden können.

Als zweites Teil-Ziel wurde dynamisches Laden von Shader-Dateien ausgehend vom Verzeichnis des Prototypen implementiert.

Als drittes Teil-Ziel wurden Shader in Form von Templates umgesetzt. Mit "Jinja2CppLight" [1] wurde schliesslich ein Template-System eingeführt, welches es erlaubt Shader als Templates zu parsen und Sektionen entsprechend mit Sub-Templates respektive Teil-Shadern zu ersetzen. Dies ermöglicht es Teile von Shadern während der Laufzeit anzupassen. Durch die Erkenntnisse des ersten Teil-Zieles konnte der aus dem Template zusammen mit den Sub-Templates generierte Shader zur Laufzeit geladen und ausgeführt werden.

Als viertes und letztes Teil-Ziel wurde ein Prototyp der Graph-Komponente (siehe Abschnitt 5.2.2) umgesetzt. Der Graph bietet ein Kontext-Menü zum Hinzufügen und Entfernen von Knoten. Für jeden eingelesenen Teil-Shader wird ein Eintrag im Kontext-Menü hinzugefügt, so dass dieser schliesslich dem Graphen hinzugefügt werden kann. Der Graph verfügt standardmässig immer über einen (Haupt-) Ausgangs-Knoten.

[1] https://github.com/hughperkins/Jinja2CppLight

Jeder der Knoten verfügt entweder über mindestens einen Eingang oder über einen Ausgang. Zusätzliche Ein- beziehungsweise Ausgänge werden je nach Typ eines Knotens automatisch, dynamisch erstellt. Die Eingänge des Haupt-Knotens sind jedoch statisch. Dieser verfügt über eine Haupt-Schnittstelle sowie eine Schnittstelle für einen Kamera-Knoten.

Bei jeder neuen Verbindung beziehungsweise bei jedem Trennen einer Verbindung innerhalb des Graphen, wird dieser neu evaluiert und die (Shader-) Ausgabe neu berechnet.

Das Rendering ist schliesslich die Berechnung der relevanten Matrizen, binden des Shaders und setzen der benötigten Uniform-Variablen. Abbildung 6.1 zeigt den aktuellen Stand des Prototypen.

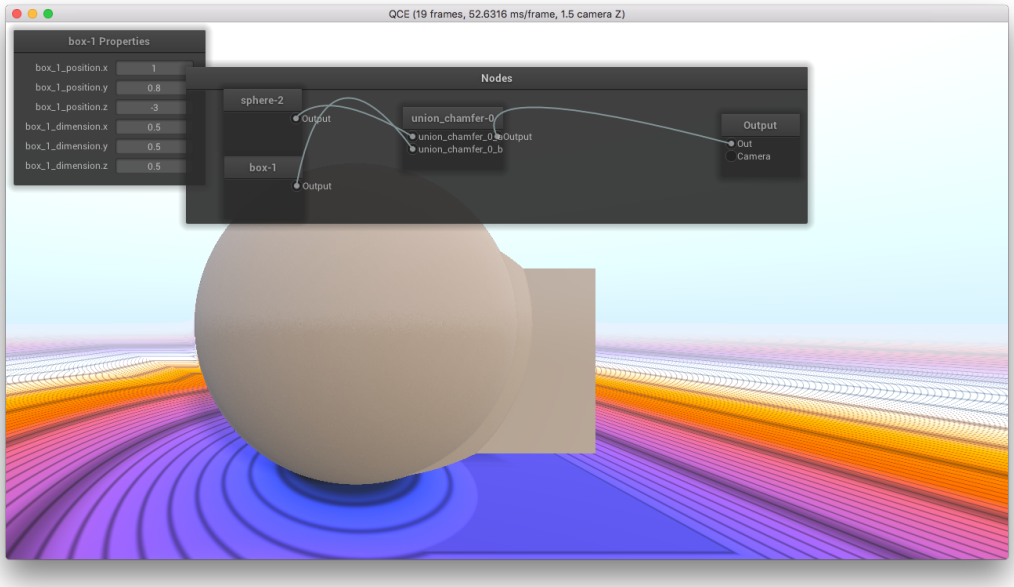


Abbildung 6.1.: Der Prototyp in Aktion

6.2. Domänenmodell und Klassendiagramm

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

In Abbildung 6.2 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.

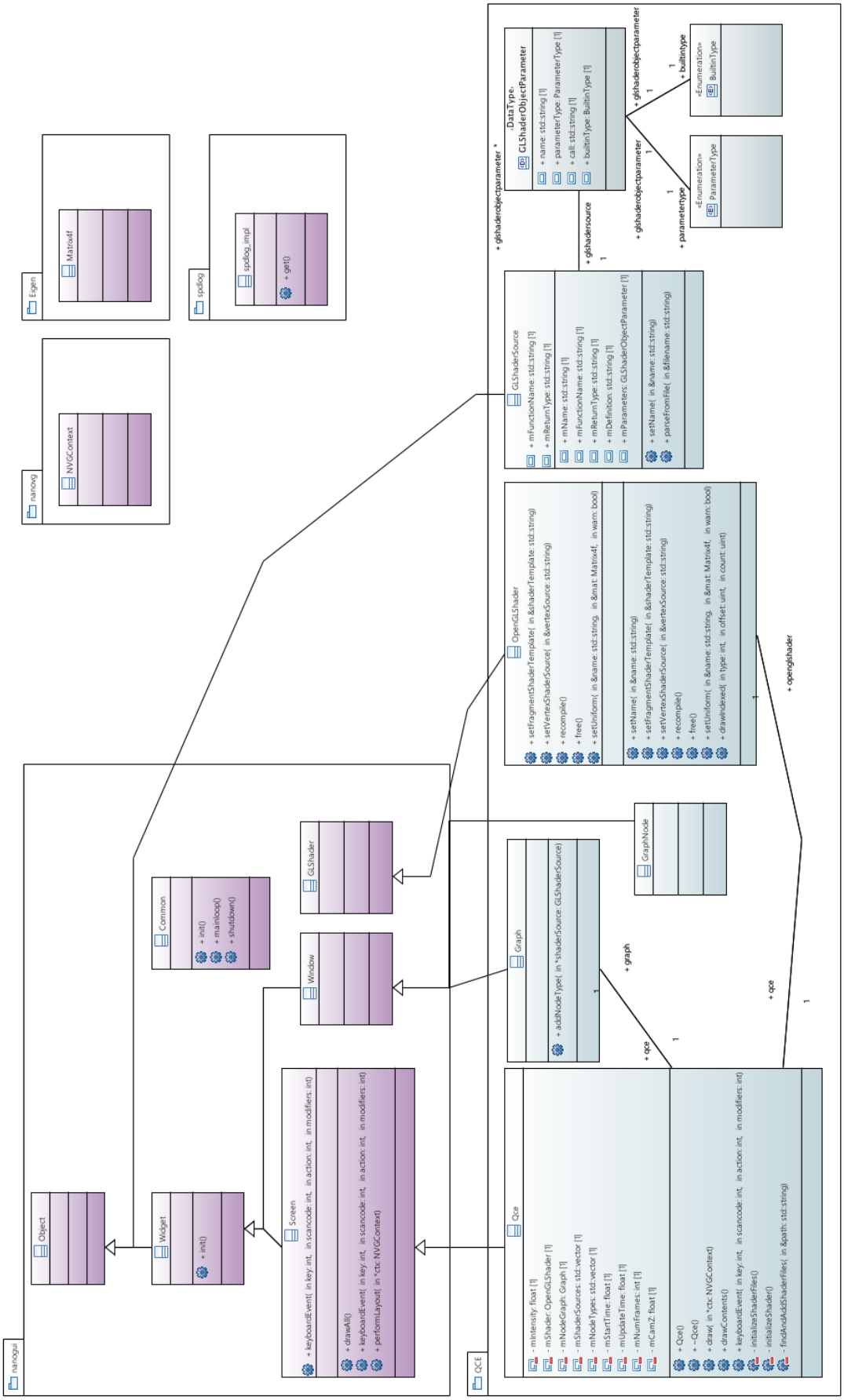


Abbildung 6.2.: Klassendiagramm der Prototyp-Applikation

6.3. Programmablauf

Nach dem Starten der Applikation erstellt diese die grafische Benutzeroberfläche und lädt alle externen Shader-Dateien im Unterverzeichnis “data”. Die Haupt-Shader-Datei wird als Grundlage für den Ausgabe-Shader genutzt, die Teil-Shader-Dateien bilden die wählbaren Objekte beziehungsweise Einträge im Kontextmenü des Graphen.

Nach dem initialen Berechnen und Darstellen der Benutzeroberfläche befindet sich die Applikation schliesslich in der Hauptschleife. Die Hauptschleife läuft so lange bis der Benutzer diese mittels Tastendruck auf die Escape-Taste unterbricht und damit die Applikation beendet.

In der Hauptschleife verarbeitet die Applikation diverse Events wie zum Beispiel Maus- und Tastatur-Eingaben. Die Verarbeitung findet in der Regel zuerst in den Kind-Klassen und dann schliesslich in der Hauptklasse statt. Dies erlaubt es diversen Komponenten, welche eben Kind-Klassen der Applikation sind, auf Ereignisse, wie zum Beispiel dem Erstellen einer Verbindung zwischen zwei Knoten, zu reagieren.

Immer wenn ein Knoten dem Haupt-Ausgabeknoten des Graphen direkt oder indirekt hinzugefügt wird, wird der Graph und somit der Shader neu berechnet beziehungsweise kompiliert.

Beendet der Benutzer schliesslich die Hauptschleife via Druck auf die Escape-Taste, so werden alle Ressourcen frei gegeben und die Applikation wird beendet.

Eine vereinfachte Darstellung des Beschriebenen findet sich in Abbildung 6.3. Eine detailliertere Darstellung findet sich in Abbildung 6.4.

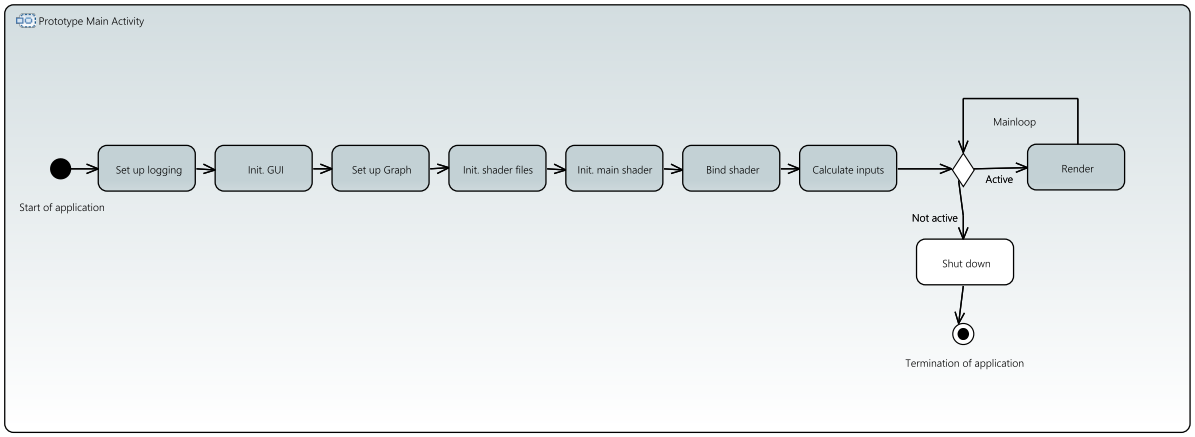


Abbildung 6.3.: Vereinfachte Darstellung des Haupt-Programmablaufes

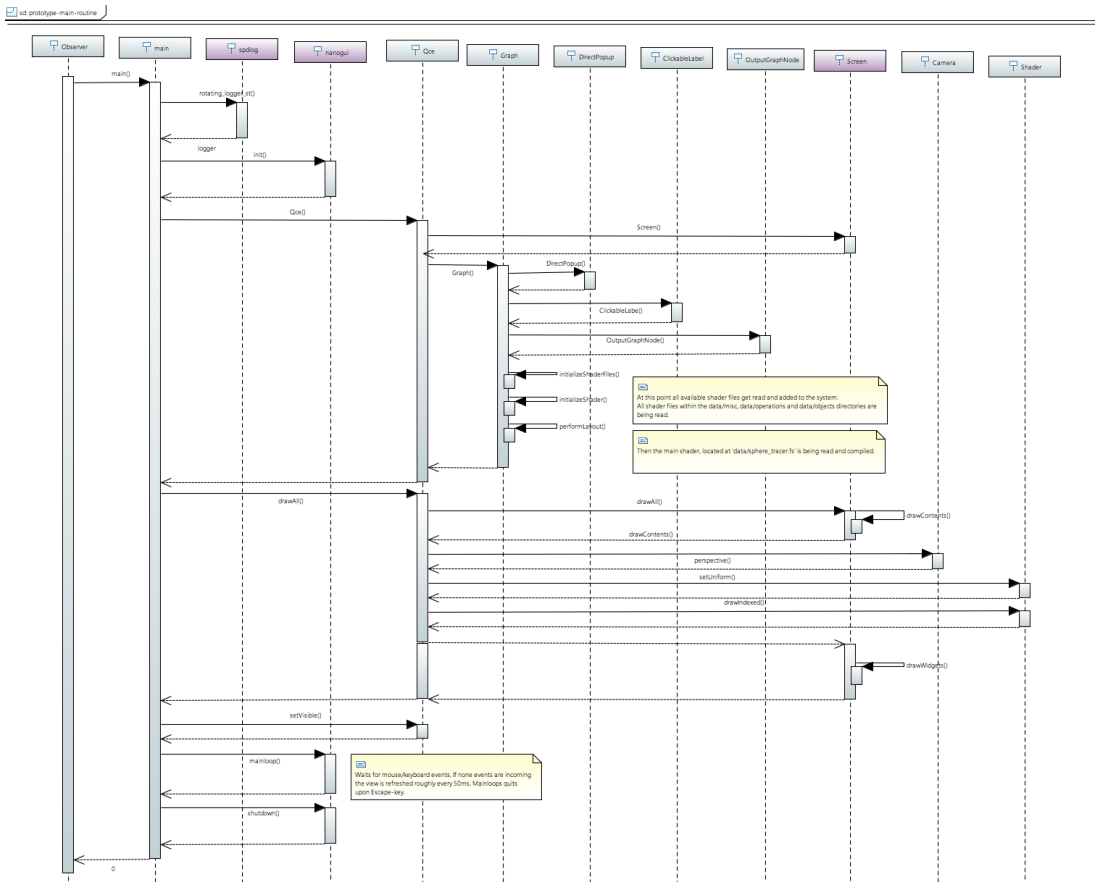


Abbildung 6.4.: Sequenz-Diagramm des Haupt-Ablaufes der Prototyp-Applikation

6.4. Komponenten

Durch die vorhergehende Projektarbeit — MTE7101 (siehe [1]) — sowie persönlicher Erfahrungen bei diversen Projekten, wählte der Autor zur Umsetzung des Prototypen die Software in der Tabelle 5.13 zur Umsetzung. Vor dieser Projektarbeit führte der Autor Recherchen betreffend Bibliotheken zur Erstellung und Darstellung von grafischen Benutzeroberflächen durch. Zuerst wurde Qt in Betracht gezogen. Diese Bibliothek ist sehr umfangreich und bietet bereits viele der benötigten Merkmale. Allerdings hat sie den Nachteil, dass sie relativ gross ist und den Prototypen aufbläht.

Der Autor stiess schliesslich auf NanoGUI von Wenzel Jakob. Es handelt sich dabei um eine minimalistische, plattformübergreifende (Grafik-) Bibliothek für OpenGL 3.x [11]. Die Bibliothek basiert auf NanoVG von Mikko Mononen. NanoGUI bietet automatische Layout-Erzeugung, C++11 Lambda-Callbacks, diverse Widget-Typen und Retina-fähiges Rendering [11]. Da dies genau den Anforderungen entsprechen zu schien, entschloss sich der Autor für die Verwendung von NanoGUI.

Während der Arbeit am Prototypen stellte sich jedoch heraus, dass viele der benötigten Komponenten nicht vorhanden sind und so mussten diese erst selbst entwickelt werden, so etwa der gesamte Graph inklusive Knoten, Ein- und Ausgängen und Verbindungen zwischen Knoten. Auch ein Kontext-Menü und einzelne, anwählbare Links (Text, welcher nach Klick eine Aktion ausführt) waren nicht vorhanden und mussten entwickelt werden.

Dies führte dazu, dass viel Zeit in die Entwicklung des Prototypen gesteckt wurde. So lag der Fokus anfangs primär darin einen lauffähigen Prototypen zu erhalten, anstatt zuerst ein (Software-) Design zu erstellen. Dies hatte schliesslich zur Folge, dass keine saubere Trennung (zum Beispiel in Schichten) stattfand und der Code weniger erweiterbar und wartbar wurde.

Die so gesammelten Erfahrungen flossen schliesslich in die Architektur für das darauffolgende Projekt (siehe Kapitel 5) ein. So fiel letztendlich der Entscheid doch wieder auf die Verwendung der Qt-Bibliothek zu Ungunsten von NanoGUI. Qt bietet eine viel umfangreichere Basis und beinhaltet viele der benötigten Komponenten und Features bereits.

Tabelle 6.1 zeigt die verwendeten Komponenten des Prototyps. Dabei handelt es sich bis auf wenige Ausnahmen um dieselben Komponenten wie für die angedachte Software-Architektur, siehe Tabelle 5.13. Es werden daher an dieser Stelle nur die zusätzlichen Komponenten aufgeführt. Im Unterschied zu der angedachten Software-Architektur kommt Qt bei dem Prototypen allerdings nicht zum Einsatz.

Tabelle 6.1.: Verwendete Software/Technologien

Komponente	**Beschreibung**	**Verweise**
NanoGUI	"NanoGUI is a a minimalistic cross-platform widget library for OpenGL 3.x. It supports automatic layout generation, stateful C++11 lambdas callbacks, a variety of useful widget types and Retina-capable rendering on Apple devices thanks to NanoVG by Mikko Mononen. Python bindings of all functionality are provided using pybind11." [11]	[2] [3]
Eigen	"Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms"[12]	[4]
spdlog	Schnelle, header-only C++ Bibliothek zur Protokollierung.	[5]
Jinja2CppLight	C++ Template-Bibliothek analog Jinja2 für Python.	[6]

6.5. Rendering

Auf das Rendering wird an dieser Stelle nicht näher eingegangen. Ein Teil wird in Abschnitt 6.1 beschrieben. Der Rest findet sich im nachfolgenden Kapitel über das verwendete Rendering, siehe Kapitel 7.

[2] `https://github.com/wjakob/nanogui`
[3] `https://github.com/memononen/NanoVG`
[4] `http://eigen.tuxfamily.org`
[5] `https://github.com/gabime/spdlog`https://github.com/gabime/spdlog
[6] `https://github.com/hughperkins/Jinja2CppLight`https://github.com/hughperkins/Jinja2CppLight

7. Rendering

Um schliesslich die erstellte Echtzeit-Animationen auch darstellen zu können — im Editor und im Player — wird ein Rendering-Verfahren benötigt.

Als Rendering-Verfahren soll primär das Ray-Tracing-Verfahren Sphere-Tracing zum Einsatz kommen. Auf dieses wird hier nicht eingegangen, da dieses bereits ausführlich in der vorhergehenden Projektarbeit behandelt wurde (siehe [1]).

Da bei der vorhergehenden Projektarbeit die *Berechnung von Normalen-Vektoren*, die *Repetitions-Operation* und deren *Komplexität* nicht ausreichend geklärt wurden beziehungsweise bei der Projektarbeit Fragen dazu aufkamen, werden die Punkte an dieser Stelle genauer erläutert. Weiter wird ein möglicher Ansatz gezeigt, um *herkömmliche 3D-Modelle (Meshes)* mittels Sphere-Tracing darstellen zu können.

Als Grundlage für das Rendering wird OpenGL gewählt, da diese Bibliothek plattformübergreifend und quelloffen ist. Mit der Einführung der Version 3 von OpenGL, beziehungsweise Version 2 von OpenGL ES, wurde die Rendering-Pipeline dahingehend geändert, dass diese nun nicht mehr fix vorgegeben sondern frei programmierbar ist[13] [14]. So gesehen findet das Rendering spätestens seit OpenGL Version 3 beziehungsweise OpenGL ES Version 2 immer per Shader statt. Ist der Aufwand der Programmierung der Rendering-Pipeline initial grösser, so ist eine frei programmierbare Pipeline jedoch viel flexibler.

Genau dieser Ansatz soll auch in der hier vorgestellten Software-Architektur für das Rendering verwendet werden: Ein modulares Rendering, welches jederzeit ersetzt werden kann. Da alles via Shader gerendert wird, ist dies so gesehen bereits gegeben. Szenen können — im Falle von Sphere-Tracing — rein aus internen Funktionen und Daten bestehen oder aber — im Falle der Verwendung von herkömmlichen Modellen (Meshes) — externe Daten beinhalten. So wäre beispielsweise zusätzlich zu Sphere-Tracing Deferred-Rendering [15] denkbar. Die Schnittstelle zwischen der Applikation und Shader, in Form von Uniform-Variablen, wird aber so oder benötigt um auf der CPU berechnete Matrizen an den Shader respektive die GPU weiterzureichen.

7.1. Berechnung von Normalen-Vektoren

Folgender Abschnitt basiert auf [1, S. 42 bis 43].

Die Oberflächen-Normale kann gemäss Hart, Sandin und Kauffman mittels dem Gradienten des Distanzfeldes für einen bestimmten Punkt auf einer impliziten Oberfläche berechnet werden [16, S. 292 bis 293].

Verwendet man dies nun zusammen mit den entsprechenden Distanzfunktionen, so ergeben sich folgende Gleichungen:

$$\boldsymbol{n}_x = f(x+\varepsilon, y, z) - f(x-\varepsilon, y, z) \tag{7.1}$$
$$\boldsymbol{n}_y = f(x, y+\varepsilon, z) - f(x, y-\varepsilon, z) \tag{7.2}$$
$$\boldsymbol{n}_z = f(x, y, z+\varepsilon) - f(x, y, z-\varepsilon) \tag{7.3}$$
$$\tag{7.4}$$

[0] `https://sites.google.com/site/richgel99/home`

Dabei ist $\boldsymbol{n} = \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix}$ die Normale der Oberfläche in Form eines Vektors mit drei Komponenten und f eine Distanzfunktion [16, S. 292 bis 293, 17, S. 13].

Es werden also insgesamt sechs Punkte abgetastet. ε ist dabei ein Wert um alle benachbarten Punkte eines gegebenen Punktes der impliziten Oberfläche entlang der Koordinatenachsen zu erhalten. Da die Berechnung der Normalen somit direkt von ε abhängt, wird für ε üblicherweise ein möglichst kleiner Wert verwendet. Hart, Sandin und Kauffman geben ε als die minimale Inkrementation eines (Licht-) Strahles an. Die Normale der Oberfläche sollte schliesslich noch normalisiert werden.

"Liefert die oben genannte Gradiente, bestehend aus 6 Punkten, eine zu geringe Genauigkeit, so kann diese gemäss Hart, Sandin und Kauffman erweitert werden [16, S. 293].
Die Erweiterung erfolgt durch Hinzunahme von Punkten, welche eine gemeinsame Kante haben. Dies erzeugt eine Gradiente bestehend aus 18 Punkten. Werden noch die Punkte hinzugenommen, welche gemeinsame Eckpunkte haben, so ergibt sich eine Gradiente bestehend aus 26 Punkten [16, S. 293]." [1, S. 43]

7.2. Repetitions-Operation

Wie unter [1, S. 37ff] aufgeführt, existieren für implizite Oberflächen diverse Operationen, wie Distanz- und Domänen-Operationen sowie -Deformationen.

Möchte man nun aber dasselbe Objekt mehrmals darstellen, so repetiert man dieses anhand einer oder mehrerer Achsen. Dies geschieht durch Anpassung der Domäne, also der Anpassung des Raumes, in dem sich eine implizite Oberfläche befindet. Es handelt sich dabei also um eine Domänen-Operation.

Die Anpassung der Domäne kann mittels der Modulo-Operation (*mod*, %) vorgenommen werden. Die Modulo-Operation gibt den (vorzeichenbehafteten) Rest einer Division zurück [18].

$$d(\boldsymbol{x}, \boldsymbol{c}) = \boldsymbol{x} \mod \boldsymbol{c} - \frac{\boldsymbol{c}}{2} \tag{7.5}$$

$$\text{repeat}(\boldsymbol{x}, \boldsymbol{c}) = f(d(\boldsymbol{x}, \boldsymbol{c})) \tag{7.6}$$

Dabei ist $\boldsymbol{x}$ der Punkt einer impliziten Oberfläche f und $\boldsymbol{c}$ der gewünschte Abstand zwischen den Objekten.

Das Objekt wird erst mit der Hälfte des Abstandes c, also $\frac{c}{2}$, anhand der Koordinatenachsen verschoben, um es im Rahmen des gewünschten Abstandes c zu zentrieren. Danach wird die Repetition der Domäne angewendet. Danach wird das Objekt zum Ursprung seines Koordinatensystemes zurück verschoben.

7.2.1. Komplexität

Es stellt sich nun die Frage, wie hoch die zusätzliche Komplexität bei Anwendung der Modulo-Operation ist. Der Autor geht davon aus, dass der Aufwand zur Darstellung von sich wiederholenden Objekten konstant ist. Die Methode zum Abtasten der Distanz, *castRay*, ändert sich in diesem Sinne konstant mit der Anzahl Objekte, die dargestellt werden sollen. Je mehr Objekte, desto höher der Aufwand. Nimmt man nun an, dass ein Objekt, welches wiederum eine Komposition von mehreren Objekten sein kann, wiederholt wird, so wird dennoch nach Erreichen der maximalen Anzahl Schritte oder der maximalen Distanz abgebrochen, was konstant ist. Vereinfacht gesagt wird schliesslich pro Iteration nur zurückgegeben, ob auf ein Objekt getroffen wurde oder nicht. Die Berechnung der Beleuchtung respektiv der Farbe einer Oberfläche erhöht die Komplexität natürlich nochmals (diese ist aufgrund der Verschachtelung $\mathcal{O}(n^2)$). Die Thematik soll an dieser Stelle jedoch nicht weiter vertieft werden, da dies den Rahmen dieser Projektarbeit deutlich sprengen würde.

7.3. Meshes

Während der Präsentation der vorhergehenden Projektarbeit, MTE7101, kam die Frage auf, ob es auch möglich ist "konventionelle" 3D-Modelle (Meshes) mittels Sphere-Tracing darzustellen. Die Modellierung bei der Anwendung von Sphere-Tracing findet üblicherweise mittels Distanzfunktionen statt und ist daher rein Mathematisch. Sphere-Tracing nutzt Distanzfunktionen und Distanzfelder zur Darstellung von impliziten Oberflächen [1, S. 31].

Betrachtet man nun aber Meshes genauer, so handelt es sich prinzipiell auch um Tiefeninformationen — wie bei einem Distanzfeld. Somit müsste es möglich sein ein Distanzfeld aus einem Mesh zu erzeugen und in einer dreidimensionalen Textur abzuspeichern. Dazu könnte ein dreidimensionales Raster erzeugt und das Mesh darin abgelegt werden. Danach müsste man die Distanzwerte von jedem Punkt des Rasters zur Oberfläche des Meshes in der dreidimensionalen Textur ablegen. Das Distanzfeld wäre dann zwar initial nicht $\in \mathbb{R}^3$, wie für das Sphere-Tracing benötigt, das Speichern in eine dreidimensionale Textur würde die Werte jedoch dann wieder auf $[0, 1]$ beschränken.

Dem Autor dieser Arbeit ist bisher kein solches Verfahren bekannt. Ob die oben beschriebene Methode wirklich funktioniert und wie gut die Resultate wären müsste ausprobiert werden.

8. Schlusswort

In dieser Projektarbeit wurde eine Software-Architektur für ein System zur einfachen Erstellung und Handhabung visueller Szenen in Echtzeit vorgestellt. Die Projektarbeit dient als Vorarbeit zur darauffolgenden Projektarbeit, MTE7103.

Zuerst wurde die *Vision* erarbeitet und festgehalten. Die vorgestellte Software-Architektur soll schliesslich zu einer Software zur Verwaltung und Darstellung von Echtzeit-Animationen führen. Dabei soll sie es Anwendern erlauben Echtzeit-Animationen in intuitiver Weise zu erstellen.

Ausgehend von der Vision wurden die *Akteure* bestimmt sowie *Use Cases* definiert. Die Use Cases 1 bis und mit 5 wurden bewusst eher grob granular gehalten um einen Überblick über die gesamte Software zu geben. Details finden sich in den Uses Cases 6 bis 10.

Aufgrund der Anforderungen und Erfahrungen des Autors wurde mögliche Software zur späteren Umsetzung der Software als *zusätzliche Anforderung* festgehalten. Ursprünglich sollte NanoGUI für die grafische Umsetzung genutzt werden. Bei der Implementation des Prototypen zeigte sich jedoch, dass die Bibliothek nur bedingt für die Anforderungen aus der Software-Architektur geeignet ist. Stattdessen wird nun Qt in Betracht gezogen, da dies den Grossteil aller Anforderungen bereits abdeckt.

Ausgehend von den Anforderungen wurden einzelne *Komponenten der Applikation* abgeleitet und bildlich dargestellt. 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.

Durch das Festhalten der Anforderungen und dem Identifizieren der einzelnen Komponenten wurde schliesslich das *Domänenmodell* erstellt, welches die Domäne mit ihren essentiellen Konzepten oder Objekten zeigt.

Um eine Vorstellung vom Ablauf der Applikationen zu erhalten, wurde der erste Use Case, “UC1: Betrachten einer Echtzeit-Animation”, als *Sequenz-Diagramm* dargestellt. Das Sequenz-Diagramm ist in diesem Sinne nicht vollständig, da dies ansonsten zu komplex und unübersichtlich würde. Es geht nur darum ein grundlegendes Verständnis der Applikationen zu entwickeln.

Die Architektur wurde dann weiter ausgearbeitet und schliesslich folgte daraus eine locker abgestufte Architektur (relaxed layered architecture). Diese wurde mittels *Paket-Diagrammen* grafisch dargestellt.

Um nun noch einen Schritt weiter zu gehen und der eigentlichen Umsetzung näher zu kommen, wurden *Klassendiagramme* der Applikation erstellt.

Zu dem Domänenmodell, dem Sequenz-Diagramm, den Paket-Diagrammen und den Klassendiagrammen ist zu sagen, dass bei diesen einige essentielle Konzepte beziehungsweise Objekte bewusst fehlen. Aufgrund der Vorgehensweise anhand des Unified Processes (UP) ist ein iteratives Arbeiten vorgesehen, da der UP auf agilen Ansätzen basiert. Das Ziel dieser Arbeit war nicht eine komplette und “korrekte” Architektur zu erstellen, welche sich direkt umsetzen lässt. Vielmehr dient diese als Grundlage für die darauffolgende Arbeit, MTE7013 — “Master Thesis”. Bei dieser sollen dann die Details in einzelnen Iterationsschritten erarbeitet und festgehalten werden.

Um Erfahrungen hinsichtlich der Umsetzung sammeln zu können sowie Erkenntnisse betreffend Software zu erhalten, wurde ein *Prototyp* umgesetzt. Der Prototyp deckt nicht die gesamte Architektur ab sondern demonstriert die Konzepte einzelner Teile der Software-Architektur. Der Prototyp erlaubt die Modellierung einer einfachen Szene, bestehend aus Primitiven, anhand der Graph-Komponente. Die Primitiven befinden sich in externen (Shader-) Dateien, welche zur Laufzeit zusammen mit einem Haupt-Shader geladen und zur Verfügung gestellt werden. Durch die

Modellierung wird der Haupt-Shader um Teile ergänzt und neu kompiliert, dies geschieht alles zur Laufzeit in Echtzeit. Die Darstellung findet schliesslich mittels dem als Sphere-Tracing bekannten Ray-Tracing-Verfahren statt. Dieses wurde in der vorhergehenden Projektarbeit vorgestellt (siehe [1]).

Im letzen Abschnitt wurde näher auf das Rendering-Verfahren eingegangen. Zudem wurde mit der Repetition eine weitere Operation für implizite Oberflächen vorgestellt und deren Komplexität betrachtet. Schliesslich wurde eine Möglichkeit in Betracht gezogen, wie herkömmliche 3D-Modelle (Meshes) via Sphere-Tracing dargestellt werden können. Dass diese Möglichkeit funktioniert ist nicht gegeben, es handelt sich lediglich um eine Idee.

Im Nachhinein betrachtet wurde in der ersten Hälfte der Projektarbeit viel Zeit in den Prototypen investiert. Der Fokus lag anfangs primär darin einen lauffähigen Prototypen zu erhalten, anstatt zuerst ein (Software-) Design zu erstellen. Dies hatte schliesslich zur Folge, dass keine saubere Trennung (zum Beispiel in Schichten) stattfand und der Code weniger erweiterbar und wartbar wurde. Als Schlussfolgerung kann man sagen, dass auch bei iterativem Vorgehen und der Erstellung von Prototypen zuerst zumindest eine minimale Dokumentation zum Beispiel in Form von Domänenmodellen, Klassendiagrammen und Sequenz-Diagrammen erstellt werden sollte.

Literatur

[1] S. Osterwalder, *Volume ray casting - basics & principles*, 14. Feb. 2016.

[2] Wikipedia Foundation, *Demoszene*, de, Page Version ID: 149578258, Dez. 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. Rumbaugh, I. Jacobson und G. Booch, *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004, ISBN: 0-321-24562-8.

[11] W. Jakob. (2016). Wjakob/nanogui: Minimalistic GUI library for OpenGL, Adresse: `https://github.com/wjakob/nanogui` (besucht am 04. 08. 2016).

[12] Benoît Jacob und Gaël Guennebaud. (2016). Eigen, Adresse: `http://eigen.tuxfamily.org/index.php?title=Main_Page` (besucht am 04. 08. 2016).

[13] OpenGL Foundation. (2015). Fixed function pipeline - OpenGL.org, Adresse: `https://www.opengl.org/wiki/Fixed_Function_Pipeline` (besucht am 04. 08. 2016).

[14] ——, (2015). Rendering pipeline overview - OpenGL.org, Adresse: `https://www.opengl.org/wiki/Rendering_Pipeline` (besucht am 04. 08. 2016).

[15] T. Saito und T. Takahashi, „Comprehensible rendering of 3-d shapes“, *SIGGRAPH Comput. Graph.*, Bd. 24, Nr. 4, S. 197–206, Sep. 1990, ISSN: 0097-8930. DOI: `10.1145/97880.97901`.

[16] J. C. Hart, D. J. Sandin und L. H. Kauffman, „Ray Tracing Deterministic 3-D Fractals“, *SIGGRAPH Comput. Graph.*, Bd. 23, Nr. 3, S. 289–296, Juli 1989, ISSN: 0097-8930. DOI: `10.1145/74334.74363`.

[17] J. C. Hart, „Ray tracing implicit surfaces“, *Siggraph 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces*, S. 1–16, 1993.

[18] L. Maignan und F. Gruau, „Integer gradient for cellular automata: Principle and examples“, in *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, Ser. SASOW '08, Washington, DC, USA: IEEE Computer Society, 2008, S. 321–325, ISBN: 978-0-7695-3553-1. DOI: `10.1109/SASOW.2008.52`.

Abbildungsverzeichnis

Tabellenverzeichnis

Auflistungsverzeichnis

Anhang

A. Meeting minutes

20160224

```
No.:              00
Date:             24.02.2016 11:30 - 12:00
Place:            Mr. Fuhrer's Office
Involved persons: Claude Fuhrer    (CF)
                  Sven Osterwalder (SO)

Meeting minutes 2016-02-24
==========================

Presentation and discussion of the current state of the work (CF, SO)
---------------------------------------------------------------------

* Kick-off meeting of the project work. (CF, SO)

* Try already choosing or at least looking at data formats for exporting data,
  one example is X3D. (CF)

* Work out details of sphere tracing which were not that clear during the
  presentation of the previous project work: normal calculation and the modulo
  operation. (CF)

* If you are in good schedule, try maybe exploring methods for importing and
  rendering meshes using sphere tracing. (CF)

Further steps/proceedings (CF, SO)
----------------------------------

* Work on prototype
* Create a framework for the documentation
* Start documenting on the progress

TODO for the next meeting
=========================

* Present the current state of the work (SO)
* Discuss the current state of the work (CF, SO)
* Define further steps/proceeding (CF, SO)

Scheduling of the next meeting
==============================

Date:  tba
Place: tba
```

```
No.:              01
Date:             01.04.2016 10:00 - 10:30
Place:            Mr. Fuhrer's Office
Involved persons: Claude Fuhrer    (CF)
                  Sven Osterwalder (SO)

Meeting minutes 2016-04-01
==========================

Presentation and discussion of the current state of the work (CF, SO)
---------------------------------------------------------------------

* Framework is set up, there has been quite some work on the prototype (SO)

  * Procedure / progress is OK, however a manual should be provided on how to
    compile and use the prototype or tools respectively. (CF)

* The preceding project may used as reference and be referenced directly in
  context. (CF)

* Have a look at the garbage collector of C++13. (CF)

* It would be good to have a rough draft of the documentation in plus/minus
  three weeks. (CF)

* The performance/complexity/time costs should be mentioned in the
  documentation as well, as this was a question at the presentation of the
  previous project work. (CF)

* The performance/complexity/time costs should be mentioned in the
  documentation as well, as this was a question at the presentation of the
  previous project work. (CF)


Further steps/proceedings (CF, SO)
----------------------------------

* Process TODOs
* Expand basic chapters about illumination models according to TODOs
* Expand chapters "scope" and "administrative"
* Re-work introduction
* Add management summary

TODO for the next meeting
=========================

* Present the current state of the work (SO)
* Discuss the current state of the work (CF, SO)
* Define further steps/proceeding (CF, SO)

Scheduling of the next meeting
==============================

Date:  06.12.2015, 1400
Place: Skype
```

```
No.:              02
Date:             27.04.2016 14:00 - 14:30
Place:            Mr. Fuhrer's office
Involved persons: Claude Fuhrer    (CF)
                  Sven Osterwalder (SO)

Meeting minutes 2016-04-27
==========================

Presentation and discussion of the current state of the work (CF, SO)
---------------------------------------------------------------------

* The progess of the prototype is quite good. The graph component is mostly
  working, shader is being composed and compiled in real time. (SO)

* What is a good extent of the documentation? (SO)
  * It should be between 20 and 100 pages in total. (CF)

* Does the tense of the documentation matter? Should it be rather written in
  present or in past? (SO)
  * The tense does not matter, do as you like. (CF)

* The documentation should be based on contents rather than on a schedule. (CF)

* Concerning the time schedule, should it represent an ideal or represent the
  actual situation/progress? (SO)
  * It should represent the actual situation/progress. (CF)

* It would be good if a first, rough draft of the documentation would be
  finished until the end of May 2016. (CF)


Further steps/proceedings (CF, SO)
----------------------------------

* Fix bugs of the prototype
* Start documenting

TODO for the next meeting
=========================

* Present the current state of the work (SO)
* Discuss the current state of the work (CF, SO)
* Define further steps/proceeding (CF, SO)

Scheduling of the next meeting
==============================

Date:  tba
Place: tba
```

```
No.:              03
Date:             15.07.2016 10:30 - 11:00
Place:            Mr. Fuhrer's office
Involved persons: Claude Fuhrer    (CF)
Sven Osterwalder (SO)

Meeting minutes 2015-11-15
==========================

Presentation and discussion of the current state of the work (CF, SO)
---------------------------------------------------------------------

* The progress has slowed down a lot unfortunately. This is mostly
  due to the focus was laid on the exams, the block-module as well as
  on the second specialization module of the studies during the last couple of
  weeks. There was no progress on the prototype anymore since the last meeting
  except for some small bug fixes. The documentation is currently being worked
  on, the progress is although not as good as expected. (SO)

* The expert, mr. Eric Dubuis, was incorporated. He will supervise the master
  thesis as an expert as he has profound knowledge on the topics of the thesis.
  You may send him the first and this project work as information regearding
  the master thesis by email. (CF)

* The presentation of this project work will most likely be held on the 12th of
  August 2016. (CF)

* Hand-in of this project work is the 29th of July but by no lather then the
  5th of August 2016. (CF)


Further steps/proceedings (CF, SO)
----------------------------------

* Process TODOs
* Expand chapters according to TODOs
* Re-work introduction and abstract
* Add management summary
* Add meeting minutes

TODO for the next meeting
=========================

-

Scheduling of the next meeting
==============================

-
```