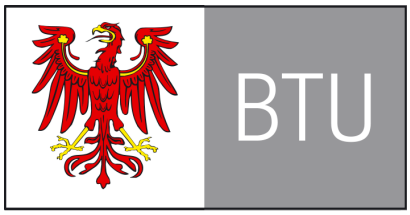
Fakultät für Mathematik,
Naturwissenschaften und
Informatik
Institut für Informatik,
Informations- und Medientechnik
Lehrstuhl Grafische Systeme



Brandenburgische Technische Universität Cottbus

Masterarbeit

Entwicklung einer Software zur Erzeugung und Echtzeitpräsentation von Innenräumen

Udo Bischof, 2103867

26. Januar 2009

1. Gutachter: Prof. Winfried Kurth
2. Gutachter: Prof. Dominik Lengyel

Inhaltsverzeichnis

1 Motivation **8**
1.1 Voraussetzungen 10

2 Visualisierung als Werkzeug **12**
2.1 Klassen von Visualisierungen 14
2.2 Architekturvisualisierung 16
2.3 Allgemeine Anforderungen an eine Visualisierung . . . 17
2.3.1 Grad der Interaktivität 18
2.3.2 Wahl der Perspektive 18
2.3.3 Wahl des Realismus 19
2.3.4 Fokussierung des Betrachters 20
2.4 Fazit 21

3 Vorbetrachtung / Anforderungsanalyse **22**
3.1 Grundlegende Anforderungen 22
3.1.1 Grundmodule des Gesamtsystems 24
3.2 Wahl der Plattform 26
3.3 Funktionale und Nichtfunktionale Anforderungen . . . 29

4 GroIMP als Plattform **31**
4.1 Modellierung mit GroIMP und XL 32
4.1.1 Wichtige GUI-Elemente 32
4.1.2 3D-Modellierung über XL 34
4.1.3 Anwendungen von XL und GroIMP 35
4.1.4 Einbindung von XL und GroIMP in eine Produktionskette 36
4.2 Technische Hintergründe 38
4.2.1 Der Szenegraph 39
4.2.2 Traversieren des Szenegraphen 43
4.2.3 Hinzufügen und Entfernen von Knoten 44
4.2.4 Graph-Locking 45
4.2.5 Die Klasse Job 46
4.3 Transactions 50

4.4 Attribute von Knoten 50
4.5 Die Registrierung 51
4.6 Import/Export 52
4.7 Grundzüge einer Erweiterung 52
4.7.1 Implementierung eigener Knoten 57
4.7.2 Toolbars und Buttons 59
4.7.3 Panels 60
4.7.4 Layouts 62
4.7.5 Hooks, Listener und Consumer 62
4.8 Fazit 67

5 2D-Canvas 70
5.1 Motivation 70
5.2 Grundlagen 71
5.3 Grundkonzepte 74
5.3.1 Canvas 74
5.3.2 Objekte auf dem Canvas 74
5.3.3 Zeichnen von Objekten 75
5.3.4 CanvasModifier und CanvasParser 77
5.3.5 Selektionen 77
5.3.6 Observer 78
5.4 Funktionalität 79
5.5 Fallbeispiel: Bildbetrachter 80
5.6 Fazit 82

6 ArchiViz als Erweiterung von GroIMP 83
6.1 ArchiViz - Basisarchitektur 84
6.1.1 Module 84
6.2 Der ArchiViz-Szenegraph in GroIMP 86
6.2.1 Wichtige Knotentypen 87
6.2.2 Knotenbeziehungen 89
6.2.3 Sichtbarkeit von Knoten 90
6.3 Projektverwaltung 91
6.4 Interaktion zwischen dem ArchiViz-Canvas und GroIMP 92
6.4.1 Darstellung der Knoten in 2D 92
6.4.2 Bekanntmachung von Änderungen 94
6.4.3 Selektion 95
6.4.4 Platzierung und Rotation 97
6.5 Durchlässigkeit zwischen GroIMP und ArchiViz 98
6.6 Persistenz 99

7 Grundrisseditor **100**
7.1 Sketching 100
7.1.1 Erzeugung der logischen Raumverteilung 102
7.1.2 Bestimmung von Räumen 102
7.2 Fazit 104

8 Textureditor **105**
8.1 Motivation 105
8.2 Photogrammetrie als Grundlage 108
8.3 Wichtige Algorithmen 110
8.3.1 Target-To-Source 110
8.3.2 Linsenkorrekturfilter 111
8.4 Fotografie mit festem Kamerastandort 112
8.5 Fotografie mit variablem Kamerastandort 114
8.6 Linsenkorrektur 116
8.7 Stitching (Registrierung) 117
8.7.1 Formale Grundlagen des implementierten Verfahrens 119
8.7.2 Stitching mit projektiven Abbildungen 128
8.7.3 Stitching-Reihenfolge 129
8.8 Herstellung der Orthogonalprojektion 129
8.9 Zusammenfassung des Algorithmus 130
8.10 Fazit 132

9 Aktionen und Reaktionen **136**
9.1 Grundprinzip 136
9.1.1 Aktionen und Reaktionen als Graph 137
9.1.2 Umsetzung für ArchiViz 140
9.1.3 Verankerung in GroIMP 142
9.1.4 Auswertung im Präsentationsmodus 143
9.2 Fazit 144

10 Zustandseditor **146**
10.1 Zustände von Knoten 146
10.2 Verankerung in GroIMP 148
10.3 Fazit 150

11 Fallbeispiel **152**
11.1 Fazit 165

12 Ausblick **166**

12.1 Weitere Grundrisswerkzeuge 166
12.2 ArchiViz unter Verwendung von XL 166
12.3 Zustände von Knoten 167
12.4 Aktionen und Reaktionen 169
12.5 Dateninput durch Scanning 175

13 Diskussion **177**

A Mathematische Grundlagen **179**
A.1 Eigenschaften von Relationen 179
A.2 Grundlagen zu Abbildungen 180
A.2.1 Abbildungsfunktion 180
A.2.2 Homogene Koordinaten 180
A.2.3 Transformationsfunktion 181
A.2.4 Transformationsverkettung 181
A.3 Abbildungen im $\mathbb{R}^2$ 181
A.3.1 Abbildungshierarchie 181
A.3.2 Projektive Abbildung 184
A.3.3 Inversion der projektiven Abbildung 187
A.4 Abbildungen im $\mathbb{R}^3$ 188
A.4.1 Affine Transformation 188

B Listings **190**

C Aufgabenstellung **194**
C.1 Funktionale Anforderungen 194
C.1.1 Datenformat 194
C.1.2 Textureditor 194
C.1.3 Editor 195
C.1.4 Raumerzeugung 195
C.1.5 Interaktivität 196
C.1.6 Platzierung von Objekten 196
C.1.7 Viewer 196
C.1.8 3D-Scanner 197
C.2 Beispielhafte Beschreibung des Workflows 197

Literatur **201**

Danksagung

Ich möchte mich an dieser Stelle bei all jenen bedanken, welche diese Arbeit ermöglichten und unterstützten.
An vorderster Stelle möchte ich mich bei Prof. Kurth und Prof. Lengyel dafür bedanken, dass eine interdisziplinäre Arbeit wie diese überhaupt möglich wurde. Darüber hinaus gilt mein Dank insbesondere Uwe Mannl, welcher mir bei vielerlei Problemen mit Rat und Tat zur Seite stand und Ole Kniemeyer, der mir in den Tiefen GroIMPs weiterhalf. Bei Herrn Dr. Preuss bedanke ich mich für die Arbeit als Lektor und für die gesamte Unterstützung, die er mir zukommen ließ.
Des Weiteren bedanke ich mich bei meiner Familie für die durchgängige Unterstützung.

1 Motivation

Die Gestaltung, Möblierung und Ausstattung von Innenräumen ist besonders bei Gebäuden mit historischem Hintergrund von großer Bedeutung. Viele alte Gebäude, wie z. B. Villen oder Schlösser verschiedenster Epochen und Baustile enthielten im Zeitverlauf verschiedene, teils wertvolle Gegenstände. Die Anzahl und Platzierung der Gegenstände variierte zu verschiedenen Zeitpunkten, sodass das Aussehen der Innenräume sich massiv verändern konnte. Darüber hinaus konnte der Zweck des Gebäudes und damit das Interieur sich verändern, sodass Gebäude, welche heute z. B. als Schulen oder Museen genutzt werden, früher eine ganz andere Bedeutung hatten. Es gilt grundsätzlich die allgemeine Annahme, dass jede Veränderung einer gewissen Planung unterliegt, welche vor der Realisierung visualisiert werden sollte. Ebenso ist eine Rückführung auf einen früheren Zustand von Interesse: Die Rekonstruktion und Präsentation solch eines früheren Zustandes stellt sich als schwierig dar, wenn frühere Objekte nicht mehr verfügbar sind, da sie z. B. einem neuen Besitzer angehören oder nicht mehr existent sind. Deshalb bietet es sich an, dies in einer virtuellen Präsentation zu tun. Das ermöglicht den direkten Vergleich verschiedener Zustände eines Innenraumes oder mehrerer Räume.
Es besteht also Interesse daran, Ideen zunächst konzeptionell zu entwickeln und mit einem anderen Zustand, z. B. dem heutigen oder einem früheren zu vergleichen.
Die eben erläuterte Motivation zeigt einen Vorgang, der typischerweise eine größere Menge an verschiedenen Softwareprodukten für eine erfolgreiche Umsetzung erfordert. Diese können - insgesamt betrachtet - einen erheblichen Kostenfaktor darstellen. Darüber hinaus steigt mit jedem zu verwendenden Produkt die Komplexität, ebenso die Gesamtmenge der Funktionalität, die nicht benötigt wird. Zerlegt man den beschriebenen Vorgang in verschiedene Einzelvorgänge, sind derzeit etwa vier verschiedene Softwareprodukte not-

wendig[1], um diese zu erzeugen:

- Zusammenfügen von Fotos zu Wandtexturen
- Erzeugung der Innenraumgeometrie für die Präsentation und Platzierung von verschiedenen Objekten.
- Festlegung der Interaktivität und Definition des Informationsgehaltes
- Durchführung der Präsentation

Es gibt bereits vielfältige Lösungen auf dem Markt, die jede dieser Teilaufgaben z.T. perfekt umsetzen. Doch es bleiben dennoch verschiedene Softwareprodukte, welche oft nur schwierig über verschiedene Schnittstellen zu verbinden sind. Darüber hinaus hat jede Software ihr eigenes Bedienkonzept, weshalb sie auf unterschiedliche Art Probleme lösen und zur Herausforderung werden, wenn die Abweichungen zwischen den Produkten sehr stark sind. In einigen Fällen ist es sogar so, dass bestimmte Produkte keine geeigneten gemeinsamen Schnittstellen besitzen, und somit Programme heran gezogen werden müssen, die allein die Lücke in der Schnittstelle z. B. durch Importieren, Umwandeln und Exportieren von Daten füllen. Ein Beispiel ist der Austausch von 3D-Geometrie: Die Industrie hat sich dort auf einige wichtige Formate geeinigt. Unterstützt allerdings ein Programm diese Formate nicht, bleibt es in den meisten Fällen außen vor. Ein Programm kann in einem Teilbereich sehr innovativ und produktiv sein, jedoch nützt dies bei größeren Projekten nichts, wenn die Daten nicht ausgetauscht werden können.
Es liegt also auf der Hand, dass an dieser Stelle eine Reduktion der verschiedenen Produkte zur exakten Erfüllung der Aufgaben notwendig ist. Diese Reduktion sollte möglichst groß sein, im optimalen Fall auf ein einziges Softwareprodukt.
In dieser Arbeit sollen alle Aspekte der Anteile der Anforderung konzeptionell beleuchtet werden mit dem Anspruch, die Erfüllung aller Aufgaben auf ein einziges Softwareprodukt zu reduzieren. Dies bedeutet, dass ein Teil der Arbeit die beispielhafte Implementierung der Konzepte und deren Zusammenfassung zu einem Softwareprodukt umfasst.

[1] Dies ist eine vereinfachte Annahme. Es gibt natürlich Softwareprodukte, die mehrere Aufgaben zusammenfassen, doch dies ist nicht der übliche Fall.

Die vorliegende Arbeit beschäftigt sich zunächst allgemein mit dem Begriff „Visualisierung“. Der Grund dafür ist, dass insgesamt eine solche erzeugt werden soll. Es wird Wissen über visuelle Informationen - nämlich über die interaktive Präsentation - vermittelt.
Darauf folgend, im Kapitel 2.3, werden die Anforderungen an das gesamte System, insbesondere an die darunter liegende Plattform, genauer beleuchtet. Eben diese Plattform wird im darauf folgenden Kapitel 4 aus zwei verschiedenen Perspektiven, nämlich der eines Anwenders und der eines Entwicklers, betrachtet. Diese genauere Betrachtung hat den Grund, dass diese Plattform bisher nur geringfügig dokumentiert wurde und dieses Kapitel einen Einstieg für einen breiten Anwenderkreis bieten soll.
Das Kapitel 5 dient zur Erläuterung einer recht komplex implementierten, aber einfach zu nutzenden 2D-Plattform, welche zur Umsetzung einiger Funktionalitäten notwendig war.
Da das vollständige System aus verschiedenen Teilen besteht, widmen sich die darauf folgenden Kapitel den verschiedenen Modulen. Die konzeptionelle Betrachtung bildet jeweils die Grundlage, während an einigen Stellen ebenso auf technische Details eingegangen wird. Insbesondere das Kapitel 8 beschäftigt sich mehr mit einigen formalen Details, da dieses Kapitel im Verhältnis zum Rest der Arbeit einen großen Teil ausmachte.
Die verschiedenen Module bilden zusammen den gesamten Arbeitsvorgang, um von den fotografierten Innenwänden bis hin zur Durchführung der Präsentation innerhalb eines einzigen Systems zu kommen. Das Kapitel 11 fasst diesen Arbeitsvorgang vollständig zusammen und zeigt ebenso die implementierten Funktionalitäten beispielhaft.
Es wird also im Zuge dieser Fallstudie geprüft, ob ein Zusammenfassen des gesamten Arbeitsvorganges möglich bzw. sinnvoll ist. Ebenso ist wichtig, ob - ausgehend von den vorgestellten Konzepten - weiterführende Konzepte denkbar sind. Das Kapitel 12 beleuchtet die Fortführung einiger Konzepte.

1.1 Voraussetzungen

Als Voraussetzung in der vorliegenden Arbeit werden folgende Punkte angenommen:

- Grundlegende Kenntnisse in der Computergrafik

- Grundlegende Kenntnisse in 3D-Modellierprogrammen
- Kenntnisse in Programmierung mit Java[1]

Der letzte Punkt trifft nicht auf alle Kapitel zu. Darüber hinaus wurde versucht, ein Gleichgewicht zwischen technischen Details und fachlichen Betrachtungen aufrecht zu erhalten.

[1] Als grundlegende Literatur für Java in computergrafischer Verwendung wird Wippler (2006) empfohlen.

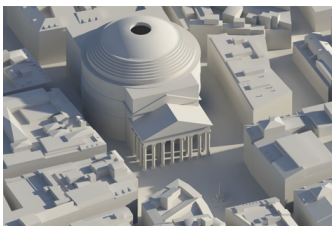
2 Visualisierung als Werkzeug



(a) Höhlenmalerei in der Höhle Lascaux in der Nähe von Montignac



(b) „helix pteron“ von Leonardo da Vinci, entn. aus Dönitz



(c) Visualisierung des Pantheon in Rom

Abbildung 2.1: Visualisierungen zu unterschiedlichen Zeitpunkten menschlicher Geschichte

Der Begriff „Visualisierung“ bedeutet, abstrakte Informationen in eine für Menschen sichtbare Darstellung umzuwandeln.
Nur wenige Menschen würden sich mit der Darstellung abstrakter Zahlenkolonnen oder Formeln begnügen. Zum Erkennen von Zusammenhängen und der Vermittlung von Ideen kann dies hinderlich sein, denn: Der Mensch denkt und erinnert in Bildern, somit liegt es nahe, Ideen und Konzepte, aber auch Sichtweisen und Interpretationen von Sachverhalten oder Objekten in eben diesen darzureichen.
Schon vor nachweislich mehr als 20000 Jahren wurden Zeichnungen angefertigt, um die Wirklichkeit abzubilden: Abbildung 2.1a ist ein Beispiel aus der Höhle Lascaux bei Montignac. Diese Zeichnung bildet die damalige natürliche Umgebung ab, indem der Künstler diese beobachtete und visualisierte - in Form einer Höhlenmalerei, eine der ältesten Kunstformen.
Nun liegt es in der Natur des Menschen, Ideen zu entwickeln. Um diese Ideen festzuhalten und möglicherweise anderen zu präsentieren, entwickelte sich eine weitere Erscheinungsform der Visualisierung. Hierbei ging es darum, eben diese Ideen in verschiedenartiger Weise darzubringen, damit auch andere Menschen einen Eindruck dieser Idee vermittelt bekommen. Abbildung 2.1b zeigt eine durch Leonardo da Vinci erstellte Zeichnung, um ein noch

nicht existierendes Objekt darzustellen: Nämlich die Idee eines Fluggerätes („Luftschrauber“). Solche Zeichnungen vermittelten bereits einen Eindruck von einem Objekt, welches noch gar nicht existiert - außer in den Gedanken des Erzeugers. Visualisierungen werden also benötigt, um Ideen und Konzepte an andere Menschen auf einfache Art und Weise weiter zu geben.
Hierbei erlebte die Visualisierung einen beispiellosen Wandel. Angefangen bei eben jenen einfachen Zeichnungen bis hin zu immer komplexer werdenden Konstruktionen, wobei sich sowohl der Anspruch als auch das Werkzeug veränderte. Die Visualisierung wurde stark durch die Kunst und deren Techniken beeinflusst, da sich die Kunst stark mit der menschlichen Wahrnehmung und der Wirkung darauf befasst, z. B. die durch Bilder.
Mit der Entwicklung und dem Fortschreiten der Computertechnik wurden die Einflüsse des rechengestützten Entwickelns[1] immer stärker. Die Verfahren der Visualisierung veränderten sich und somit auch die Wirkung auf den Menschen.
Wo früher ein oder mehrere Bilder einen Eindruck vermitteln sollten, ist es heute möglich, ein noch nicht existierendes Objekt im Rechner zu entwerfen - oder ein existierendes nachzubauen - um es anschließend aus beliebigen Blickwinkeln zu betrachten. Es können Videosequenzen mit virtuellen Kamerafahrten und Animationen entwickelt werden, ebenso kann der Betrachter als Interakteur eingesetzt werden. In verschiedenen Echtzeitumgebungen kann der Betrachter entscheiden, aus welchem Blickwinkel er das Objekt betrachten möchte, er ist nicht mehr gebunden an den Standpunkt des Entwicklers bzw. Künstlers.
Abbildung 2.1c zeigt einen Ausschnitt einer Visualisierung des Pantheons in Rom. Diese stellt das Pantheon, den Platz davor und die Gebäude darum herum stark vereinfacht dar. Dies zeigt: Eine Visualisierung muss der Wirklichkeit nicht exakt entsprechen und muss es durchaus nicht immer versuchen. Oftmals ist das Weglassen von (visuellen) Informationen nützlich, um eine Vereinfachung zu erreichen und um damit auf ganz bestimmte Informationen hinzuweisen. Dies macht den Informationsgehalt eines solchen Modells besser durchschaubar und hilft, das Verständnis zu erhöhen.

[1] CAD - Computer Aided Design

Natürlich kann - sofern es der Intention des Designers[1] entspricht - der Betrachter zusätzlich in seinen Möglichkeiten der Interaktion eingeschränkt werden. Oftmals ist dies auch sinnvoll, da manchmal nicht alle Aspekte eines Objektes von Interesse sind.

2.1 Klassen von Visualisierungen

Nachfolgend wird eine mögliche Variante der Klassifikation von Visualisierungstypen gezeigt:

- Technische bzw. wissenschaftliche Visualisierung
  Diese kann durchaus einen Design-Prozess enthalten, soll aber eher die Funktionalität oder Herstellungsweise eines Objektes darstellen. Dazu gehört auch die Datenvisualisierung z. B. von Geodaten.

- Künstlerische Visualisierung

- Funktionale Visualisierung
  Dies sind üblicherweise Visualisierungen, mit denen Menschen in ihrem täglichen Leben umgehen. Ein Beispiel sei ein GPS[2]-Navigationsgerät: Straßen werden in mehr oder minder abstrakter Form für den Anwender visualisiert. Theoretisch ist jede GUI[3] eine Form der Visualisierung mit Interaktionsmöglichkeiten, zumindest unterliegt sie den gleichen Anforderungen.

Die drei genannten Klassen bilden für sich genommen die Extrema der Visualisierung. Zwischen ihnen herrscht ein weicher Übergang. Das bedeutet, dass sich die Eigenschaften nicht klar abgrenzen lassen.
In (Schumann und Müller, 2000, S. 5 ff.) wird eine grundlegende Klassifikation der Ziele wissenschaftlich-technischer Visualisierungen, also von einem Teil obiger Klassifikation gezeigt:

[1] Das Wort *Designer* fasst die Begriffe „Entwickler“ und „Künstler“ relativ treffend zusammen. Ein Designer lässt sich nur schwer vollständig in die Kunst einordnen - je nach Anwendungsgebiet - da oftmals technische Aspekte mit in den Design-Prozess einfließen.
[2] Global Positioning System
[3] Graphical User Interface

„Es soll

- die Analyse
- das Verständnis und
- die Kommunikation

von

- Modellen,
- Konzepten und
- Daten

in der Wissenschaft und im Ingenieurbereich erleichtert werden.“

Hierbei wird in erster Linie Bezug auf die Visualisierung einer bestimmten Datenmenge genommen, um diese auszuwerten. Diese Daten können Messwerte oder Resultate bestimmter Berechnungen sein. Allgemeiner betrachtet ist dies nicht immer der Fall: Eine Auswertung wird bei Daten vorgenommen, die bereits existieren, und um darauf eine neue Sichtweise zu entwickeln. Doch die Visualisierung wird allgemein ebenso eingesetzt, um Neues zu modellieren und darzustellen bzw. zu veranschaulichen: Etwa Ideen, Konzepte, aber auch Konstruktionen, Anpassungen von Vorhandenem oder auch völlig neue Darstellungsarten und Sichtweisen auf vorhandene Objekte.
Es gibt noch weitere, speziellere Klassifikatoren von Visualisierungen (entnommen aus (Schumann und Müller, 2000, S. 175)):

- Dimensionalität des Darstellungsraumes
  zur Unterscheidung zwischen 2- und 3-dimensionalen Darstellungstechniken
- Zeitabhängigkeit der Darstellung
  zur Unterscheidung zwischen statischen und dynamischen Darstellungen
- Vollständigkeit
  zur Unterscheidung, ob in einem Ausgangsbild alle Daten auf einmal präsentiert werden [...]

2.2 Architekturvisualisierung

Im Folgenden soll zu einem Teilgebiet des allgemeinen Themas „Visualisierung“ übergegangen werden: der Architekturvisualisierung. Sie beschäftigt sich in erster Linie mit dem Entwurf und der Darstellung von architektonischen Gebilden wie Gebäuden, Plätzen oder Innenräumen, aber auch mit wesentlich abstrakteren Inhalten. Insgesamt ist der Begriff „Architektur“ relativ schwer zu erfassen und zu definieren. Die Architekturvisualisierung bedient sich unterschied-



Abbildung 2.2: Beispiel einer typischen Architekturvisualisierung[1].

licher Techniken sowohl in der Erzeugung als auch in der Darstellung. Sie ist in die in Kap. 2.1 gezeigten Klassen nur schwer einzusortieren: Zutreffend wäre - je nach Anwendungsfall mehr oder weniger - jede der Klassen.
Nachfolgend sei auf einige wenige Beispiele eingegangen. Zu beachten ist, dass diese nicht ausschließlich im Kontext der Architekturvisualisierung vorkommen.

Zeichnung

> „Zeichnungen geben in bildmäßigen oder technischen Kodierungen Auskunft über Denkvorstellungen, ...“ (Entnommen aus (Knauer, 2002, S. 7))

[1] Entnommen aus (Lengyel und Toulouse, Umbau eines Industriegebäudes)

Die Zeichnung ist gewissermaßen die „ursprüngliche“ Form des Entwerfens und Konstruierens. Der Erzeuger muss sich allerdings frühzeitig mit bestimmten Vorgaben und Zielen auseinander setzen (Vgl. Kap. 2.3), da die Flexibilität relativ gering ist. Jedoch ist die Zeichnung nach wie vor ein wichtiger Bestandteil der Visualisierung.

CAD

CAD ist ein allgemeiner Überbegriff für das Konstruieren, Modellieren, Entwerfen und Planen am Computer. Der Einsatz von Rechnern bietet dem Anwender ein breites Spektrum an flexiblen Werkzeugen, um Ideen umzusetzen, anzupassen und zu präsentieren. Insgesamt betrachtet ist CAD die konsequente Weiterentwicklung der Zeichnung, insbesondere wenn es um das flexible Eingreifen in den Entwicklungsprozess geht. Es gibt eine große Auswahl an Softwareprodukten, welche sich auf diesen Bereich konzentrieren, wobei alle verschiedene Vor- und Nachteile haben und sich oftmals auf ein bestimmtes Teilgebiet konzentrieren.

Visualisierung von Innenräumen

Die Architekturvisualisierung lässt sich in die Visualisierung Außen- und Innenräumen unterteilen. Die Visualisierung erfordert ähnliche Werkzeuge wie die von Außenräumen, doch es stehen andere Objekte bzw. Aspekte im Mittelpunkt, z. B. Möbel, Raumaufteilung, Beleuchtung usw.

2.3 Allgemeine Anforderungen an eine Visualisierung

Da es, wie bereits angesprochen, vielfältige Anwendungsgebiete für Visualisierungen gibt, ist es nicht einfach, allgemeine Anforderungen zu definieren. Um eine Visualisierung ihrem jeweils deklarierten Ziel nahe zu bringen, müssen dennoch bestimmte Anforderungen an diese Visualisierung gestellt und bestimmte Entscheidungen getroffen werden. Im Folgenden werden einige dieser Anforderungen bzw. Entscheidungen erläutert.

2.3.1 Grad der Interaktivität

Es gibt verschiedene Grade der Interaktivität, von denen im Vornherein bei einer Visualisierung einer festzulegen ist. Hierbei ist zu beachten, dass der Aufwand der Erzeugung meist mit dem Grad der Interaktivität steigt. Ebenso gilt: Je interaktiver, desto geringer die Bildqualität. Natürlich steigt die Rechenleistung heutiger Systeme stark über die Zeit hinweg an, allerdings ist der Unterschied zwischen einer vorberechneten Sequenz und einer Echtzeitsequenz teilweise noch recht hoch.
Darüber hinaus gilt es bei hoher Interaktivität festzulegen, welche Aktionen ein Anwender überhaupt tätigen kann. Hierbei kann man zwischen allgemeinen Aktionen, wie z. B. Kamera-Bewegungen, und objektspezifischen Aktionen unterscheiden. Es folgt eine Auflistung möglicher Abstufungen der Interaktivität:

- Interaktiv
  Notwendig: 3D-Engine, Verarbeitung von Nutzeraktionen
- Semi-interaktiv
  Feste Animation / festes Rendering, aber interaktiv steuerbar. Beispiel: 360-Grad-Fotografien
- Non-interaktiv
  Festes Rendering / Rendersequenz, Diagramme usw.

Auch hier gilt, dass die Übergänge keinesfalls hart sind: Es ist möglich, dass eine Visualisierung aus mehreren Teilen besteht, und jeder Teil hat seinen eigenen Grad der Interaktivität. Ebenso kann es sein, dass die verschiedenen Interaktivitätsgrade das gleiche Ausgangsmaterial haben. Ein 3D-Modell eines Gebäudes kann sowohl in einer non-interaktiven Rendersequenz als auch in einer interaktiven Präsentation verwendet werden. Oftmals sind nur geringe Anpassungen am Modell notwendig.

2.3.2 Wahl der Perspektive

> „Da wir überzeugt sind, dass der unmittelbare visuelle Eindruck ein essentieller Bestandteil eines jeden Projektes ist, betätigt sich unser eigener Planungsvorgang, in zunehmendem Maße, im Bereich der Perspektive“ (Entnommen aus (Lengyel, 2002, S. 3))

Insbesondere bei Visualisierungen mit einem geringem Grad an Interaktivität obliegt dem Erzeuger verstärkt die Wahl der Perspektive. Kann ein Anwender sich in einer dreidimensionalen Umgebung frei bewegen, ist diese Wahl auf den Anwender übertragen. Besteht die Visualisierung allerdings aus einem Bild oder einer Bildsequenz, sei diese gerendert oder gezeichnet, hat der Erzeuger die volle Kontrolle darüber, aus welchem Blickwinkel der spätere Betrachter das Objekt zu sehen bekommt.
Die Wahl der Perspektive ist einer der zentralen Punkte bei einer Visualisierung: Sie entscheidet in großem Maße, welche Aussage getroffen werden kann. Besonders an dieser Stelle hat die rechnergestützte Visualisierung (z. B. CAD) einen Vorteil: Die Wahl der Perspektive kann im Nachhinein getroffen werden. Zwar schränkt sich das ein, wenn z. B. ein Gebäude in vorhandene Fotografien integriert werden soll - ein gängiges Prinzip in der Architekturvisualisierung - aber theoretisch ist es ja möglich, die Fotografien bei einer Anpassung der Perspektive erneut zu machen. An dieser Stelle sei erneut auf (Knauer, 2002, S. 70 ff.) verwiesen.
Zu beachten ist allerdings, dass eine frühe Auswahl der Perspektive ein zielgerichtetes Modellieren und genauere Planungen ermöglicht. Je früher die Perspektive feststeht, desto mehr Zeit kann in die eigentliche Modellierung fließen.

2.3.3 Wahl des Realismus

Eine weitere entscheidende Frage ist die Wahl des Grades des Realismus. Der Begriff Realismus soll an dieser Stelle bedeuten, wie stark sich die Visualisierung an der realen Welt in verschiedenen Ausprägungen orientiert. Diese Ausprägungen können Form, Farbe und Oberfläche, Bemaßung, Detaillierungsgrad aber auch andere sein.
Bei der Entscheidung über den Realismus müssen viele Aspekte einbezogen werden, nachfolgend seien die wichtigsten erläutert:

- Technische Machbarkeit
  Diese zieht bei manchen Visualisierungen eine harte Grenze: Manche Ideen sind technisch nur schwer umzusetzen und als Visualisierung zu manifestieren.

- Zeitaufwand
  Dieser Punkt steht in direktem Zusammenhang mit dem Zu-

satzaufwand, der für einen hohen Grad an Realismus notwendig ist.

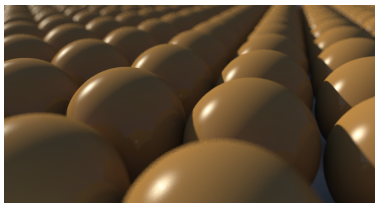
- Homogenität[1]
  Soll eine Visualisierung beispielsweise in reale Fotos integriert werden, ist oftmals ein sehr hoher Grad an Realismus gewünscht und notwendig, damit die entsprechende Wirkung - z. B. ein realistischer Blick auf das Aussehen eines Objektes in seiner zukünftigen Umgebung - erreicht werden kann. Hierbei müssen entsprechende Anpassungen durchgeführt werden, die zeitaufwändig und sehr komplex werden können. Dieses Problem haben Visualisierungen, die vollständig aus dem Rechner stammen, nur, wenn es als Anforderung definiert wurde. Es ist aber auch möglich, bewusst diese Homogenität zu durchbrechen und eine eher wirklichkeitsfremde Darstellung innerhalb einer realistischen Umgebung zu erzeugen.

2.3.4 Fokussierung des Betrachters

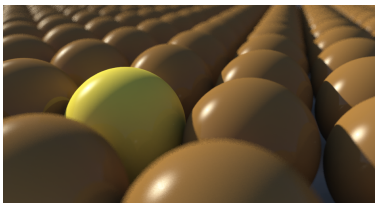
Um die Wirkung einer Visualisierung zu maximieren, muss oftmals der Fokus des Betrachters - und damit sein Interesse - auf bestimmte Elemente gelenkt werden. Bei non-interaktiven Visualisierungen, z. B. Renderings oder digitalen Kompositionen, kann man dies durch verschiedene Techniken erreichen. Beispiele sind Farben, Wahl des Bildausschnitts, Wahl der Perspektive usw.
Durch die hohe Einschränkung des Anwenders, nämlich auf das „blanke“ Betrachten, hat der Designer mehr Möglichkeiten, die Aufmerksamkeit des Betrachters gezielt auf bestimmte Elemente zu lenken.
Je mehr eine Visualisierung interaktiv ist, desto mehr muss der Designer auf andere Mittel setzen, um den Anwender zu fokussieren. Ein Beispiel wäre die automatische Drehung der Kamera auf ein bestimmtes Objekt, wenn der Anwender eine bestimmte Aktion durchführt. Eine andere Variante ist das so genannte Highlighting, welches bestimmte Objekte z. B. durch Färbung hervor hebt.

Abbildung 2.3 zeigt eine Gegenüberstellung zweier Darstellungen. Es ist deutlich zu erkennen, dass der Blick des Betrachters auf das

[1] Vgl. (Lengyel, 2002, S. 1)



(a) Objekte ohne Fokus



(b) Objekte mit Fokus durch Einfärben

Abbildung 2.3: Visualisierungen mit und ohne Fokussierung durch Färbung

eingefärbte Objekt in Abb. 2.3b gezogen wird, während dies in Abb. 2.3a nicht der Fall ist.

2.4 Fazit

> „One Picture is Worth Ten Thousand Words“[1] - Fred R. Barnard

Abschließend betrachtet ist festzustellen, dass die Visualisierung, in welcher Form auch immer sie zum Einsatz kommt, ein wichtiges und sinnvolles Werkzeug zur Entwicklung, Vermittlung und Darstellung von Vorgängen, Konzepten und Ideen ist: Sei es im Bereich der Architektur, Wissenschaft oder Technik. In Anbetracht der Denkweise des Menschen ist sie eine direkte Form der Kommunikation und kann Informationen hoher Dichte auf kleinem Raum unterbringen. Durch Interaktivität kann der Informationsgehalt - ohne den Betrachter dabei zu überfordern - weiter gesteigert werden.

[1] „Ein Bild sagt mehr als zehntausend Worte“

3 Vorbetrachtung / Anforderungsanalyse

Dieses Kapitel analysiert die Anforderungen und ordnet diese in ein Gesamtkonzept ein. Dies ist notwendig, um anschließend das System an sich zu modellieren und verschiedene Funktionalitäten ablesen zu können. Ergebnis ist die Einteilung des Systems in verschiedene Module, die die Gesamtfunktionalität abdecken. Darüber hinaus müssen einige grundlegende Begriffe erläutert werden.
Grundlage bildet die definierte Anforderung der Aufgabenstellung im Kapitel C.1 auf Seite 194, da darüber ein Rahmen für die Funktionalität geschaffen wird.
Neben der Funktionalität setzt sich dieses Kapitel mit vorhandenen Systemen, die als Grundlage für die Implementierung dienten, auseinander. Ebenso wird ein kurzer Blick auf nicht verwendete Systeme geworfen, da diese gleichwertig zu der letztendlich eingesetzten Lösung betrachtet wurden.

3.1 Grundlegende Anforderungen

Zunächst sind einige Anforderungen an das Gesamtsystem als fundamental zu betrachten:

- Darstellung beliebiger dreidimensionaler Objekte in einem Echtzeitfenster
  Hierbei ist die darunter liegende Darstellungstechnik (z. B. OpenGL oder DirectX[1]) unwichtig. Entscheidend ist, dass die Darstellungsqualität gewisse Möglichkeiten bietet. Dazu gehört, dass Texturen in Echtzeit dargestellt werden, da auf die erzeugten Innenwände Texturen projiziert werden sollen.

[1] DirectX ist eine Grafikbibliothek der Firma Microsoft.

- Veränderung dieser Objekte
  Dieses Verändern bedeutet nicht, dass ein Objekt direkt über polygonale Werkzeuge modelliert bzw. angepasst werden muss. Als möglicher Dateninput können Modelle anderer Softwarepakete[1], die sich auf Modellierung spezialisiert haben, verwendet werden.

- Präsentation dieser Objekte
  Zur Präsententation der erzeugten Szene kann die selbe Plattform eingesetzt werden, allerdings muss die GUI entsprechend angepasst werden.

Mögliche Objekte können hierbei sein:

- Innenwände
  Ein Raum besteht aus mindestens drei Innenwänden, welche durch den Designer platziert und gestaltet werden.

- Möbelstücke
  Möbelstücke können z. B. aus anderen Modellierprogrammen importiert und anschließend platziert werden.

- Gemälde
  Ein Gemälde ist - als eine stark vereinfachte Darstellung - eine Fläche mit einer Textur, welche an einer Innenwand platziert werden kann.

- beliebige 3D-Objekte
  Über Möbelstücke oder Gemälde hinaus können natürlich auch andere 3D-Objekte importiert und platziert werden.

Die ersten drei Objekte unterliegen den gleichen Eigenschaften wie allen anderen, beliebigen 3D-Objekte. Der Unterschied besteht in der Klassifikation: Ein Möbelstück kann als ein eben solches klassifiziert sein. Doch aus struktureller Sicht unterscheiden sich diese Objekte nicht von anderen 3D-Objekten.
Die Erzeugung eines Raumes beinhaltet folgende Elemente:

- Ein Grundriss bildet die Grundlage.
  Dieser kann als Bilddatei importiert und im Hintergrund des Editors angezeigt werden, damit die Innenwände entsprechend des Grundrisses platziert werden können.

[1] Z.B. Wings3d, zu finden unter http://www.wings3d.com

- Die simple Erzeugung der Räume, also die Platzierung der Innenwände eines oder mehrerer Räume.
  Hierbei gibt es vielfältige Möglichkeiten, diese Innenwände zu erzeugen und zu editieren.

3.1.1 Grundmodule des Gesamtsystems

Es gibt laut Aufgabenstellung eine Trennung zwischen zwei Kernfunktionalitäten: dem Erzeugen der Präsentation und ihrer Durchführung. Es liegt deshalb nahe, das gesamte System in zwei Untersysteme (Editor und Präsentation) zu gliedern. Diese Untergliederung hat den Zweck, den Editor, also die Funktionalität für den Designer, und den Präsentationsmodus, gedacht für den anschließenden Betrachter, sowohl logisch als auch funktional zu trennen. Das heißt, dass beide Teilsysteme auf die selbe Datenbasis, nämlich die erzeugte Szene zugreifen. Der Editor gestattet hierbei andere Möglichkeiten als der Präsentationsmodus.
Diese Trennung bedeutet, dass theoretisch zwei unterschiedliche Softwaresysteme entstehen und auch getrennt ausführbar sein können. Doch eine harte Trennung beider Systeme liegt nicht in der Anforderung: Ein flexibles „Umschalten“ zur Laufzeit ist durchaus denkbar.
Es ist also festzuhalten, dass es zwei Teilsysteme[1] gibt: Editormodus und Präsentationmodus. Der wichtigste Unterschied liegt darin, dass der Präsentationsmodus keine Daten verändern sollte, sondern nur vorhandene Daten auswertet und auf Benutzereingaben reagiert.
Begründet ist diese Trennung darin, dass der Präsentationsmodus den Betrachter von jeglichen Editorfunktionen ausschließen kann bzw. soll. Die Präsentation bietet nur durch den Erzeuger erlaubte Möglichkeiten, ohne die Szene an sich zu verändern. Dies ist üblich, wenn die Präsentation „unmoderiert“ abläuft, d. h. ohne dass der Erzeuger die Präsentation selbst durchführt, sondern der Betrachter die Präsentation steuert. Abbildung 3.1 zeigt eine grobgranulare Aufschlüsselung des Systems „Editor“. Es sind die verschiedenen Mo-

[1] Der Begriff „Teilsystem“ muss grundlegend nicht vollständig treffend sein. Effektiv kann ein System zwei Modi besitzen, die sich im Verhalten so unterscheiden, als wären es zwei unterschiedliche Systeme. Hierbei ist die Funktionalität unterteilt in Teilsysteme, aber diese bauen auf die gleiche programmatische Grundlage auf. Somit bezieht sich „Teilsystem“ nur auf das durch den Nutzer sichtbare Verhalten.

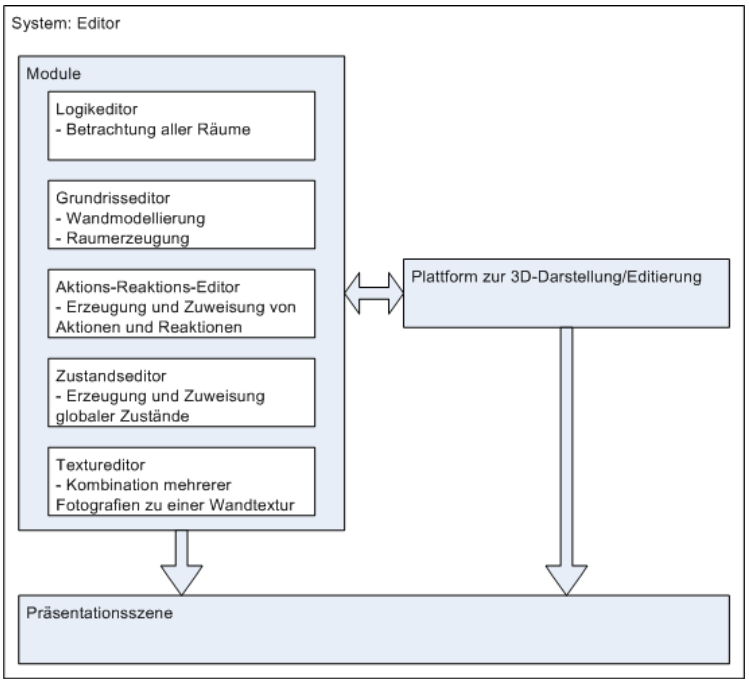


Abbildung 3.1: Fachliches Modell des Systems „Editor“

dule aufgelistet, darüber hinaus ist die Interaktion mit der Plattform (siehe Abschnitt 3.2) dargestellt. Diese Abbildung orientiert sich eher an den fachlichen Anforderungen als an der späteren Umsetzung.
Neben den Modulen und der Plattform ist die darunter liegende Datenschicht dargestellt. Hierbei sei beachtet: Diese Schicht kann durchaus Bestandteil der Plattform selbst sein. Diese Darstellung soll allerdings vermitteln, dass es solch eine Schicht geben muss, was auch üblicherweise der Fall ist.
Innerhalb dieser Schicht, welche im Regelfall ein Szenegraph ist, wird die durch den Editor erzeugte Szene gespeichert und kann während der Präsentation abgerufen und ausgewertet werden.

Das System „Präsentation“ ist in Abbildung 3.2 aufgeführt. Es ist zu sehen, dass die im Editor wichtigen Module keine Rolle zur Auswertung der Szene spielen. Der Teil „Aktions-Reaktions-Auswertung“ bezieht sich darauf, dass Interaktionen des Benutzers mit der Plattform über dieses Modul ausgewertet und entsprechend zuvor definierte Reaktionen ausgeführt werden. Es wird auf die selbe Datenbasis, nämlich die erzeugte Szene, zugegriffen, ohne diese zu verändern.

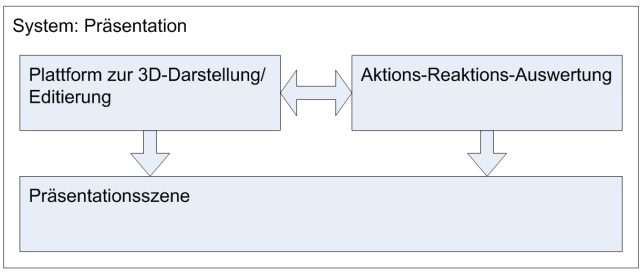


Abbildung 3.2: Fachliches Modell des Systems „Präsentation“

3.2 Wahl der Plattform

Eine der grundlegendensten Anforderungen ist, dass das Gesamtsystem aus interaktiven 2D- und 3D-Elementen besteht. Es steht fest, dass im Rahmen dieser Arbeit das Aufbauen auf vorhandenen Lösungen zwingend notwendig ist, der Grund ist schlicht die verfügbare Zeit. Um erweiterte Konzepte zur Modellierung und Präsentation zu entwerfen und umzusetzen, darf es nicht notwendig sein, zunächst viel Zeit in allzu grundlegende Funktionen zu investieren. Es zeigte sich, dass bereits geeignete Lösungen existieren.
Gesucht ist also eine Plattform, welche einigen grundlegenden Anforderungen genügt, eventuell einige erweiterte Konzepte beinhaltet und die aufsetzende Implementierung bestmöglich unterstützt. Über die Wahl der Plattform werden implizit weitere Entscheidungen getroffen:

- Programmiersprache
  Die Erweiterung einer Plattform ist zumeist in der Programmiersprache umzusetzen, in der die Plattform selbst umgesetzt wurde. Dies ist nicht zwingend so, aber durchaus die Regel.

- Aufwand in der Implementierung
  Verwendet die Plattform eine eher unbekannte Programmiersprache, ist es schwieriger, freie Bibliotheken zu finden und einzusetzen. Darüber hinaus sind manche Probleme in einer Programmiersprache leichter als in einer anderen zu lösen.

- Zu implementierende Grundfunktionalität
  Bietet die Plattform eine bestimmte Funktionalität nicht, welche von vielen Modulen gebraucht wird, muss diese nachimplementiert werden. Das ist nicht wünschenswert, da relativ

viel Zeit in die Implementierung solcher Grundkonzepte investiert werden muss und eine Konzentration auf das eigentliche Thema schwieriger wird.

- Erweiterungsmöglichkeit
  Grundsätzlich kann in zwei Modelle der Erweiterung unterschieden werden: Entweder man erweitert die Plattform direkt, indem der Quellcode angepasst wird, oder man hat die Möglichkeit, über Plugins[1] zu erweitern. Bei letzteren muss die Plattform an sich nicht angepasst werden. Ist die Plugin-Architektur flexibel genug, kann durchaus die gesamte Erweiterung als ein Plugin oder verschiedene Plugins realisiert werden. In einigen Fällen kann auch beides notwendig werden: Zum einen basiert die Erweiterung ausschließlich auf zusätzlichen Plugins, zum anderen werden durch Anpassung des Quellcodes zusätzliche Funktionalitäten direkt in die Plattform eingefügt.

- Komplexität
  Hierbei ist die Gesamtmenge der vorhandenen Funktionalität gemeint und wie sie bei der Erweiterung angesteuert wird. Dies ist davon abhängig, wie die Schnittstellen und Konzepte der Plattform umgesetzt sind.

- Bedienung
  Bietet die Plattform bereits interaktive Möglichkeiten, stellt sich die Frage: Wie gut ist die Bedienung? Lässt sich diese anpassen bzw. erweitern? Ist die GUI interaktiv anpassbar? Setzt die GUI auf bekannte Bibliotheken, z. B. Swing[2] oder Qt[3] auf? Indirekt hat dies auch Auswirkungen auf den zu investierenden Programmieraufwand: Sollte die GUI schwierig anzupassen bzw. zu erweitern sein, kann dies erhebliche Zusatzaufwände in der Implementierung bedeuten.

Es gibt bereits eine große Anzahl fertiger Lösungen auf dem Markt, welche bereits ähnliche Anforderungen wie die hier definierten

[1] Ein Plugin bietet die Möglichkeit, ein Programm in seiner Funktionalität zu erweitern, ohne den Quellcode des Hauptprogrammes erweitern zu müssen.

[2] Swing ist eine auf AWT aufbauende Klassenbibliothek, welche die Implementierung von grafischen Benutzeroberflächen stark vereinfacht. Vgl. hierzu (Horstmann und Cornell, 2008a, s. 282 ff.).

[3] Qt ist eine GUI-Bibliothek für C++ , siehe Nokia Cooperation (2008–2009).

erfüllen. Doch sind diese Lösungen oftmals kommerzieller Art und somit nicht quelloffen. Da für diese Arbeit jedoch einige unterschiedliche Anforderungen gestellt wurden, welche diese Lösungen vermutlich nicht vollständig enthalten, war zu prüfen, inwiefern eine Kombination verschiedener Lösungen - seien es vollständige Produkte oder Bibliotheken, quelloffen oder kommerziell - zum Ziel führen würden. Ein Hindernis stellt hierbei eine eventuell nicht vorhandene Quelloffenheit dar: Es kann für einige Module zutreffen, dass die Funktionalität nicht über eine Plugin-Architektur hinzugefügt werden kann, wenn dafür keine Möglichkeiten seitens der API[1] vorgesehen sind. Somit droht die Gefahr, dass zu einem bestimmten Zeitpunkt der Entwicklung eine Grenze des Machbaren erreicht wird, die nicht ohne weiteres zu überwinden ist. Nur quelloffene Software bietet die Möglichkeit, direkt am Kern der Software Veränderungen durchzuführen.
Nachfolgend sei ein kurzer Überblick gegeben, welche Plattformen und Bibliotheken betrachtet wurden:

- Xj3D
  ist eine in Java geschriebene, quelloffene X3D-Bibliothek mit direkter Anbindung an verschiedene Echtzeitrenderer (z. B. OpenGL).

- BSContact
  ist ein kommerzieller X3D-Browser der Firma Bitmanagement. Der Browser ist in C++ geschrieben und bietet vielfältige Erweiterungsmöglichkeiten, welche sich allerdings auf die Präsentation von 3D-Daten, weniger auf deren Erzeugung konzentrieren.

- Lightfeather[2] + Qt + Boost[3]
  bilden als 3D-Engine mit den Bibliotheken Qt und Boost - welche alle quelloffen sind - eine mögliche Grundlage eines vollständigen Produktes. Jedoch war der hohe Programmieraufwand nicht vertretbar, da jegliche Interaktivität zunächst neu implementiert werden musste.

Als geeignete Quelle zum Auffinden und Vergleichen verschiedener Plattformen erwies sich DevMaster.net (2003–2009) als wertvoll.

[1] Application Programming Interface - Programmierschnittstelle einer Anwendung
[2] Siehe Meyer.
[3] Siehe Dawes u. a. (2008).

Neben diesen Varianten bot sich an, die am Lehrstuhl „Grafische Systeme“ der BTU Cottbus entwickelte Plattform GroIMP zu nutzen. Das folgende Kapitel beleuchtet diese Plattform, worauf schlussendlich die Wahl fiel, genauer.
Für die Umsetzung der geforderten 2D-Funktionalität war eine Suche nach quelloffenen Plattformen, welche von vornherein bestimmte Erleichterungen boten, erfolglos. Die meisten Plattformen, wie z. B. ImageJ[1], entsprachen nicht den Anforderungen[2], basieren ausschließlich auf Pixeln, während einige Anforderungen ein übergeordnetes Koordinatensystem brauchen. Dem entsprechend wurden die in Java enthaltenen Bibliotheken AWT[3] und Swing eingesetzt, welche bestimmte Grundlagen boten.
Als Resultat der Analyse ergab sich auch die Wahl der verwendeten Werkzeuge zur Umsetzung: Eclipse 3.3 als IDE und Java 5 als Programmiersprache.

3.3 Funktionale und Nichtfunktionale Anforderungen

Dieser Abschnitt behandelt die zugrunde liegende Aufgabenstellung, welche die wichtigsten funktionalen Anforderungen absteckt. Deshalb wird auf die funktionalen Anforderungen nicht weiter eingegangen.
Andererseits haben die nichtfunktionalen Anforderungen, welche in der Aufgabenstellung ebenso implizit verankert sind, einen hohen Anspruch, weshalb an dieser Stelle noch einmal darauf eingegangen wird. Die wichtigsten Anforderungen an das System sind:

- Flexiblität
  Dieser Begriff ist recht vielseitig zu verstehen: Er soll sich an dieser Stelle auf die Wiederverwendbarkeit der Module beziehen. Das bedeutet, dass sich z. B. die implementierte 2D-Funktionalität für andere Aufgaben anpassen und wiederverwenden lässt.

[1] Siehe Burger und Burge (2006).
[2] Kapitel 5 erläutert die Notwendigkeiten eines nicht auf Pixel basierendes 2D-System als Grundlage genauer.
[3] Advanced Windowing Toolkit

- Erweiterbarkeit für spätere Aufgaben
  Sollte eine Erweiterung des Systems auf neue Aufgaben durchgeführt werden, muss es so gestaltet sein, dass solche Erweiterungen problemlos möglich sind.

- Einfache Workflows
  Darin liegt der höchste Anspruch. Ein einfacher Workflow ist schwer zu planen, da sich oft erst nach der Umsetzung zeigt, dass einige Teile eines Workflows zu schwierig, zu ungenau oder schlicht zu lang sind. Das bedeutet, dass nach einer Implementierung durchaus einige Veränderungen möglich sein müssen, falls Anpassungen gefordert werden.

Die resultierenden Implementierungen haben das Ziel, möglichst alle funktionalen und nichtfunktionalen Anforderungen umzusetzen.

4 GroIMP als Plattform



Abbildung 4.1: Mit GroIMP erzeugter Baum (Erstellung: S. Rogge, entnommen aus Kurth u. a. (2008))

GroIMP ist eine vom Lehrstuhl Grafische Systeme der BTU Cottbus in Java entwickelte Plattform zur dreidimensionalen Visualisierung. In erster Linie ist GroIMP als Visualisierungssoftware zur regelbasierten Programmiersprache XL[1] konzipiert worden. Kennzeichen sind die offene Struktur des Szenegraphen, welche beliebig erweitert werden kann, und die Möglichkeit der Erweiterung durch Plugins. Darüber hinaus ist die Benutzeroberfläche anpassbar und umfasst frei definierbare Layouts. Die Visualisierung ist durch OpenGL, aber auch durch verschiedene andere, nicht auf Echtzeit basierende Render-Methoden (z. B. Raytracing) möglich. Benutzerinteraktionen wie die Transformation der Kamera oder andere Benutzereingaben sind frei zugänglich und können an die Anforderungen angepasst werden.
Da der ursprüngliche Einsatzbereich von GroIMP außerhalb üblicher Modellierpraktiken, wie z. B. polygonaler Modellierung liegt, war es als Herausforderung zu betrachten, GroIMP um sinnvolle Werkzeuge zu erweitern. Die Modellierung von 3D-Objekten wird in GroIMP in erster Linie durch die Sprache XL zur Verfügung gestellt, jedoch ist dieser regelbasierte Ansatz nur eingeschränkt zur Modellierung von Innenräumen geeignet. Natürlich wäre dies durch die

[1] Auf die Grundlagen der Programmiersprache XL wird an dieser Stelle nicht genauer eingegangen. Für weitere Informationen wird Kniemeyer (2008) und Kurth u. a. (2008) empfohlen.

Formulierung von XL-Regeln möglich, doch dies ist nicht sonderlich intuitiv.
GroIMP bietet ein System zur Verwendung verschiedener Shader, welche direkt im Echtzeit-Fenster über OpenGL darstellbar sind. Dies stellt auch die Eignung von GroIMP als Plattform für die Anforderung der Darstellung (nicht nur der Modellierung) von Innenräumen dar. Die Fähigkeit, beliebige Texuren auf polygonale Oberflächen zu projizieren und in Echtzeit darzustellen ist als Grundanforderung an die Plattform selbst - und damit an GroIMP - zu verstehen.
Die folgenden Kapitel sollen einen Überblick über die Plattform GroIMP bieten. Es gibt eine Unterteilung in zwei Abschnitte, die diesen Überblick aus Anwendersicht (Abschnitt 4.1) und Entwicklersicht (Abschnitt 4.1) bieten sollen. Die Entwicklersicht hat eine große Bedeutung, da die Interna von GroIMP größtenteils undokumentiert sind und somit eine Hilfestellung für zukünftige GroIMP-Entwickler gegeben werden soll. Ebenso begründen und erläutern diese Kapitel einige später durchgeführte Implementierungen im Rahmen von *ArchiViz*. Die Anwendersicht soll darüber hinaus ebenso XL einbeziehen, da GroIMP seine Stärken in der Modellierung in erster Linie über XL ausspielt.

4.1 Modellierung mit GroIMP und XL

Dieser Abschnitt befasst sich mit der Plattform GroIMP aus der Sicht eines Anwenders. Er soll aufzeigen, wie GroIMP am besten einzusetzen und in Bezug auf XL in kreative Verfahrensweisen einbezogen werden kann. Es soll ein Überblick über die wichtigsten Funktionen und die Anbindung an XL geschaffen werden.

4.1.1 Wichtige GUI-Elemente

In GroIMP können mehrere verschiedene GUI-Layouts definiert werden. Jede Definition macht bestimmte GUI-Elemente sichtbar und blendet andere dafür aus. Es gibt bereits mehrere definierte GUI-Layouts, welche für die wichtigsten Aufgaben zusammengestellt wurden. An dieser Stelle sei eines davon gezeigt, welches für die Modellierung mit XL am wichtigsten ist. Zur Aktivierung muss in Hauptmenü „Panels“, „Set Layout“ und anschließend „RGG Lay-

out“ ausgewählt werden. Typischerweise ist dieses Layout standardmäßig eingestellt.

In Abbildung 4.2 wird dieses Layout gezeigt, die wichtigsten Ele-

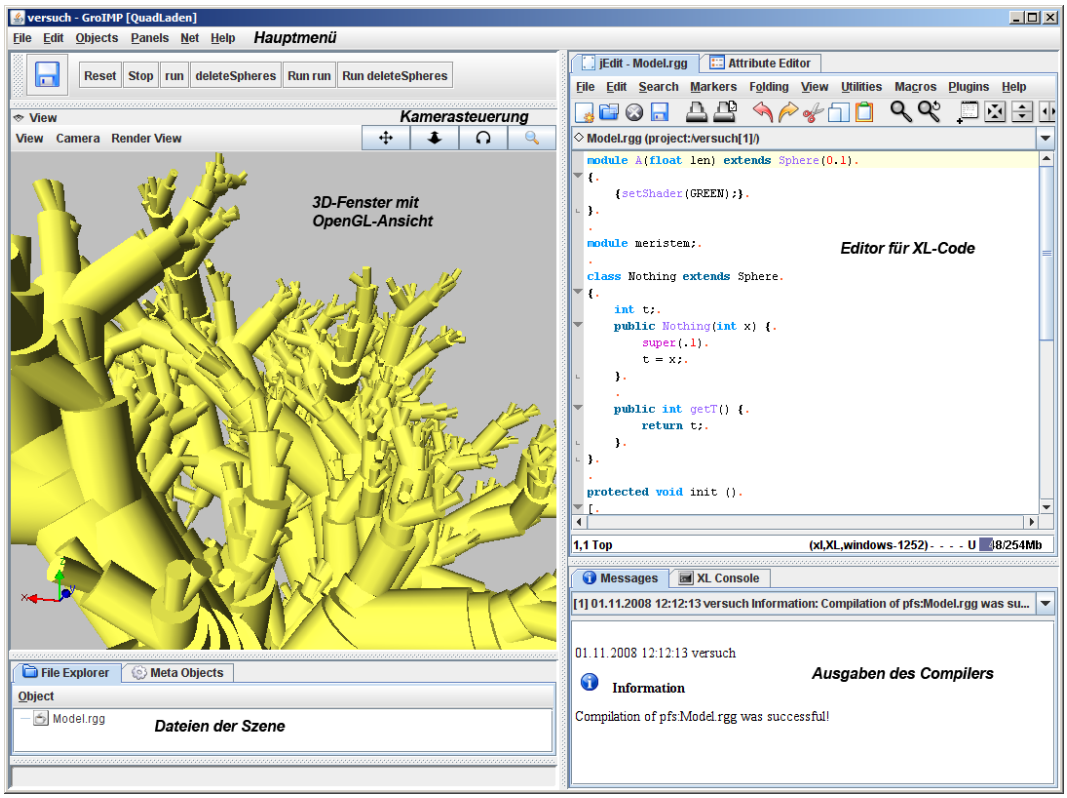


Abbildung 4.2: Das Layout „RGG“ in GroIMP

mente sind beschriftet. Nachfolgend sind diese Elemente genauer erläutert:

- Hauptmenü
  Enthält die wichtigsten Funktionen GroIMPs, z. B. zum Umschalten zwischen verschiedenen Layouts oder zum Laden und Speichern von Dateien. Ein wichtiges Untermenü sind die Voreinstellungen (preferences), zu finden im Menü „Panels“.

- 3D-Fenster
  Stellt den Blick durch die virtuelle Kamera auf die 3D-Szene dar. In der Abbildung 4.2 ist OpenGL als Echtzeit-Renderer aktiviert.

- Kamerasteuerung
  Um die Kameraposition bzw. den Blickwinkel anzupassen, bieten diese Buttons die entsprechenden Funktionen.

- Dateien der Szene
  Der Fileexplorer zeigt alle angelegten XL-Code-Dateien an (*.rgg), aber auch andere importierte bzw. verlinkte Dateien, z. B. Bilder.

- Editor für XL-Code
  Es gibt zwei verschiedene Editoren: Zum einen den in GroIMP integrierten Editor, zum anderen den an GroIMP angebundenen freien Editor JEdit. Letzterer bietet komfortable Möglichkeiten zum Editieren von XL-Code. Ein Umschalten zwischen beiden Editoren ist über die Voreinstellungen (preferences) möglich.

- Ausgaben des Compilers
  Im Fenster „messages“ werden eventuelle Kompilierfehler oder andere Ausgaben angezeigt.

Zu beachten ist, dass dies nur einen kleinen Ausschnitt aller möglichen GUI-Elemente darstellt. Da GroIMP über Plugins erweiterbar ist, kann je nach Version die Verfügbarkeit bestimmter GUI-Elemente stark variieren.

4.1.2 3D-Modellierung über XL

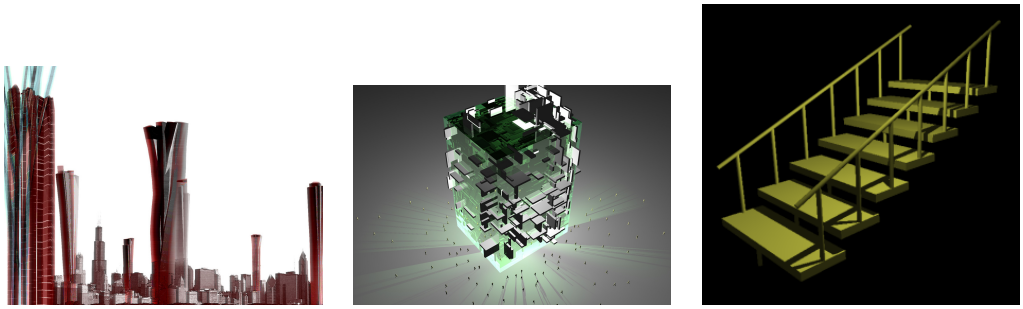
Wie bereits erwähnt, beinhaltet GroIMP selbst keine üblichen Modelliermechanismen für polygonale Objekte. Es fehlen also Funktionen zur manuellen Erzeugung oder Editierung vorhandener polygonaler Objekte. Für NURBS-Objekte[1] verhält es sich ebenso. Jedoch ist die Funktionalität durch die Anbindung an XL keineswegs für kreative Prozesse eingeschränkt: Einzig die Herangehensweise ist für einen Anwender, der die üblichen Mechanismen gewohnt ist, ungewöhnlich. Natürlich lassen sich nicht alle Techniken über XL umsetzen, aber die regelbasierte Generierung von Geometrie ist der „manuellen Erzeugung“ in einigen Aspekten und Anwendungsfällen überlegen.
Insbesondere wenn regelmäßige Strukturen modelliert werden sollen, ist eine Abstraktion über XL mit anschließender Generierung

[1] Non Uniform Rational B-Splines (NURBS), Vgl. (Nischwitz u. a., 2007, S. 52 ff.).

durch Abarbeitung der Regeln ein wertvoller Mechanismus. Da die Möglichkeit besteht, über XL theoretisch vollständig auf den Szenegraphen GroIMPs zuzugreifen und diesen zu verändern (d.h. alle Knoten, Kanten und Attribute von Knoten), bieten sich komplexe Techniken zur Erzeugung von Geometrie.
Die daraus resultierenden Möglichkeiten sind vielfältig. Da insbesondere die Modellierung von Pflanzen als ein Kernbereich der Anwendung von XL und deren Visualisierung in GroIMP vorgesehen ist, ist dieser Bereich natürlich ein bevorzugter Anwendungsbereich der Modellierung mit XL. Der regelbasierte Ansatz von XL kann aber durchaus seine Anwendung in anderen Fachbereichen finden.

4.1.3 Anwendungen von XL und GroIMP

Im Wintersemester 2006/2007 fand erstmals das Seminar „Artifizielle Wachstumsprozesse"(artificial growth processes) für Architekturstudenten an der BTU Cottbus statt. Hierbei wurden die Möglichkeiten, XL und GroIMP in der Architektur einzusetzen, das erste Mal tiefgehender erkundet. Es war weniger Ziel, Gebäude zu entwerfen, sondern eher, Algorithmen in XL zu entwickeln, welche Gebäude erzeugen[1]. Die Ergebnisse zeigen, dass XL und GroIMP



(a) (b) (c)

Abbildung 4.3: Eine Auswahl verschiedener Ergebnisse des Seminars „Artifizielle Wachstumsprozesse“: 4.3a durch Christopher Jarchow, 4.3b durch Liang Liang und 4.3c durch Manuela Fritzsche. 4.3a und 4.3b wurden außerhalb von GroIMP nachbearbeitet.

einiges an Potential zur Nutzung zur Visualisierung in der Architektur besitzen. Dieses Potential kann durch vermehrte Anwendung

[1]Nach (Kniemeyer, 2008, S. 341)

noch erweitert werden. Der algorithmische Ansatz zur Erzeugung von Strukturen und Geometrie kann als sinnvolle Ergänzung zu vorhandenen Mechanismen eingesetzt werden.

4.1.4 Einbindung von XL und GroIMP in eine Produktionskette

Der Begriff „Produktionskette“ bedeutet an dieser Stelle das Zusammenspiel verschiedener Softwareprodukte, um ein bestimmtes Ziel zu erreichen. Hierbei bauen verschiedene Softwareprodukte aufeinander auf, dem entsprechend müssen die Schnittstellen der verschiedenen Produkte und deren Position in der Kette berücksichtigt werden. Um die Vorteile GroIMPs und die der Programmiersprache XL voll nutzen zu können, ist eine Integration in eine vorhandene Produktionskette für einige Anwendungsbereiche notwendig. Hierzu gehört unter anderem die Architektur: Nur selten kann ein Projekt, z. B. eine Visualisierung eines Gebäudes, in nur einem Softwareprodukt realisiert werden. Da GroIMP und XL recht spezielle Arbeitsweisen besitzen, sind diese eher ungeeignet, um eine solche komplexe Visualisierung allein zu übernehmen. Vorteilhaft ist aber, wenn der Einsatz von GroIMP und XL in einem bestimmten Teilbereich der gesamten Kette konzentriert wird.
Ein einfaches Beispiel sei der Entwurf eines Gebäudes, dessen Grundriss vorliegt. Dieses Gebäude soll in 3D visualisiert werden. Die Stärke von GroIMP liegt, wie bereits angesprochen, nicht in der manuellen und exakten Modellierung dreidimensionaler Geometrien: Es wäre nicht besonders intuitiv und wahrscheinlich auch nicht zeiteffizient, das Gebäude komplett über XL zu modellieren. Es sei nicht behauptet, dass das grundsätzlich nicht möglich ist, aber ob dies sinnvoll ist, sei zu bezweifeln. Sinnvoller ist es, zunächst ein Programm zu nutzen, das sich auf solche Aufgaben spezialisiert hat. Dazu gehört z. B. ArchiCAD[1], ein Modellier- und Konstruktionsprogramm, das exakte bauteilorientierte Modellierung[2] ermöglicht. Erfahrene Architekten können mit Programmen wie diesen sehr effizient Entwürfe für Visualisierungen in 3D umsetzen.
Nun haben allerdings die meisten Programme einige Schwachstel-

[1] Siehe Graphisoft Deutschland GmbH (2009)

[2] Bauteilorientierte Modellierung bedeutet, dass die kleinsten Elemente der Szene nicht zwingend Polygone oder Nurbs-Kurven sind, sondern Bauteile, wie sie im Hausbau sinnvoll sind, z. B. Wände, Fensterrahmen usw.

len. Eine ist z. B. die Texturierung und der mangelnde Umgang mit Shadern. Das hat zur Folge, dass für eine exakte Visualisierung[1] oftmals die Mittel fehlen, um allein in diesem Programm die gesamte Visualisierung umzusetzen. Eine weitere häufige Schwachstelle ist der Umgang mit Pflanzenobjekten. Oftmals soll ein modelliertes Gebäude in seiner zukünftigen, natürlich wirkenden Umgebung dargestellt werden. Diese Umgebung besteht nicht nur aus anderen Gebäuden oder gepflasterten Plätzen, sondern auch aus Bäumen, Sträuchern oder ähnlichem.
Nun gibt es zwei Möglichkeiten: Entweder greift der Architekt auf vorhandene Pflanzenbibliotheken zurück, oder es müssen die Pflanzenobjekte manuell erzeugt werden. Der Vorteil der ersten Lösung liegt in der Zeiteffizienz: Es müssen nur vorhandene Objekte kopiert werden. Dies passiert jedoch auf Kosten der Flexibilität. Ein einfaches Beispiel ist, dass die vorhandenen Bibliotheksobjekte auf eine bestimmte Jahreszeit ausgelegt sind, doch die Visualisierung soll eine andere darstellen[2]. Eine Veränderung der Bibliotheksobjekte ist oftmals zeitintensiv auf Grund mangelnder Parametrisierung und bringt auch nicht immer die gewünschten Ergebnisse.
Die zweite Lösung bringt XL und GroIMP ins Spiel: XL eignet sich hervorragend, um Pflanzenobjekte parametrisiert zu modellieren bzw. zu generieren. Hierbei ist die Flexibilität sehr hoch. Zunächst wird zwar etwas mehr Zeit zur Erzeugung benötigt, denn der entsprechende Code muss zunächst geschrieben werden. Doch darüber hinaus können bestimmte Anpassungen schnell umgesetzt werden. Für den häufigen Einsatz von XL und GroIMP in diesem Kontext ist es erstrebenswert, eine kleine Datenbank an Beispielen aufzubauen, auf die zurück gegriffen werden kann. Die Problematik „Pflanzen in einer Visualisierung“ wiederholt sich oft und kann somit zunehmend zeiteffizient gelöst werden.
Die Einbettung GroIMPs und somit auch von XL in eine Produktionskette kann verschiedenartig stattfinden. Dies ist in erster Linie von der Problemstellung abhängig. Was aber durchgängig eine Rolle spielt, ist die Schnittstelle, über die Daten mit anderen Programmen ausgetauscht werden können.

[1] Der Begriff „exakt“ bezieht sich auf die Möglichkeit, 3D-Objekte nach Maß zu erzeugen.
[2] Das hat den Hintergrund, dass in eine Visualisierung mit hohem Anspruch an die visuelle Qualität oftmals reale Fotos integriert werden. Diese werden zu einem bestimmten Zeitpunkt im Jahr angefertigt, doch die Produktion der Visualisierung kann um Monate verschoben stattfinden.

- Export von Objekten
  GroIMP bietet zwei essentielle Datenformate zum Export an: *DXF*[1] und *X3D*. Beide Formate sind geeignet, um 3D-Daten an andere Programme weiter zu geben. Durch GroIMP bzw. XL erzeugte Objekte können somit exportiert und durch andere Programme genutzt werden.

- Import von Objekten
  Um vorhandene Objekte in GroIMP zu importieren, bieten sich die Formate *DXF*, *X3D* und *OBJ*[2] an.

- Rendering und Export von Bildern
  In GroIMP stehen verschiedene Renderer zur Verfügung: Einer davon ist der Raytracer „Twilight“. Es gibt aber auch eine Anbindung an PovRay. Die damit erzeugten Bilder können im weiteren Produktionsprozess eingesetzt werden.

Ein beispielhaftes Resultat einer solchen Produktionskette zeigt Abbildung 4.4. Die zu sehende Geometrie wurde vollständig in GroIMP über XL erzeugt und anschließend nach Autodesk Maya über das Format *DXF* exportiert[3]. Das Bild wurde zuletzt über den Renderer „Maxwell Render“, der eine direkte Anbindung an Maya besitzt, erzeugt.

4.2 Technische Hintergründe

Nachfolgend werden einige technische Interna GroIMPs näher erläutert, um zukünftigen GroIMP-Entwicklern einen Einstiegspunkt zu geben und um einige Erfahrungen mit GroIMP zu dokumentieren. GroIMP ist ein komplexes Gebilde, welches leider jedoch recht schwach dokumentiert ist. Die meisten der hier genannten Punkte mussten einzeln recherchiert oder schlicht durch Ausprobieren ergründet werden. Die Abschnitte 4.2 bis 4.7 können durch Leser, welche sich auf die Anwendungen in der Architektur beschränken wollen, durchaus übersprungen werden.

[1] *DXF* ist ein Austauschformat für CAD-Daten der Firma Autodesk.

[2] *OBJ* ist ein weit verbreitetes Format der Firma Autodesk zum Austausch von Geometriedaten polygonaler Objekte.

[3] Leider bietet Maya keinen direkten Import von *DXF*. Somit muss die exportierte *DXF*-Datei zunächst über ein Konvertierungsprogramm z. B. in *OBJ* überführt werden.



Abbildung 4.4: Über XL generierte Geometrie importiert in Maya und gerendert in Maxwell Render mit anschließender Nachbearbeitung in Adobe Photoshop.

4.2.1 Der Szenegraph

Zunächst besteht GroIMPs Szenegraph - dem Konzept eines Graphen entsprechend - aus Knoten (*Nodes*) und Kanten (*Edges*). Jedoch liegt die Besonderheit in den Kanten: Hierbei ist eine Unterscheidung von Kantentypen über einen 32-Bit-Wert (also über eine Bitmaske) möglich. Diese Kantenmarkierungen haben unterschiedliche Bedeutung. Es gibt zwei grundlegende Kantentypen: *successor*- und *branch*-Kanten. Beide stellen die Beziehung „Nachfolger von“ bzw. „Vorgänger von“ dar[1], dabei stehen *branch*-Kanten für Verzweigungen, und *successor*-Kanten dürfen nur exakt zu einem Nachfolger führen.
Der Subgraph, den beide Kantentypen ausgehend vom Wurzelknoten aufspannen, muss eine Baumstruktur (also keine Maschen und Zyklen) ergeben, damit für jeden Knoten, der sichtbar sein soll, eine globale Transformation[2] errechnet werden kann. Dies passiert Top-Down vom Root-Knoten durch die Bildung eines eindeutigen Pfades zum darzustellenden Knoten. Wichtig ist, dass exakt ein Pfad bestehend aus diesen beiden Kantentypen zu einem darzustellenden Knoten führt. Ausschließlich dann kann die Berechnung der globalen Transformation für eben diesen Knoten durchgeführt werden. Abbildung 4.5 stellt beispielhaft einen Szenegraphen mit *successor*-

[1] Auch definiert als „parent-child“-Beziehung

[2] Die globale Transformation ist die Verkettung aller Transformationen entlang eines Pfades des Szenegraphen.

und *branch*-Kanten dar. Neben den *successor*- und *branch*-Kanten

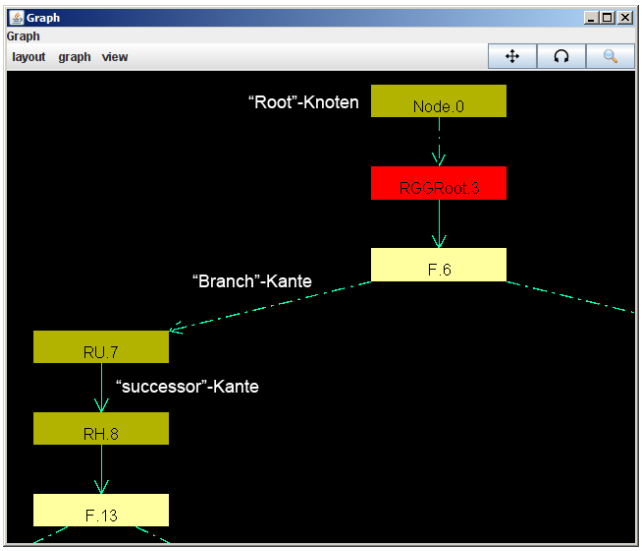


Abbildung 4.5: Beispielhafter Szenegraph aus GroIMP

ist es möglich, eine Reihe anderer Kantentypen zu definieren, welche eine Beziehung zwischen den Knoten herstellt. Über diese Kantentypen kann ebenso iteriert werden, wobei allerdings nicht die Einschränkung auf eine Baumstruktur gilt. Es kann also eine beliebige Graphstruktur dieser Kanten angenommen werden.
Knoten sind zusammen mit den Kanten die elementaren Bestandteile des Szenegraphen von GroIMP. Eine Klasse wird zu einem durch GroIMP verwendbaren Knoten, wenn diese von der Klasse `de.grogra.graph.impl.Node` ableitet. Diese Klassen können sich in einem Package eines Plugins befinden. Die in GroIMP vorhandenen Knoten sind in erster Linie im Package `de.grogra.imp3d.objects` definiert. Alle Knoten leiten direkt oder indirekt von der Klasse `Node` ab. Das Klassendiagramm 4.6 zeigt die Klassenhierarchie um die Klasse `Node`. Der Fakt, dass die Klasse `Node` von `Edge` ableitet, ist ein Konzept von GroIMP, um Speicher zu sparen[1].

Transformationsknoten

An dieser Stelle ist noch kurz zu erläutern, was in GroIMP der Begriff „Transformation“ bedeutet. Der grundlegende Begriff „Transfor-

[1]Siehe (Kniemeyer, 2008, S. 382)

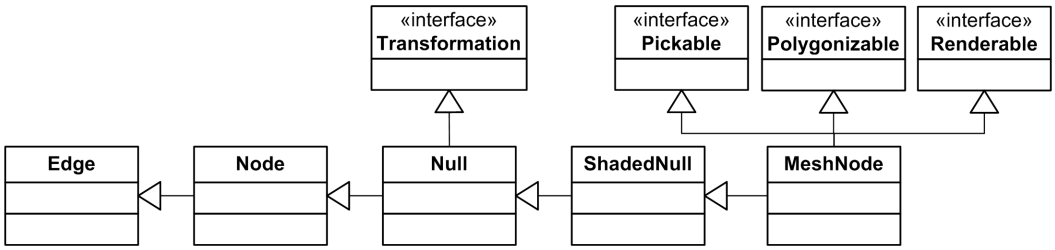


Abbildung 4.6: Klassenhierarchie von `Node`

mationsmatrix“ bzw. „Transformationsfunktion“ ist im Kapitel A.2.3 genauer erläutert.
Nun besitzt die Klasse `Node` selbst keine Transformation, um Objekte im Raum zu positionieren. Dies wird erst durch die Implementierung des Interface `Transformation`[1] sicher gestellt. Die Klasse `Null` implementiert dieses Interface und ist somit nutzbar als Teil einer Transformationshierachie[2]. Die im Kapitel 4.2 benannte „Bildung eines eindeutigen Pfades“ ist eben eine solche Transformationshierarchie. Für den Szenegraph an sich (also die Bildung globaler Transformationen) ist allein das Interface `Transformation` relevant.
Darüber hinaus gibt es noch das Interface `Transform3D` im package `de.grogra.math`, dieses beschreibt genau eine solche Transformationsfunktion. Mehrere Klassen implementieren dieses Interface. Abb. 4.7 zeigt dies im Zusammenhang. Klassen können ein `Transform3D` als Attribut halten, was jedoch nicht für jede implementierte Klasse in GroIMP zutrifft.

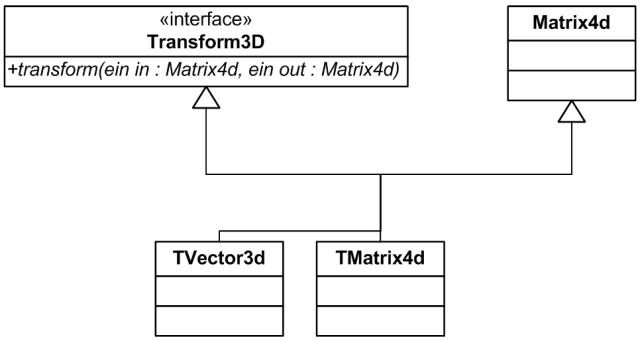


Abbildung 4.7: Das Interface Transform3D

[1] Zu finden im package `de.grogra.imp3d.objects`.
[2] Vgl. Kap. A.2.4

Shader

Ein Knoten vom Typ `Null` enthält eine Transformation, ist aber noch kein in der Visualisierung (in Bezug auf die Darstellung im Echtzeit-Fenster) darstellbarer Knoten, da er weder einen Shader noch Geometrie besitzt. Die Klasse `ShadedNull` erweitert `Null` um die Möglichkeit, einen Shader hinzuzufügen und zu speichern. `Shader` ist ein Interface, dessen Implementierung durch zwei wichtige Klassen durchgeführt wird: `Phong` (über die abstrakte Klasse `Material`) und `RGBAShader`. Abbildung 4.8 zeigt ohne Anspruch auf Vollständigkeit das Klassendiagramm[1] um die Klasse `Phong`. Hierbei stellt `Phong` eine Implementierung der Berechnungsvorschrift zur Darstellung eines Pixels dar[2].
Der durch die Klasse `Phong` zur Verfügung gestellte Shader hat den Vorteil der hohen Flexibilität, ebenso können die einzelnen Kanäle (siehe Klassendiagramm der Abb. 4.8) mit anderen ChannelMaps, also Implementierungen des Interface `ChannelMap` belegt werden. Hierbei wird eine Belegung des *Diffuse*-Kanals - z. B. mit einer `ImageMap` - direkt im 3D-Fenster von GroIMP über OpenGL dargestellt. Zu beachten ist, dass nicht alle Implementierungen von `ChannelMap` darstellbar sind. Zugewiesene Shader können über die Oberfläche von GroIMP, nämlich dem Attribut-Editor, angepasst werden.

Geometrie

Um wirklich Objekte im 3D-Fenster darzustellen (bzw. in einem der zur Verfügung gestellten Render-Plugins, wie z. B. der eingebaute Raytracer oder PovRay), benötigen diese polygonale oder auf NURBS basierende Geometrie. Es gibt in verschiedenen Plugins, wie z. B. `RGG` oder `IMP-3D` eine Reihe von implementierten Primitivobjekten. Dazu gehören die Klassen `Cylinder`, `Box`, `Sphere`, `Cone`, `Heightfield` und `Plane`, aber auch die im Zusammenhang des Plugins `RGG` implementierten Turtle-Grafiken[3]. Diese Klassen sind im package `de.grogra.turtle` untergebracht.
Darüber hinaus gibt es noch die Möglichkeit, Polygone und

[1] Basierend auf (Kniemeyer, 2008, S. 393)
[2] Mehr Informationen unter (Möller und Haines, 1999, S. 68 ff.).
[3] Weitere Informationen unter Prusinkiewicz und Lindenmayer (1990)

Nurbs darzustellen. Hierbei sind die Klassen `MeshNode` und `NURBSSurface` maßgeblich.

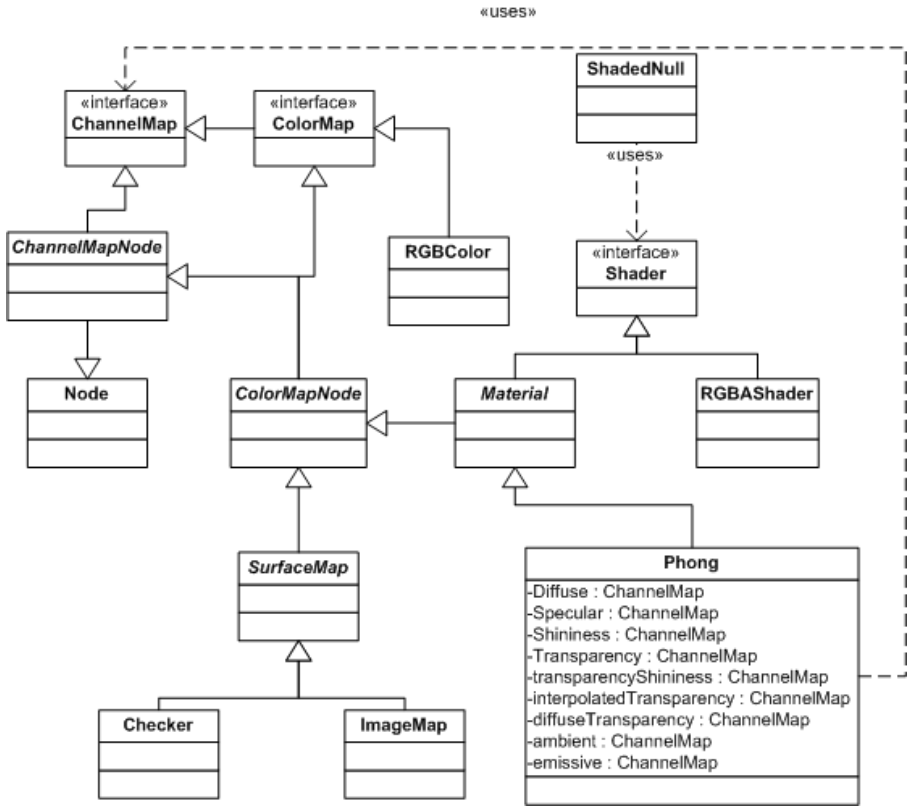


Abbildung 4.8: Klassenhierarchie des Shadersystems von GroIMP

4.2.2 Traversieren des Szenegraphen

In vielen Situationen ist es notwendig, manuell den Szenegraphen zu durchlaufen. Ebenso ist eine Filterung nach bestimmten Typen sinnvoll oder das Traversieren nur über bestimmte Kanten erlaubt. Dies lässt sich gut über eine rekursive Methode implementieren:

```
1  private void parseWithFilter(   Node                  root ,
2                                  ArrayList<Node>       list ,
3                                  NType                 filter ,
4                                  int                   edgeBits ,
5                                  boolean               exactEdgeMatches ) {
6    if (root.getNType( ).equals( filter )) {
7      if (!list.contains( root )) list.add( root );
8    }
9    for (Edge e = root.getFirstEdge( ); e != null; e = e.getNext( root )) {
10     if (exactEdgeMatches) {
11       if ((e.getEdgeBits( ) & edgeBits) != edgeBits) continue;
12     } else {
13       if ((e.getEdgeBits( ) & edgeBits) > 0) continue;
14     }
15     Node t = (Node) e.getTarget( );
16     if (t != root) {
17       parseWithFilter( t , list , filter , edgeBits , exactEdgeMatches );
18     }
19   }
20 }
```

Listing 4.1: Eine Methode zum Durchlaufen des Graphen

Der Vorteil besteht darin, dass diese Methode sehr flexibel einsetzbar ist. Der Parameter `root` erlaubt, das Durchlaufen des Graphen von einem bestimmten Knoten aus zu beginnen. Über den Parameter `filter` kann die Suche auf bestimmte Knotentypen eingeschränkt werden. Die Definition des Filters kann über `klassenname.$TYPE` - was ein Attribut vom Typ `NType` ist - durchgeführt werden. Die zweite mögliche Einschränkung - das Durchlaufen nur bestimmter Kantentypen - ist durch den Parameter `edgebits` realisierbar. `exactEdgeMatches` bedeutet, dass im Falle einer Übergabe von `true` nur jene Kanten beachtet werden, welche exakt der Vorgabe durch `edgebits` entspricht. Im Falle von `false` bedeutet dies, dass, sobald nur ein Bit in beiden Werten, nämlich in `edgebits` und dem Wert der Kante selbst, übereinstimmt, die Kante akzeptiert wird. Als Rückgabewert ist `list` gedacht.

4.2.3 Hinzufügen und Entfernen von Knoten

Das Hinzufügen ist relativ einfach:

```
1  public void addChild( Node parent , Node child ) {
2    parent.getOrCreateEdgeTo( child ).addEdgeBits(
3                                          Graph.SUCCESSOR_EDGE ,
4                                          graph.getActiveTransaction( ) );
5  }
```

Listing 4.2: Einfügen eines Knotens in den Graphen

Die Methode in Listing 4.2 kann, unter Erweiterung um einen Parameter `edgeBits`, jede Art von Kante zwischen Knoten hinzufügen. Existiert diese Kante bereits, wird nur das jeweilige Bit hinzugefügt, sofern auch dieses noch nicht existiert. Das Objekt `graph` ist vom Typ `GraphManager` und kann über die Methode `getGraph()` der Klasse `Node` bestimmt werden.
Das Entfernen eines Knotens verläuft äquivalent, indem alle Edgebits eines Knotens zu allen anderen übergeordneten Knoten gelöscht werden. Zum vollständigen Entfernen der Beziehung zwischen zwei Knoten ist folgende Methode notwendig:

```
1 public void removeChild( Node parent , Node child ) {
2   parent.getOrCreateEdgeTo( child ).removeEdgeBits(
3                                                 0xFFFFFFFF ,
4                                                 graph.getActiveTransaction( ) );
5 }
```

Listing 4.3: Entfernen einer Beziehung zwischen zwei Knoten

Eine Iteration über alle Kanten entfernt den Knoten - und mit ihm alle Knoten im darunter liegenden Subgraph - vollständig. Zu beachten ist: Das Objekt im Speicher wird erst dann durch den *garbage collector* entfernt, wenn alle anderen Referenzen auf den Knoten entfernt wurden. Da auch außerhalb der Graphstruktur Referenzen auf Instanzen von Knoten gehalten werden können, müssen diese mit einbezogen werden.

4.2.4 Graph-Locking

Der Code in den Listings 4.2 und 4.3 führt Veränderungen am Szenegraphen von GroIMP aus. Wird diese Veränderung außerhalb des GroIMP-Threads durchgeführt, und dies ist der Fall, wenn eigene Swing-Komponenten innerhalb von GroIMP eingesetzt werden, müssen Aufrufe durch diese Komponenten über das `invokeLater`-Konzept in den Thread von GroIMP eingereiht werden (Vgl. (Goetz u. a., 2006, Kap. 3.3, S.42 ff.)). GroIMP besitzt eine eigene Implementierung dieses Konzeptes. Hierbei werden Kommandos in eine Queue einsortiert und ausgeführt, sobald der Thread von GroIMP dazu bereit ist. Das bedeutet auch, dass zu dem Zeitpunkt der Ausführung keine anderen Veränderungen am Szenegraphen stattfinden können und somit die Konsistenz gesichert ist.

4.2.5 Die Klasse Job

Um ein solches Kommando an GroIMP weiter zu geben, wird eine Implementierung der Klasse `Job` benötigt. Listing 4.4 zeigt die Nutzung dieser Klasse.

```
1  new Job(context) {
2    @Override
3    protected void runImpl( Object arg , Context ctx) {
4      //TODO: Implementierung des Jobs hier einfügen
5    }
6    @Override
7    protected void done() {
8      //TODO: Code zur Ausführung nach Abschluss des Jobs hier einfügen
9    }
10 }.execute();
```

Listing 4.4: Eine Implementierung der Klasse Job

Mit der Nutzung der Klasse `Job` ergeben sich allerdings auch einige Schwierigkeiten, welche zu einem erheblich größeren Implementierungs- und insbesondere Testaufwand führen können:

- Ausschließliche Zulassung von `final`-Variablen
  Dies ist weniger ein Problem, sondern mehr ein zusätzlicher programmiertechnischer Aufwand: Jegliche in der Klasse `Job` genutzten Variablen müssen als `final` deklariert werden. Es ist allerdings möglich, unter Erhöhung des Implementierungsaufwandes, die Klasse `Job` so zu erweitern, dass die notwendigen Variablen direkt übergeben und in dieser Klasse gespeichert werden.

- Verspätete Ausführung
  Dies ist das größte Problem. Hierbei entstehen bei der Ausführung eines Jobs zwei verschiedene Steuerpfade.

Das zweite Problem sei etwas genauer erläutert, da dies eine größere Hürde bei der Implementierung und Nutzung eigener Swing-Komponenten darstellen kann. Listing 4.5 stellt eine typische Situation der Nutzung eines Jobs dar. Hierbei ist Code vor (`method2()`) und Code nach (`method3()`) der Ausführung des Jobs zu durchlaufen. Die Ableitung von der abstrakten Klasse `Job` führt zu einer Implementierung der Methode `runImpl( Object arg , Context ctx )`. Hierbei soll die Methode `method()` ausgeführt werden.

Obwohl dieser Code in linearer Form vorliegt, ist seine Ausführungsreihenfolge eine andere. Das Sequenzdiagramm in Abbildung 4.9 zeigt einen möglichen Ablauf vom Start des Jobs bis hin zur Auführung des Codes innerhalb der Methode `runImpl(..)` .

```
1  public void startJob() {
2
3    //Code vor dem Job
4    methodBeforeJob();
5
6    new Job(context) {
7      @Override
8      protected void runImpl( Object arg , Context ctx) {
9        methodInJob();
10     }
11     @Override
12     protected void done() {
13
14     }
15   }.execute();
16
17   //Code nach dem Job
18   methodAfterJob();
19 }
20
21 public void methodInJob() {
22   System.err.println("in job     " + System.currentTimeMillis( ) );
23 }
24 public void methodBeforeJob() {
25   System.err.println("before job " + System.currentTimeMillis( ) );
26 }
27 public void methodAfterJob() {
28   System.err.println("after job  " + System.currentTimeMillis( ) );
29 }
```

Listing 4.5: Aufführungsreihenfolge von Code in einem Job

```
1 before job 1221659158520
2 after job  1221659158522
3 in job     1221659158523
```

Listing 4.6: Ausgabe des Listings 4.5

- `Swing-Object`
  ist eine Instanz der Klasse, die die Methode `startJob()` enthält. Das kann beispielsweise eine Swing-Komponente sein.

- `Job`
  ist die konkrete Instanz, die durch das `new` in Zeile 6 erzeugt wird.

- `IMPJobManager`
  enthält die Hauptschleife von GroIMP und sorgt dafür, dass Instanzen der Klasse `Job` in die Event-Queue eingereiht werden.

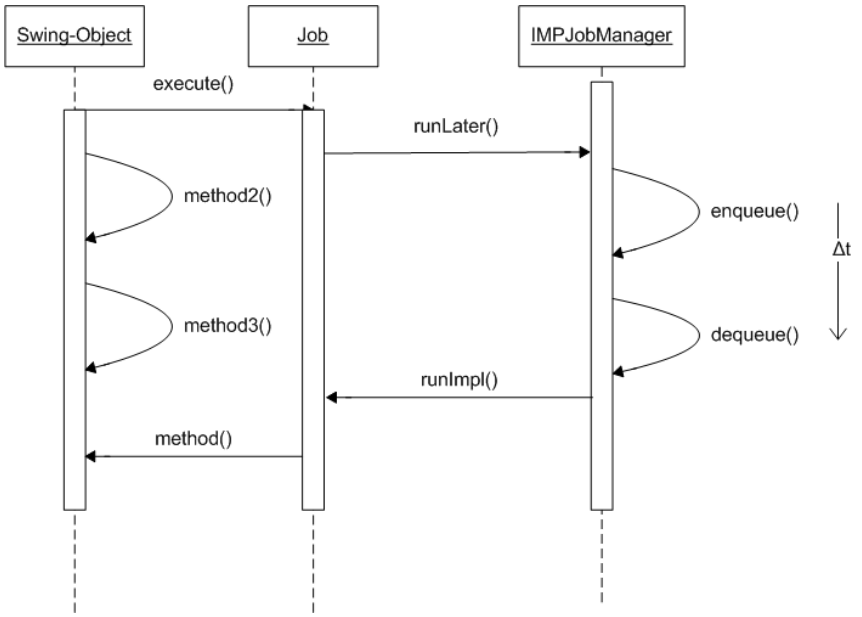


Abbildung 4.9: Sequenzdiagramm der Ausführung eines Jobs (stark vereinfacht)

Die Ausführung durch das `.execute()` in Zeile 15 bedeutet nicht, dass der Code in Zeile 9 unmittelbar ausgeführt wird. Zunächst wird durch die Klasse `Job` ein `runLater()` in der Klasse `IMPJobManager` ausgeführt, was den Job zunächst in eine Queue einfügt. Entscheidend ist, dass der Zeitraum Δt zwischen dem `enqueue()` und `dequeue()` nicht vorhersagbar ist. Listing 4.6 zeigt die Testausgabe des Beispielcodes. Es ist deutlich zu sehen, dass die Reihenfolge der Ausführung anders als im Quellcode ist.

Anzumerken ist, dass das Sequenzdiagramm in Abb. 4.9 stark vereinfacht ist. In Wahrheit sind die Mechanismen bei weitem komplexer, deren vollständige Erkläuterung hier zu weit gehen würde.
Die Nutzung der Klasse `Job` hat mehrere Konsequenzen:

- Präzisiere Planung
  Der Programmierer sollte sich von vornherein klar machen, an welcher Stelle auf GroIMP von außen zugegriffen werden muss, und gezielt die Planungen darauf ausrichten.

- Erhöhte Sorgfalt bei der Implementierung
  Es gibt mehrere technische Notwendigkeiten, die fortlaufend zu beachten sind. Dazu gehört, möglichst keinen Job innerhalb des Ablaufes eines Jobs zu starten. Dies würde die Komplexität erhöhen. Ebenso sollten nicht sehr viele Jobs in kurzer Zeit gestartet werden. Das kann dazu führen, dass der Overhead der Auführung insgesamt größer wird als der eigentlich auszuführende Code. Insbesondere, wenn z. B. auf Events des Benutzers, welche in hoher Geschwindigkeit eintreten können, reagiert wird. Hierbei kann GroIMP sehr träge und auch bis hin zum Absturz gebracht werden.

- Höherer Zeitaufwand beim Test
  Durch die verspätete Ausführung kann es passieren, dass sich der Zustand im aufrufenden Thread derart verändert hat, dass der Code innerhalb des Jobs auf veraltete Daten zugreift. Dies muss bedacht und durch geeignete Maßnahmen verhindert werden. Dazu gehört, tiefe Kopien von bestimmten Objekten zu übergeben, da sonst über den mehrfachen Zugriff Race Conditions entstehen können.

  > „Eine *Race Condition* tritt auf, wenn die Korrektheit einer Berechnung vom relativen Timing oder der Verschränkung mehrerer Threads abhängt während der Laufzeit; in anderen Worten: Wenn das Erlangen der richtigen Antwort auf Zufall basiert.“((Goetz u. a., 2006, Seite 20, Kap. 2.2.1))

  Ebenso kann auf andere Synchronisationsmechanismen zurück gegriffen werden, aber auch das erfordert eine erhöhte Aufmerksamkeit bei der Implementierung.
  Nun ist es besonders schwierig, solche nicht deterministischen

Verhaltensweisen (eben der Zeitpunkt der Ausführung des Jobs) zu testen. Hierbei kann es passieren, dass ein bestimmter Fehler im Durchschnitt nur alle eintausend Durchläufe auftritt oder nur in Zusammenhang eines bestimmten Zustandes. Dies erfordert, dass mehr Tests durchgeführt werden müssen als bei strikt linearen Programmen.

- Zeitaufwändige Fehlerbeseitigung
  Dies steht in Konjunktion zu dem vorherigen Punkt: Ist durch Test ein Fehler gefunden worden, ist dessen Beseitigung zeitweise schwer zu überprüfen.

Der Einsatz eines Profilers[1] zeigte, dass der Aufruf von Code über einen Job schnell zu einem Flaschenhals in Bezug auf die Ausführungsgeschwindigkeit führen kann.

4.3 Transactions

Damit nicht für jede relativ kleine Veränderung von GroIMP sämtliche *Listener* benachrichtigt werden, welche dann eventuell nur eine unwichtige Veränderung mitgeteilt bekommen, können Veränderungen in sogenannten *Transactions* gesammelt werden. Diese Sammlung macht GroIMP automatisch. Wird über die Klasse `Job` gearbeitet, werden sämtliche Transaktionen nach Beendigung des Jobs automatisch angewendet.
Ferner ist es möglich, innerhalb eines Jobs manuell alle Transaktionen anzuwenden. Dies kann notwendig sein, damit der darauf folgende Code auf den aktuellen Zustand des Graphen Zugriff hat. Hierfür ist folgende Codezeile notwendig:

```
1 node.getGraph( ).getActiveTransaction( ).commitAll( );
```

Die Variable `node` ist eine Instanz der Klasse `Node`.

4.4 Attribute von Knoten

Die für GroIMP implementierten Knoten haben verschiedenartige Attribute. Hierbei sei ein Attribut ein Feld einer Klasse (bzw. deren

[1] Ein Profiler misst die Ausführungsdauer und die Anzahl der Durchläufe von Code.

Instanz). Nun kann es notwendig werden, diese Attribute zu schreiben oder zu lesen. Nun hat dieses Attribut allerdings Auswirkungen auf GroIMP, und diese sollen unmittelbar bemerkbar werden (z. B. durch die Änderung einer Textur eines Objektes).
Ein Beispiel sei ein Attribut der Klasse `ShadedNull`:

```
1 protected Shader shader = null;
```

Um dieses Attribut mit dem obigen Anspruch zu setzen, ist folgender Code notwendig:

```
1 shader$FIELD.setObject( node , null, shader ,
2             node.getGraph( ).getActiveTransaction( ) );
3 node.getGraph( ).getActiveTransaction( ).commitAll( );
```

Darüber hinaus müssen diese Zeilen sich innerhalb eines Jobs befinden, sofern der Aufruf aus einem anderen Thread statt findet. Hierbei ist

- `node` der Knoten, der das Attribut hält
- `shader` der neue Wert des Attributs

Die Methode `commitAll()` in Zeile 3 sorgt dafür, wie schon im Abschnitt 4.3 erwähnt, dass die Veränderung des Attributes unmittelbar mitgeteilt wird. Hierbei werden auch andere eventuell angesammelte Transaktionen ausgeführt.

4.5 Die Registrierung

Die Registrierung von GroIMP ist im Grunde eine Sammlung von Daten in XML-Struktur, die aus den verschiedenen Plugins und deren jeweiligen *plugin.xml*-Dateien zusammen gesetzt wird. Diese Daten enthalten alle Einstellungen von GroIMP, beispielsweise welche Methode ein Button der GUI aufruft. Wichtige Teile der Registrierung befinden sich im Plugin *IMP-3D* und *Platform*, dies kann allerdings nicht pauschalisiert werden, da jedes Plugin zum Verhalten von GroIMP beitragen kann.
Es ist möglich, durch das Setzen von Attributen oder das Hinzufügen von Tags die Registrierung zu erweitern, z. B. über die *plugin.xml*

eines Plugin-Projektes. Auf diese Art und Weise können GroIMP Panels, Buttons, Layouts oder Menüpunkte hinzugefügt werden, ohne GroIMP an sich zu verändern. Auf den Aufbau einer *plugin.xml* wird im Kapitel 4.7 genauer eingegangen.

4.6 Import/Export

GroIMP bietet die Möglichkeit, Szenen als Projekt zu speichern. Hierbei wird der gesamte Graph serialisiert[1] und in einer XML-Struktur untergebracht. Darüber hinaus wird das aktuelle Layout der Oberfläche mit gespeichert, sodass das Aussehen bei einem erneuten Laden des Projektes erhalten bleibt. Beim Speichern hat der Anwender die Auswahl, in welchem Format die Szene gespeichert werden soll. Der Standard ist das GroIMP-Szenenformat, komprimiert mit *gsz* (*Project-Archive*).
Über das GroIMP-eigene Projektformat hinaus bieten sich - hinzugefügt über verschiedene Plugins - andere Speicherformate[2], beispielsweise X3D[3].
Neben dem Laden von Projekt-Dateien ist es möglich, einige andere Datenformate zu lesen und in die Szene einzufügen. Hierzu gehört ebenso *X3D*, aber auch *DXF* oder *OBJ*.

4.7 Grundzüge einer Erweiterung

Dieser Abschnitt beschäftigt sich mit den grundlegenden Schritten, welche für eine Erweiterung von GroIMP notwendig sind. Diese Ausführungen basieren größtenteils auf Kniemeyer (b).

Auschecken

Der erste Schritt besteht zunächst darin, einige minimal notwendige Projekte von der Projektquelle (siehe sourceforgeGroIMP) herun-

[1] Die Serialisierung ermöglicht, ein im Speicher vorliegendes Objekt in einen linearen String umzuwandeln, um diesen z. B. in einer Datei zu speichern. Anschließend kann das Objekt aus diesen Informationen wieder hergestellt werden. Vgl. hierzu: (Krüger, 2006, S. 957 ff.)

[2] Eine vollständige Übersicht aller Formate ist zu finden unter (Kniemeyer, 2008, S. 389)

[3] Extended 3D, eine auf XML basierende Szenenbeschreibungssprache

ter zu laden, wobei ein Subversion-Werkzeug[1] dafür notwendig ist. Für die Entwicklungsplattform *eclipse* ist *Subclipse*[2] ein gängiges Werkzeug, nämlich ein Client für Subversion, welcher sich direkt in eclipse integriert. Folgende Projekte sind für eine grundlegende Funktionalität hilfreich:

- Plattform, Plattform-Core, Plattform-Swing
  enthalten das Basissystem und die GUI.

- Graph
  ist eine Implementierung der GroIMP-Graph-Schnittstellen.

- Math, Vecmath, Utilities, JEdit
  sind nützliche Bibliotheken für mathematische Berechnungen, JEdit ist ein fortschrittlicher Editor für Java und XL.

- Imp, Imp-2d, Imp-3d
  bilden die Grundlage für Visualisierungen in 2D und 3D.

Darüber hinaus können noch folgende hinzugefügt werden:

- X3D, DXF
  sind Importer und Exporter für das X3D und DXF-Datenformat.

- XL, XL-Compiler, XL-Core, XL-Impl, XL-VMX, RGG
  sind die notwendigen Projekte zur Nutzung von XL mit direkter Anbindung an GroIMP.

- Raytracer, Sunshine, POV-Ray
  erweitern GroIMP um die Möglichkeit, Szenen mit verschiedenen Rendertechniken darzustellen.

Zwischen den Projekten existieren bestimmte Abhängigkeiten, sodass manche Projekte nicht ohne bestimmte andere nutzbar sind. Eine vollständige Übersicht ist (Kniemeyer, 2008, S. 380) zu entnehmen.

[1] Subversion (SVN) ist ein Versionierungssystem für Dateien und Quellcode.
[2] Siehe `http://subclipse.tigris.org`

Anlegen des Projektes

Nach dem Auschecken der Projekte kann ein neues Projekt als Plugin für GroIMP angelegt werden. Wichtig ist, dass dieses neue Projekt sich in einem Parallelordner zu den grundlegenden Plattform-Projekten befindet. Damit kann GroIMP das Projekt selbstständig als Plugin einordnen und es müssen keine zusätzlichen Maßnahmen zu den folgenden ergriffen werden.
Anschließend muss - ausgehend von der durch *eclipse* erzeugten Ordner-Struktur, wonach ein Ordner „*src*“ automatisch angelegt wird - das Plugin zunächst angemeldet werden. Dies passiert über die zu erzeugende Datei *plugin.xml*. Darüber hinaus ist im Hauptverzeichnis des Plugins eine Datei names *build.xml* anzulegen. Diese ist notwendig, um das Projekt entsprechend bestimmter Vorgaben zu kompilieren und eine Jar-Datei zu erzeugen. Listing B.1 zeigt die verwendete *build.xml*. Darüber hinaus muss das Plugin einer bestimmten Ordnerstruktur folgen, welche Kniemeyer (b) oder der Datei *INSTALL* des Projektes *Build* zu entnehmen ist.

build.xml

Über die XML-Datei *build.xml* werden die für *Ant*[1] notwendigen Einstellungen gesetzt. Dies ist notwendig, damit Ant entsprechend die für die spätere Ausführung von GroIMP notwendigen Bibliotheken in der jar-Datei vereinigt. Ant sollte unabhängig von der genutzten Entwicklungsplattform für das Release genutzt werden.
Nachfolgend sind einige der wichtigen Tags genauer erläutert:

- `<antcall target="-addproject">`
  fügt ein anderes Projekt ein. Hierbei ist zu beachten: Auf welche Projekte an dieser Stelle zu verweisen ist, ist nach den Abhängigkeiten nach (Kniemeyer, 2008, S. 380) auszuwählen. Ist auf ein Projekt bereits durch ein anderes verwiesen, muss dieses nicht erneut aufgeführt werden.

- `<target name="src">`
  ist notwendig für das *enhance*-Skript, falls eigene Knotenty-

[1] „Ant steuert den Build-Prozess über XML-Dateien, die Projekte, Abhängigkeiten und Arbeitsschritte in Form von Tasks definieren.“((Edlich und Staudemeyer, 2006, S. 7, Abs. 3))

pen zum Einsatz kommen. Dies wird im Kapitel 4.7.1 genauer erläutert.

plugin.xml

Die Datei *plugin.xml* dient dazu, ein Projekt als Plugin in GroIMP anzumelden und zugleich dessen Registrierung zu erweitern. Hierfür sammelt GroIMP bei der Initialisierung alle *plugin.xml*-Dateien und setzt diese zur Registrierung zusammen. Für das Projekt müssen ebenso eventuell verwendete Bibliotheken angemeldet werden. Dies ist notwendig, damit diese bei der Ausführung von GroIMP gefunden werden und deren Code ausgeführt wird. Listing B.2 zeigt die für ArchiViz implementierte *plugin.xml*. Nachfolgend seien einige wichtige Punkte bzw. Tags näher erläutert.

- `<id>` enthält unter anderem den Package-Pfad des Plugins.
- `<library>` bindet externe Libraries ein. Dies ist notwendig, damit GroIMP auf den darin enthaltenen Code zugreifen kann, auch wenn die Aufrufe der Bibliothek nur aus dem Plugin selbst heraus ausgeführt werden. Das Attribut `prefixes` enthält eine Liste der verwendeten Packages.
  Es ist notwendig, hier das eigene Plugin anzumelden, da das
- `<plugin>` verweist auf die in dem Plugin verwendeten anderen Plugins (bzw. Packages) GroIMPs. Hierbei können die übergeordneten Packages gewählt werden. Im Allgemeinen sind dies die selben Packages, welche sich im Java-Build-Path des Plugin-Projektes befinden.

Ebenso ist es möglich, vorhandene Einstellungen neu zu setzen bzw. zu erweitern. Ein Beispiel, solche Einstellungen zu überschreiben, ist folgendes:

```
1 <de.grogra.pf.ui.registry.MainWindow menuVisible="false">
2 </de.grogra.pf.ui.registry.MainWindow>
```

Das Setzen des Attributs `menuVisible` sorgt dafür, dass das Hauptmenü abgeschaltet wird. Theoretisch kann jeder Wert der Registrierung in dem Projekt neu gesetzt werden.

Beim Speichern eines Projektes legt GroIMP eine weitere xml-Datei an, welche die Registrierung ihrerseits erweitert. Zum einen wird auf diese Art das aktuelle Layout gespeichert, zum anderen können an dieser Stelle auch projektweise spezifische Einstellungen durchgeführt werden.
Bei einem der früheren Entwürfe von ArchiViz wurde dieses Konzept genutzt, um beim Start eines festgelegten Projektes GroIMP entsprechend anzupassen. Doch es erwies sich als günstiger, die ArchiViz-GUI direkt über das Plugin statt über ein zu startendes Projekt in GroIMP zu verankern.
Somit ist es möglich, bestimmte Startzustände direkt in GroIMP als Projekt zu laden. Um ein Projekt direkt beim Start vom GroIMP zu laden, muss dieses als Kommandozeilen-Option übergeben werden:

```
1 --project-tree "X:\your_folder\groIMP_startproject.gsz"
```

Empfehlenswert ist auch eine Startoption, die den Speicher der virtuellen Maschine reguliert (VM-Argument):

```
1 -Xms712m -Xmx1048m
```

Somit werden eventuelle Speicherprobleme durch den höheren Startspeicher (Option -Xms) von Anfang an umgangen.

plugin.properties

Die Datei *plugin.properties* ist eine anzulegende Datei, welche bestimmten Elementen der *plugin.xml* Strings, also Namen zuweist. GroIMP verarbeitet beide Dateien gemeinsam. Die Listings 4.7 und 4.8 soll diesen Zusammenhang erläutern. Hierbei verweist die Zeile in Listing 4.8 auf einen Pfad in der *plugin.xml*, worauf nach dem „ =“ ein `String` zugewiesen wird. Abbildung 4.10 zeigt die Wirkung der String-Übergabe: Das definierte Panel trägt den entsprechenden Namen aus der Datei *plugin.properties*.
Für die Erklärung der Definition eines benutzerdefinierten Panels wird auf Kapitel 4.7.3 verwiesen.

```
<registry>
 <ref name="ui">
  <ref name="panels">
   <directory name="archiviz">
  <panel name="presentation">
  ...
  </panel>
  ...
   </directory>
   ...
  </ref>
  ...
 </ref>
</registry>
```

Listing 4.7: Auszug aus der *plugin.xml* des Plugins ArchiViz

```
/ui/panels/archiviz/presentation.Name = Presentation Window
```

Listing 4.8: Auszug aus der *plugin.properties* des Plugins ArchiViz

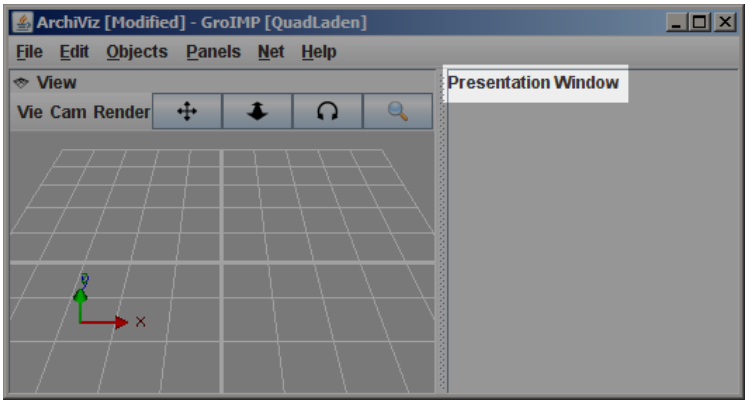


Abbildung 4.10: Name eines benutzerdefinierten Panels

4.7.1 Implementierung eigener Knoten

Die folgenden Ausführungen basieren in erster Linie auf Kniemeyer (a). Grundlegend ist der Fakt, dass eine Klasse von `Node` ableitet, ausreichend, um einen neuen Knoten zu definieren und GroIMP zur Verfügung zu stellen. Allerdings wird dieser Knoten nicht beachtet, wenn dieser im Projekt-Format von GroIMP gespeichert werden soll. Darüber hinaus kann es zu ungewünschten Nebeneffekten

kommen.
Um einen Knoten mit seinen Attributen (ein Attribut ist ein Feld der Klasse mit einer bestimmten Sichtbarkeit) zur Speicherung zuzulassen, steht ein Perl-Skript (*enhance.pl*) zur Verfügung, welches über *Ant* - gesteuert über die Datei *build.xml* - ausgeführt wird. Dieses Skript ergänzt die erstellte Klasse um einigen Code, der zum Lesen und Schreiben der Attribute notwendig ist. Die Zeilen 25 bis 27 des Listings B.1 sind notwendig, um das Skript über *Ant* auszuführen.

Das Enhance-Skript

```
1 import de.grogra.graph.impl.Node;
2 public class Cube extends Node {
3   float width = 1;
4   //enh:field getter setter
5
6   //enh:insert
7 }
```

Listing 4.9: Beispiel eines Knotens

Listing 4.9 (entnommen aus Kniemeyer (a)) ist ein kleines Beispiel einer Klasse, die von `Node` ableitet. Durch das Setzen der Optionen `getter setter` in Zeile 4 werden zusätzlich Methoden zum Lesen und Schreiben des Feldes `width` erzeugt. Diese Optionen sind nicht zwingend notwendig und sollen nur zusätzlichen Programmieraufwand ersparen.
Nach der Ausführung des *enhance*-Skriptes wird der generierte Code nach dem Tag `//enh:insert` in Zeile 6 eingefügt.
Nun ist der Typ des Feldes von einem primitiven Datentyp. Mit primitiven Datentypen weiß GroIMP in Bezug auf die Serialisierung umzugehen.
Nicht serialisierbare Objekte, die dennoch als Attribute verwendet werden sollen, müssen dem *enhance*-Skript bekannt gemacht werden. Die drei Klassen in den Listings 4.10, 4.11 und 4.12 sollen ein erweitertes Beispiel zur Anwendung des *enhance*-Skriptes darstellen. Hierbei leitet `BasicState` von der Klasse `ShareableBase` ab, was die Grundlage für eine Speicherung eines Attributs dieses Typs bildet. Nun ist zu beachten, dass die Klasse, welche direkt von `ShareableBase` ableitet, mit der Zeile

```
1 //enh:sco de.grogra.persistence.SCOType
```

```
1 public class OwnNode {
2   GlobalNodeState state;
3   //enh:field getter setter
4 }
```

Listing 4.10: Klasse OwnNode

```
1 public class GlobalNodeState
2                     extends BasicState {
3   //enh:sco
4 }
```

Listing 4.11: Klasse GlobalNodeState

```
1 public class BasicState
2                     extends ShareableBase {
3   //enh:sco de.grogra.persistence.SCOType
4 }
```

Listing 4.12: Klasse BasicState

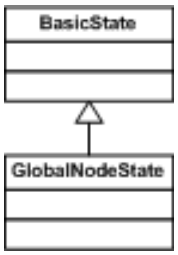


Abbildung 4.11: Klassenhierarchie

und jede weitere davon ableitende mit

```
1 //enh:sco
```

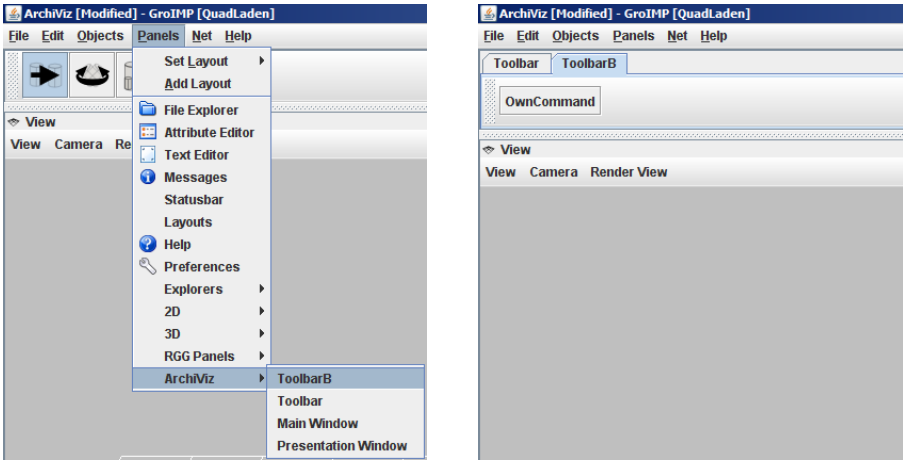
zu markieren ist. Wenn diese Grundlagen eingehalten werden, kann eine Klasse wie `OwnNode` durch GroIMP gespeichert werden, inklusive nicht-primitiven Attributen. Ebenso ist es möglich, eine Liste dieser Objekte zu speichern, die Anwendung des Skriptes muss analog durchgeführt werden.

4.7.2 Toolbars und Buttons

Dieser Abschnitt stellt einige Informationen zur Erzeugung benutzerdefinierter Toolbars und Buttons zusammen.
Listing 4.13 zeigt beispielhaft die Erzeugung eines Buttons und einer Toolbar über eine Erweiterung der Registrierung.

```
<registry>
 <ref name="ui">
  <ref name="panels">
   <directory name="archiviz">
    <toolbar name="toolbarB">
     <exists name=".available" ref="/project"/>
     <directory name="menu">
      <directory name="src">
       <!--Erzeugung eines Kommandos und Verweis auf static-Methode-->
       <command name="OwnCommand"
        run="de.grogra.ext.archiviz.MainMenu.createTheMenu"/>
      </directory>
     </directory>
    </toolbar>
   </directory>
  </ref>
 </ref>
</registry>
```

Listing 4.13: Erzeugung eines Buttons und einer Toolbar



(a) ToolbarB als neuer Menüpunkt (b) Ein neuer Button der ToolbarB

Abbildung 4.12: Menüs aus GroIMP nach der Erzeugung einer Toolbar

4.7.3 Panels

Dieser Abschnitt beschreibt die Schritte zur Erzeugung eines eigenen Panels, innerhalb dessen beliebige Swing-Komponenten untergebracht werden können. Zunächst muss eine Klasse erzeugt werden, welche von `de.grogra.pf.ui.swing.PanelSupport` ableitet. Listing 4.16 zeigt eine solche Implementierung beispielhaft.

```
1  public class SurroundingWindow extends PanelSupport {
2   private OwnPanel ownPanel;
3
4   public SurroundingWindow( Context ctx , Map params ) {
5    super(new SwingPanel(null));
6    ownPanel = new OwnPanel();
7    Container c = ((SwingPanel) getComponent ()).getContentPanel();
8    c.add(ownPanel);
9    initialize ((WindowSupport) ctx.getWindow(), params);
10  }
11
12  public class OwnPanel extends JPanel{
13   //eigene Swing-Elemente hier einfügen
14  }
15 }
```

Listing 4.14: Klasse zur Bereitstellung eines GroIMP-Panels

Die lokale Klasse `OwnPanel` erbt von einem normalen cnJPanel aus Swing, sodass innerhalb dessen ebenso alle anderen Swing-Komponenten ohne Einschränkung genutzt werden können. Hierbei ist zu beachten: Wie bereits im Kap. 4.2.4 beschrieben, werden diese Swing-Komponenten durch einen eigenen Event-Thread unabhängig von GroIMP ausgeführt. Das bedeutet, dass jegliche Benutzerinteraktionen, die Auswirkungen auf GroIMP haben sollen, in einen `Job` verpackt werden müssen. Nun muss das definierte Panel über eine Erweiterung der Registrierung noch GroIMP zur Verfügung gestellt werden:

```
1  <directory name="archiviz">
2   <panel name="ownPanel">
3    <object expr="de.grogra.ext.archiviz.SurroundingWindow">
4     <!--Definition einer zu übergebenen Variable-->
5     <!--"context" ist ein feststehender Begriff-->
6     <var name="context"/>
7     <vars/>
8    </object>
9   </panel>
10 </directory>
```

Listing 4.15: Klasse zur Bereitstellung eines GroIMP-Panels

Der anzuzeigende Name des Panels muss über die Datei *plugin.properties* (Vgl. Kap. 4.7) definiert werden. Anschließend kann `ownPanel` über das Hauptmenü GroIMPs aktiviert und auf der GUI platziert werden. Nach einem Speichern der Szene werden diese Informationen direkt in die Datei *project.gs* in der Projekt-Datei geschrieben, welche ihrerseits ebenso die Registrierung erweitert, wenn das Projekt erneut geladen wird. Ebenso steht dieses Panel

bei einer Layout-Definition zur Verfügung.

4.7.4 Layouts

Dieser Abschnitt zeigt beispielhaft, wie ein Layout in GroIMP definiert wird. Ein Layout ist eine bestimmte Anordnung von Toolbars und Panels. Es ist möglich, zur Laufzeit von GroIMP zwischen den Layouts beliebig zu wechseln.

```
<ref name="layouts">
 <layout name="archiviz">
  <exists
   name=".available"
   ref="/project" />
  <main>
   <!--Aufteilung in linken und rechten Teil-->
   <split orientation="horizontal">
    <!--Aufteilung des linken Teils in oben und unten-->
    <split orientation="vertical">
     <!--Verweis auf zuvor definierte Toolbar-->
     <link source="/ui/panels/archiviz/toolbar" />
     <!--Verweis auf 3D-Fenster-->
     <link source="/ui/panels/3d/defaultview" />
    </split>
    <!--Verweis auf Panel-->
    <link source="/ui/panels/archiviz/main" />
   </split>
  </main>
 </layout>
</ref>
```

Listing 4.16: Registrierung zur Bereitstellung eines GroIMP-Panels

In Zeile 17 wird auf ein äquivialent zu Kapitel 4.7.3 definiertes Panel verwiesen.

4.7.5 Hooks, Listener und Consumer

Oftmals ist es notwendig, auf bestimmte Aktionen GroIMPs zu reagieren. Es gibt eine Reihe von Möglichkeiten, auf Ereignisse zu reagieren, wovon eine Auswahl nachfolgend beschrieben wird.

Hooks

GroIMP befindet sich während der Laufzeit in verschiedenen übergeordneten Zuständen. Nun kann es notwendig sein, dass bestimmer Code eines Plugins nur in einem bestimmen Zustand aus-

geführt wird. Dies kann auch ein Zeitpunkt sein, denn GroIMP durchläuft beim Start verschiedene Zustände. Die Registrierung von bestimmten (statischen) Methoden, welche zur Ausführung gebracht werden sollen, passiert auch hier in der *plugin.xml* des jeweiligen Plugins. Hierfür wird durch Referenzierung bestimmter *Hooks*, welche bestimmten Zuständen von GroIMP zugeordnet sind, entsprechend auszuführender Code angemeldet. Folgende *Hooks* sind in GroIMP definiert:

- `postboot`
- `startup`
- `shutdown`
- `configure`
- `complete`
- `projectloaded`

Nachfolgend ein kleines Beispiel, wie eine statische Methode `init` ausgeführt wird, wenn ein Projekt vollständig geladen wurde:

```
<ref name="hooks">
 <ref name="projectloaded">
  <!--beliebiger Name-->
  <hook name="archiviz">
   <!--statische Methode init bekommt als Parameter
       ein Objekt vom Typ Context-->
   <void method="de.grogra.ext.archiviz.Init.init">
    <!--context als Parameter von init-->
    <var name="context" />
   </void>
  </hook>
 </ref>
</ref>
```

Listing 4.17: Ausführung von Code nach dem Laden eines Projektes

Die aufgerufene Methode ist entsprechend definiert:

```
public static void init(final Context ctx) {...}
```

Listing 4.18: Definition der Methode `init`

Selektionen

GroIMP bietet an, Objekte im 3D-Fenster direkt per Mausklick zu selektieren. Um auf dieses Event mit benutzerdefiniertem Code zu reagieren, muss ein entsprechender *Listener*[1] registriert werden. Entscheidend ist, wann diese Registrierung statt finden soll. Dies kann entsprechend Kapitel 4.7.5 entschieden werden. Üblicherweise ist es ausreichend, wenn die Registrierung als *Hook* von `projectloaded` stattfindet. Theoretisch ist es aber möglich, jederzeit entsprechende *Listener* zu registrieren, *Hooks* bieten hierfür eine Hilfe.
Eine Klasse muss das Interface `de.grogra.util.EventListener` implementieren, damit diese Klasse als *Listener* eingesetzt werden kann. Alternativ kann das Interface auch direkt implementiert werden.

```
1  ViewSelection.PROPERTY.addPropertyListener(
2       View3D.getDefaultView( context ),
3       new EventListener( ) {
4         UIPropertyEditEvent e = (UIPropertyEditEvent) event;
5         ViewSelection v = ViewSelection.get( e );
6         ViewSelectionChanged vsc = (ViewSelectionChanged) e.getInfo( );
7           //herausfinden des selektierten Knotens
8         Node nodeToHandle = (Node) vsc.getPath( ).
9                     getObject(vsc.getPath( ).getNodeAndEdgeCount( ) - 1);
10          //weiterer Code
11    });
```

Listing 4.19: Registrierung eines Selektionslisteners

Der in Listing 4.19 durchgeführte Aufruf mit Implementierung des Interface registriert einen *Listener*, der den selektierten Knoten bestimmt. Anschließend ist es möglich, mit weiterem Code auf dieses Event zu reagieren.

Änderung von Kanten

Eine Implementierung des Interface `EdgeChangeListener` im package `de.grogra.graph` führt bestimmten Code genau dann aus, wenn sich eine Kante im Graphen von GroIMP ändert. Implizit bedeutet das, auch auf das Hinzufügen oder Entfernen von Knoten reagieren zu können. Das Listing 4.20 zeigt, wie ein neu-

[1] Ein *Listener* ist eine Klasse, die das Interface *Listener* implementiert, welches dem Beobachter im Beobachter-Muster (vgl. Gamma u. a. (2004)) entspricht. Hierbei können Klassen andere Klassen beobachten, wobei die beobachtete Klasse die beobachtende Klasse nicht kennt.

```
1 context.getWorkbench( ).getRegistry( ).getProjectGraph( ).addEdgeChangeListener(
2       new EdgeChangeListener( ) {
3         public void edgeChanged( Object source , Object target ,
4                                  Object edge , GraphState gs ) {
5        //Reaktion hier implementieren
6                //source und target können nach Node gecastet werden
7       }
8   } );
```

Listing 4.20: Reaktion auf eine Kantenänderung

er `EdgeChangeListener` angemeldet werden kann. Das Objekt `context` kann beispielsweise wie in Listing 4.17, Zeile 9 beschafft werden.

Veränderung von Attributen

Die Veränderung eines Attributes bedeutet im Grunde, dass eine Transaktion (Vgl. Kap. 4.3) durchgeführt bzw. abgeschlossen wird. Um auf dieses Ereignis reagieren zu können, müssen zweierlei Schritte durchgeführt werden:

1. Implementierung von `Transaction.Consumer`
   Das Interface Transaction.Consumer ist von der Klasse zu implementieren, welche die Nachrichten empfangen soll. Um beispielsweise auf die Veränderungen eines Attributes (bzw. eines Feldes) zu reagieren, kann der entsprechende Code in die Implementierung der Methode

   ```
   1 public void setField( PersistenceCapable o,
   2                       PersistenceField field,
   3                       int[] indices, Reader reader);
   ```

   eingefügt werden. Ein simples

   ```
   1 System.out.println("Setfield: " + o);
   ```

   liefert Ausgaben wie diese:

   ```
   1 Setfield: de.grogra.ext.archiviz.nodes.AVZMeshNode[id=6]@f346dc
   ```

An dieser Stelle seien noch einige weitere Methoden erwähnt, welche im Umgang mit Events aus GroIMP wichtig sind.

- `public void begin();` und
- `public void end();`
  werden unmittelbar vor und nach dem Event ausgeführt. Sie können genutzt werden, um Initialisierungen und Deinitialisierungen durchzuführen.
- `public void insertComponent(...);` und
- `public void removeComponent(...);`
  werden nach dem Hinzufügen bzw. Löschen von Array-Komponenten ausgeführt.

Für alle Methoden gilt: Über das Objekt vom Typ `PersistanceCapable` ist ein expliziter Cast zu Node möglich, da Node dieses Interface direkt implementiert. Allerdings sollte das vorher über `instanceof` geprüft werden, da auch andere Klassen dieses Interface implementieren können. So lässt sich leicht nachvollziehen, welcher Knoten das Ereignis ausgelöst hat.

2. Implementierung des Interface `XAListener`
   Diese Implementierung bildet die Grundlage, dass die verschiedenen *Consumer* mit Nachrichten versorgt werden. Listing B.3 im Kap. B zeigt eine beispielhafte Implementierung, welche normalerweise nicht grundlegend verändert werden muss.

3. Registrierung des `XAListener` und Hinzufügen von `Transaction.Consumer`
   Hierfür sind nur wenige Zeilen notwendig:

```
1  GraphManager gm =
2    context.getWorkbench( ).getRegistry( ).getProjectGraph( );
3  AVZXAListener listener = new AVZXAListener( gm );
4  gm.addXAListener( listener );
5
6  //hinzufügen beliebig vieler Consumer
7  //con1 und con2 implementieren das Interface Transaction.Consumer
8  listener.addSuppliable( con1 );
9  listener.addSuppliable( con2 );
10 ...
```

Listing 4.21: Registrierung des XAListeners

Auf diese Art und Weise können beliebig viele unterschiedliche Consumer, sofern sie das Interface `de.grogra.persistence.Transaction.Consumer` implementieren, hinzugefügt werden. Dies erlaubt spezifische Reaktionen auf die Veränderungen von Attributen.

4.8 Fazit

GroIMP ist eine Plattform, die seitens des Lehrstuhls Grafische Systeme mit hohem Aufwand entwickelt und gepflegt wurde. Im Laufe dieser Arbeit entwickelten sich einige Standpunkte, welche nachfolgend kurz beschrieben werden: GroIMP ist ...

- ... komplex
  Mit insgesamt über 3000 Klassen[1] entwickelte sich GroIMP zu einer komplexen Visualisierungs-Software.

- ... flexibel und erweiterbar
  Durch die auf Plugins basierende Struktur ist GroIMP in vielerlei Hinsicht erweiterbar und kann somit in vielen Bereichen eingesetzt werden.

- ... wenig dokumentiert
  Durch die zunächst geringen Anforderungen an GroIMP (GroIMP wuchs mit seinen Aufgaben) wurde der Dokumentation zunächst weniger Priorität eingeräumt. Die Dokumentation ist jedoch einer der nächsten geplanten wichtigen Schritte. Dieses Kapitel soll unterstützend Informationen bieten, um GroIMP als Plattform für eventuelle Erweiterungen effizient zu nutzen.

- ... durchdacht
  GroIMP nutzt verschiedene, fortschrittliche Konzepte, um performanter zu sein als vergleichbare Software. So ist es beispielsweise möglich, Szenegraphen mit mehr als 1.000.000 Knoten zu handhaben.

- ... an einigen Punkten verbesserungsfähig
  Insbesondere die Modelling-Fähigkeit und die Benutzerinteraktion bedürfen noch einiger Verbesserungen. Eine intuitive

[1] Vgl. (Kniemeyer, 2008, S. 375)

Benutzeroberfläche (die Basis hierfür ist bereits vorhanden) und Werkzeuge, die die tägliche Arbeit erleichtern, bilden die Grundlage für weitere Entwicklungsschritte und können den Benutzerkreis spürbar erhöhen.

- ... eine gute Grundlage für ArchiViz
  GroIMP dient als Plattform für das in dieser Arbeit entwickelte Tool. Hierbei bot GroIMP eine breite Basis, die für das Plugin *ArchiViz* genutzt werden konnte.

Nach der längeren Arbeit mit GroIMP ergeben sich einige Notwendigkeiten, denen bei der weiteren Entwicklung von GroIMP Aufmerksamkeit geschenkt werden sollte:

- Undo-Funktion
  Bisher ist es nicht möglich, Aktionen wie das Verschieben eines Objektes rückgängig zu machen.

- Einfache numerische Eingabe und Anzeige von Transformationsparametern.
  Es sollte jederzeit möglich sein, die Transformation eines Objektes einzusehen oder anzupassen, ohne im Attributeditor arbeiten zu müssen. Ebenso ist es wenig sinnvoll, dabei direkt die Transformationsmatrix zu editieren: Diese sollte in Translation, Rotation und Skalierung aufgelöst werden.

- Ein Umschalten zwischen lokalem und globalem Koordinatensystem beim Verschieben.
  Nach einer Rotation eines Objektes ist es nur noch möglich, dieses entlang des lokalen Koordinatensystems zu verschieben. Doch dies schränkt die Benutzerfreundlichkeit sehr ein.

Weniger notwendig ist es, GroIMP zu einem vollständigen polygonalen Modellierwerkzeug auszubauen: Hierfür gibt es bereits gut entwickelte Werkzeuge, teilweilse sogar als Open Source. Beispiele sind Blender und Wings3D. Interessanter ist es, die Interaktion mit diesen Programmen zu verbessern, in dem Modelle nahtlos ausgetauscht werden können. Insbesondere folgender Workflow könnte für GroIMP wertvoll sein:

1. Erzeugung eines Modells in einem Modellierwerkzeug

2. Export des Modells, Import in GroIMP

3. Erweiterung bzw. Bearbeitung des Modells in GroIMP und XL
4. Export des Modells

Hierbei sollte insbesondere Wert darauf gelegt werden, dass ein zuvor importieres Modell nach dem erneuten Export aus GroIMP exakt die gleiche Eigenschaften besitzt (Koordinatensystem, Skalierung), sodass die Objekte leicht in der ursprünglichen Szene zu identifizieren sind. Dazu gehört auch, dass GroIMP der Umgang mit Polygonen mit mehr als drei Vertices hinzugefügt wird.
Darüber hinaus ist GroIMP darauf ausgelegt, mit teilweise extrem vielen Szenegraph-Knoten umzugehen. Das beherrscht nicht jedes Programm. Deshalb wäre es günstig, durch XL generierte Modelle in ihrer Knotenzahl reduzieren zu können, indem beispielsweise Transformationsknoten oder Ketten von Geometrieknoten zusammengefasst werden.

5 2D-Canvas

5.1 Motivation

Im Zuge der Planung des Plugins ArchiViz und dessen Funktionalität in Bezug auf die Bearbeitung von Bildern stellte sich heraus, dass als eine Basis für weitere Implementierungen ein Werkzeug zur Darstellung und Manipulation von Grafiken notwendig wurde. Dies sollte in erster Linie ein übergeordnetes Koordinatensystem darstellen, in welchem auf Pixel basierende Grafiken - z. B. Bilder - beliebig platziert und dargestellt werden. Ebenso sollte eine grundlegende Funktionalität geschaffen werden, dazu gehört:

- Selektion von Objekten, z. B. durch Anklicken.
  Diese Selektionen sollten durch eine Liste abrufbar sein.

- Bewegen von Objekten.
  Dies sollte durch direktes „Klicken-Und-Ziehen“ möglich sein.

- Die Möglichkeit, verschiedene *Observer*[1] für bestimmte Aktionen des Benutzers zu registerieren.
  Das ermöglicht, weitere Funktionalität zu implementieren, ohne in vorhandenen Code eingreifen zu müssen.

- Einfaches Definieren, Platzieren und Darstellen von zu zeichnenden Objekten.

- Freies Scrollen und Zoomen.
  Damit hat der Anwender die Möglichkeit, selbst über den Bildausschnitt zu entscheiden, also Details zu betrachten oder sich einen Überblick zu verschaffen.

[1] Observer ist ein Entwurfsmuster, welches ermöglicht, Änderungen einer Klasse anderen Klassen mitzuteilen, ohne diese fest aneinander zu binden. Siehe dazu (Gamma u. a., 2004, S.287 ff.).

Java bietet über die *AWT*[1] und *Swing* bereits ausgefeilte Konzepte, um Grafiken zu erzeugen, darzustellen und zu manipulieren, sodass es in der vorliegenden Form als Grundlage dienen konnte, aber deutlich erweitert werden musste. Darüber hinaus sind Aktionen wie Selektion von Objekten nicht vorgesehen: Sind Objekte einmal gezeichnet, ist eine Rückführung auf das eigentliche Objekt (also die Instanz der Klasse im Speicher) nicht mehr ohne weiteres möglich. Das bedeutet: Ein gezeichneter Kreis ist nur eine Ansammlung von Pixeln, ein Objekt `Circle`, das den Kreis zeichnet, hat anschließend keine Verbindung mehr zu diesen Pixeln. Attribute wie „Radius“ spielen dann keine Rolle mehr.
Dieses Kapitel soll das entwickelte Werkzeug aus technischer Sicht näher beleuchten. Zu beachten ist, dass an dieser Stelle nur eine auszugsweise Dokumentation möglich ist. In erster Linie soll erläutert werden, welche Klassen und Methoden für einen erfolgreichen Einsatz des Werkzeuges maßgeblich sind und welche Funktionalität zur Verfügung steht.

5.2 Grundlagen

Nachfolgend seien zunächst einige wichtige Begriffe erläutert:

- Canvas[2]
  ist das übergeordnete Koordinatensystem. Sämtliche Attribute zur Platzierung, Gestaltung und Veränderung von Objekten orientieren sich an diesem Koordinatensytem. Dieses Koordinatensystem kann auch als „Weltkoordinatensystem“ bezeichnet werden[3]. Einige weitere Details zum Begriff „Canvas“ sind dem Kapitel 8.7.1 auf Seite 119 zugeordnet, wobei einige Eigenschaften formal erläutert werden. Der wichtigste Punkt ist, dass der *Canvas* ein Koordinatensystem basierend auf den reellen Zahlen darstellt, während das AWT ein Koordinatensystem basierend auf den natürlichen Zahlen als Grundlage hat.

[1] Das Advanced Window Toolkit ist eine Klassenbibliothek zur Erzeugung von Benutzeroberflächen und Grafiken in Java, vgl. (Horstmann und Cornell, 2008a, S. 282 ff.) und (Horstmann und Cornell, 2008b, S. 521 ff.)
[2] Der Begriff *Canvas* kann als „Kompositionsfläche“ übersetzt werden.
[3] Anmerkung: Die Bezeichnung „Ein Punkt auf dem Canvas“ sei äquivalent zu „Ein Punkt im Weltkoordinatensystem“.

- Viewport
  stellt einen rechteckigen Ausschnitt des Canvas auf dem Bildschirm dar. Der Viewport bildet ein eigenes Koordinatensystem, das „Bildkoordinatensystem“. Dieser Ausschnitt wird letztendlich in Pixel umgewandelt und dargestellt. Hierfür wird das AWT eingesetzt, da dies bereits viele wichtige Funktionen zum Zeichnen implementiert. Damit dies funktioniert, muss eine Umrechnung vom Weltkoordinatensystem in des Bildkoordinatensystem durchgeführt werden. Zur Transformation zwischen Welt- und Bildkoordinatensystem sei an dieser Stelle auf (Zeppenfeld, 2004, S. 67 ff.) und (Xiang und Plastock, 2003, S. 118 ff.) verwiesen.
  Abbildung 5.1 zeigt den dreistufigen Prozess von der Erzeugung eines Objektes bis zur Darstellung.

- Origin (Ursprung) des Viewports
  ist ein Punkt im Weltkoordinatensystem und bezeichnet die linke obere Ecke des Viewports.

- Größe des Viewports (Höhe und Breite)
  sind ebenfalls Angaben im Weltkoordinatensystem. Zusammen mit dem Urspung ist bestimmbar, welcher Ausschnitt des Canvas sichtbar sein soll.

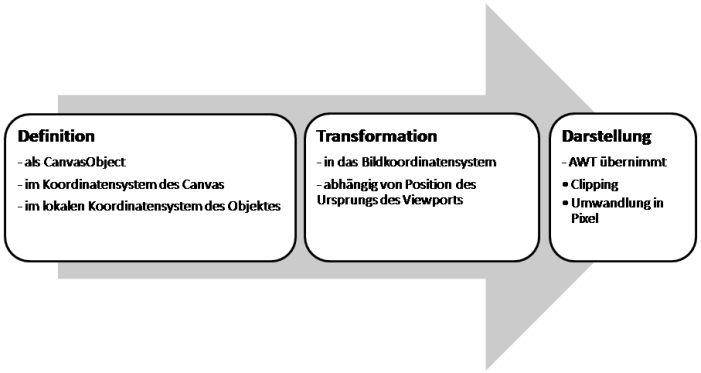


Abbildung 5.1: Prozess der Erzeugung und Darstellung eines Objektes

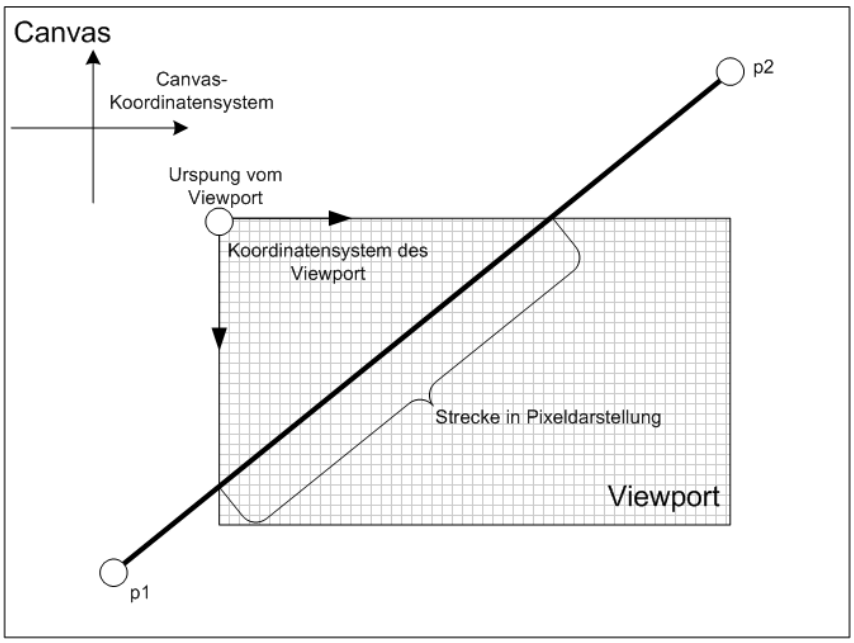


Abbildung 5.2: Darstellung einer Linie im Canvas-Koordinatensystem und im Viewport

Abbildung 5.2 zeigt einige aufgeführte Begriffe anhand eines Beispieles. Die Attribute der darzustellenden Linie (also der Startpunkt p_1 und Endpunkt p_2) befinden sich im Weltkoordinatensystem. Der Viewport stellt einen Ausschnitt auf dem Bildschirm dar und nutzt dafür das AWT.

Anmerkung:
Der Begriff „Canvas“ hat in verschiedenen Kontexten verschiedene Bedeutungen:

- Weltkoordinatensystem *Canvas*
- Klasse `Canvas` bzw. konkrete Instanz dieser
- Subsystem „2D-Canvas“

Insgesamt gesehen haben aber all diese Bedeutungen den selben Zusammenhang: Sie stellen die zweidimensionale Zeichenfläche dar, auf der Objekte beliebig platziert bzw. gezeichnet werden können. Somit ist eine gleiche Bezeichnung in verschiedenen Kontexten durchaus berechtigt.

5.3 Grundkonzepte

Insgesamt besteht der *Canvas* aus einer Reihe von Klassen, wobei bei deren Implementierung auf einfache Anwendbarkeit geachtet wurde.

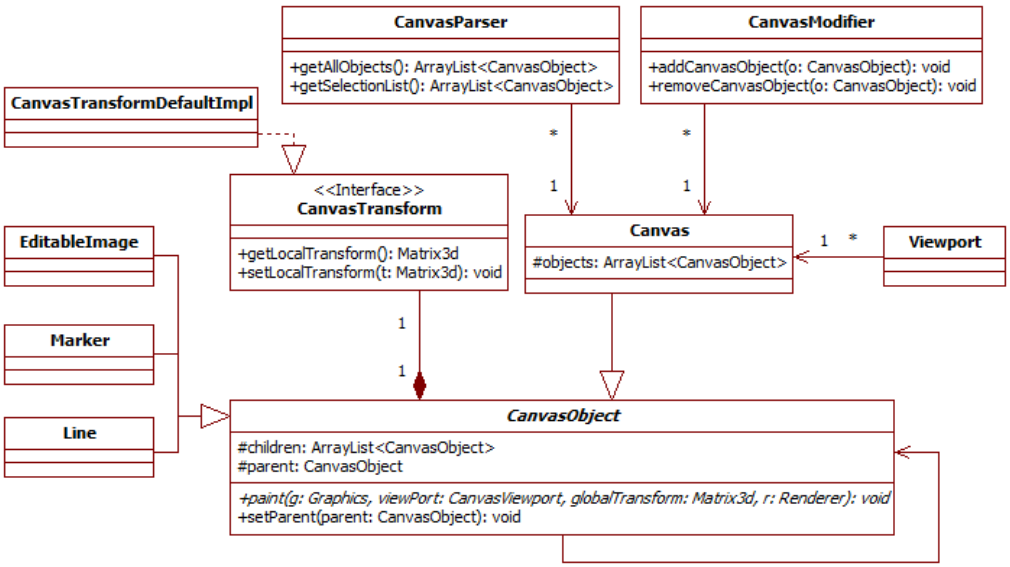


Abbildung 5.3: Hauptklassen

Abbildung 5.3 zeigt - ohne Anspruch auf Vollständigkeit - einen Überblick über die wichtigsten Klassen und Methoden.

5.3.1 Canvas

Die Klasse `Canvas` stellt den Kern des *Canvas* dar: Sie speichert Verweise auf alle Objekte, die auf dem *Canvas* dargestellt werden sollen. Die Ableitung von CanvasObject hat den Grund, dass der *Canvas* selbst z. B. ein Grundgitter zeichnen kann, welches den gleichen Regeln unterliegt wie jedes andere Objekt auf dem *Canvas*.

5.3.2 Objekte auf dem Canvas

Jedes auf dem Canvas darzustellende Objekt muss von der abstrakten Klasse *CanvasObject* ableiten. Sie stellt neben der Klasse

`Canvas` die wichtigste Klasse dar. Sie speichert die Transformation[1] des Objektes, ausgelagert in die jeweilige Implementierung des Interface `CanvasTransform`. Die Objekte auf dem *Canvas* werden intern in einem Szenegraphen gespeichert, mit dem *Canvas* selbst als Wurzel. Im einfachsten Fall hat dieser eine flache Hierachie: Jedes Objekt hat den *Canvas* als Vater. Es ist aber möglich, über die Methode

```
1 public void setParent( CanvasObject parent ){...}
```

der Klasse `Canvas` einem Objekt einen Parent zuzuweisen. Das bedeutet, dass sich die globale Transformation über die Transformationshierarchie[2] der Vorgänger-Objekte im Szenegraphen bildet.

5.3.3 Zeichnen von Objekten

Klassen, die von `CanvasObject` ableiten, müssen die abstrakte Methode

```
1 public abstract void paint( Graphics g , CanvasViewport c ,
2                             Matrix3d globalTransform, Renderer r);
```

implementieren. Diese Methode beschreibt, wie ein Objekt auf dem *Canvas* zu zeichnen ist. Nun gibt es zwei Möglichkeiten, auf dem Canvas zu zeichnen: Die erste ist, über das übergebene Objekt vom Typ *Graphics* des AWT auf Pixel basierend Elemente direkt zu zeichnen. Eine Beschreibung der nutzbaren Methoden kann aus (Horstmann und Cornell, 2008b, S. 521 ff.) entnommen werden. Zu beachten ist, dass diese Methode direkt im Bildkoordinatensystem zeichnet (also im Koordinatensystem des Viewport). Das hat zur Folge, dass solch ein Objekt unabhängig vom *Canvas* gezeichnet wird. Nutzbar ist dies, um z. B. Text oder andere Informationen immer an der gleichen Stelle im Viewport zu zeichnen.
Die zweite Möglichkeit besteht darin, die Klasse `Renderer`, welche als Parameter `r` der Methode `paint` übergeben wird, zu nutzen, um Linien oder andere Objekte direkt im Koordinatensystem des *Canvas* zu definieren. Dies kann unabhängig von dem gewählten

[1] Siehe Kap. A.2.1
[2] Vgl. Kap. A.2.4

Bildausschnitt des Viewports passieren, die notwendigen Umrechnungen übernimmt die Klasse `Renderer`. Die darin enthaltenen Methoden zeichnen im lokalen Objektraum des konkreten *CanvasObject*. Nachfolgend ein kleiner Auszug der gegebenen Methoden:

```
1 public void paintLine( Point2d start , Point2d end,
2                        double thickness, Color color )
```

zeichnet eine Linie von dem Punkt *start* zu *end* mit der Dicke *thickness* und der Farbe *color*.
Methoden mit dem Begriff *„Global“* im Namen zeichnen nicht im lokalen, sondern im globalen Objektraum, wie z. B. folgende Methode:

```
1 public void paintGlobalLine( Point2d start , Point2d end,
2                              double thickness , Color color )
```

Dies kann dazu verwendet werden, dass bestimmte grafische Elemente unabhängig von der Transformation des Objektes auf dem Canvas gezeichnet werden. Die Parameter verhalten sich äquivalent zu der voran gegangenen Methode.

```
1 public void paintText( String text , Point2d localPosition , double size )
```

zeichnet einen Text mit der Standard-Schriftart auf dem *Canvas*. Der Vorteil dieser Methode ist, dass der Text sich im gleichen Verhältnis zu allen anderen Objekten skaliert, je nachdem, ob der Anwender ein- oder auszoomt. Ebenso hat die Transformation des Objektes eine direkte Auswirkung: Der Text lässt sich somit leicht skalieren. Somit ist es möglich, Beschriftungen einzufügen, die sich verhalten wie jedes andere konkrete *CanvasObject*.
Eine Umrechnung vom Bildschirmkoordinatensystem in das Canvas-Koordinatensystem und zurück bietet das Interface `CanvasViewport`, welches ebenfalls als Parameter an die Methode `paint` übergeben wird. Wichtig sind die folgenden beiden Methoden:

```
1 //rechnet einen gegebenen Wert in Pixel des Bildkoordinatensystems um
2 public int getDistanceInPixel(double value);
3 //rechnet einen Punkt vom Canvas-Koordinatensystem
4 //in des Bildkoordinatensystem um
5 public Point getPoint2dAsPixel(Point2d p)
```

5.3.4 CanvasModifier und CanvasParser

Direkte Anpassungen des *Canvas* (über die konkrete Instanz der Klasse `Canvas`) sind nicht zugelassen. Hierfür müssen die beiden Klassen `CanvasModifier` und `CanvasParser` eingesetzt werden. Sie enthalten alle notwendigen Methoden zum Verändern und Abfragen des *Canvas*.
Über den CanvasModifier können Objekte hinzugefügt und entfernt werden, mit den Methoden:

```
1 //fügt ein Objekt dem Canvas hinzu
2 public void addCanvasObject( CanvasObject o ) {...}
3 //entfernt ein Objekt vom Canvas
4 public void removeCanvasObject( CanvasObject o , boolean removeChildren ) {...}
```

Abfragen des *Canvas* können über `CanvasParser` durchgeführt werden. Z.B. ist eine Liste der selektierten Objekte auslesbar über

```
1 //gibt alle selektierten Objekte zurück
2 public ArrayList<CanvasObject> getSelectionList() {...}
```

5.3.5 Selektionen

Das Interface `Selectable` definiert unter anderem die Methode

```
1 public boolean isSelected( Point mouseClickScreenSpace ,
2                            Point2d mouseClickCanvasSpace ,
3                            CanvasViewport v );
```

und muss durch Klassen, die von `CanvasObject` ableiten, implementiert werden. Hierbei ist für jedes Objekt zu entscheiden, ob ein Mausklick durch den Anwender, welcher durch den Parameter `mouseClickCanvasSpace` repräsentiert wird, als „Objekt wurde selektiert“ eingestuft wird. Entsprechend gibt in diesem Fall die Methode ein `true` zurück. Somit können unabhängig von der Darstellung des Objektes bestimmte Bereiche des *Canvas* zur Selektion genutzt werden - dies ist von Objekt zu Objekt neu zu implementieren.

5.3.6 Observer

Es gibt zwei wichtige Interfaces, welche ein Observer implementieren muss, um auf bestimmte Ereignisse reagieren zu können:

- `CanvasSceneObserver`
  wird bei einer Veränderung des Szenegraphen benachrichtigt.
- `SelectionListObserver`
  erhält eine Benachrichtigung, falls z. B. ein Objekt selektiert oder deselektiert wurde.

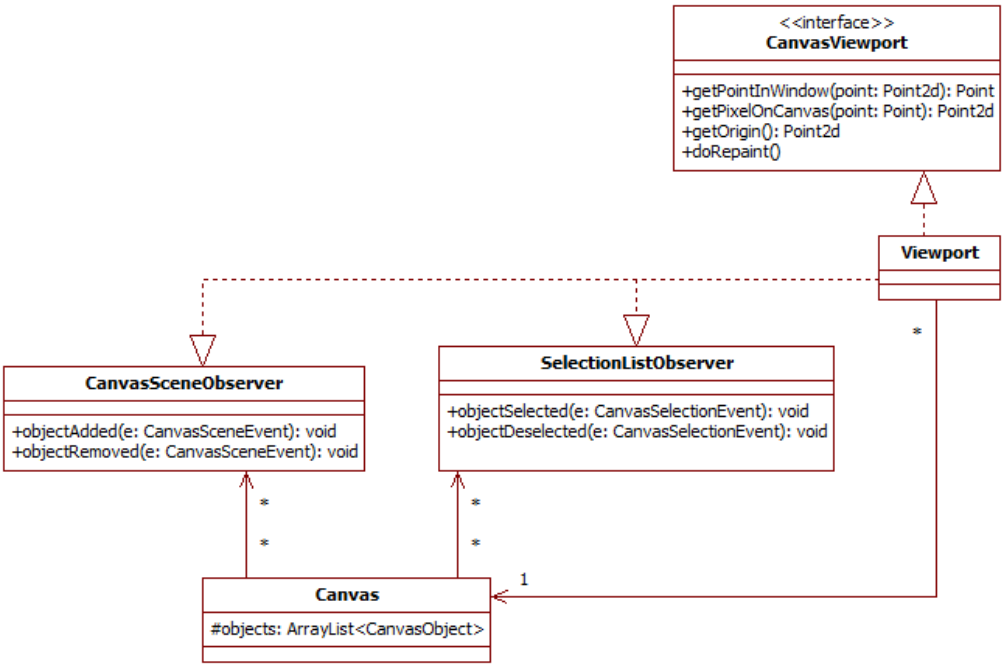


Abbildung 5.4: Observerklassen

Abbildung 5.4 zeigt diese Interfaces im Zusammenhang. Es ist zu sehen, dass die Klasse `Viewport` selbst diese Interfaces implementiert und auf diese Ereignisse reagiert. Beispielsweise wird der Viewport neu gezeichnet, wenn ein Objekt selektiert wurde.
Instanzen, welche diese Interfaces implementieren, müssen beim Canvas angemeldet werden, da dieser die Selektionsliste und die Liste aller Objekte enthält und über alle Informationen entsprechender Änderungen verfügt.

5.4 Funktionalität

Nachfolgend sei ein Überblick über die Funktionalität des implementierten Systems gegeben:

- Freies Bewegen des Viewports
  Das ist möglich durch das Drücken der Shift-Taste und Klicken + Ziehen.
- Zoom über Mausrad
  Zooming wird beschleunigt durch Strg-Taste, verlangsamt durch Shift-Taste.
- Selektion von Objekten
  Das ist nur möglich, sofern dies für das jeweilige Objekt implementiert ist.
- Grid-Snapping[1]
  Dies wird aktiviert durch Drücken und Halten der Taste „X“
- HUD[2]
  zeigt verschiedene, anpassbare Informationen direkt im Viewport.
- Wahlweise Anzeige eines Grundgitters
- Direktes Zeichnen im Koordinatensystem des Canvas,
  Clipping, Zoom und Bewegung werden automatisch berechnet.
- Mehrere Viewports ein und des selben Canvas
  ermöglicht mehrere Sichten zugleich.
- Bewegung von Objekten
  Ist ein Objekt bewegbar, kann der Anwender es durch Klicken und Ziehen verschieben.
- Transformationshierarchien
  Dadurch können Objekte in eine Vater-Kind-Beziehung eingebracht werden. Das hat zur Folge, dass komplexe Objekte aus mehreren Teilobjekten zusammengesetzt werden können, ebenso ist eine Gruppierung möglich.

[1] Grid-Snapping ist das Einrasten von beliebigen Koordinaten in diskrete Schritte anhand eines darunter liegenden Gitters.
[2] Heads Up Display

Zu beachten ist, dass dies keinesfalls alles abdeckt: Viele kleinere Funktionen erhöhen weiterhin die Funktionalität.

5.5 Fallbeispiel: Bildbetrachter

Um die einfache Initialisierung und Verwendung zu demonstrieren, sei an dieser Stelle ein kleines Fallbeispiel erläutert. Hierbei ist das Ziel, einen einfachen Bildbetrachter für Bilder des Formates JPG zu implementieren. Das Klassendiagramm der Abbildung 5.5 zeigt die Klassenhierarchie. `PicViewerMainWindow` bildet das umgebende Fenster. Die abstrakte Klasse `GenericEditorPanel` ist eine Hilfestellung bei der Initialisierung: Ein Ableiten von dieser Klasse erzeugt bereits einen voll funktionstüchtigen *Canvas*. Allerdings müssen vorher ein paar Klassen initialisiert und übergeben werden.

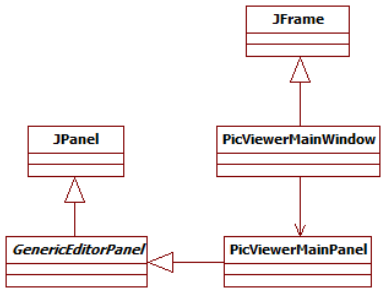


Abbildung 5.5: Klassenbeziehungen des Fallbeispieles

```
1  public PicViewerMainWindow(ImageHandler imageHandler) {
2    Canvas canvas = new Canvas(true, new CanvasTransformDefaultImpl());
3    CanvasParser parser = new CanvasParser(canvas);
4    DefaultCanvasModifier modifier = new DefaultCanvasModifier(canvas);
5    GlobalOptions o = new GlobalOptions();
6    HUD hud = new HUD();
7    DefaultMovementHandler h = new DefaultMovementHandler(canvas , o);
8    Viewport viewport = new Viewport(modifier, parser, o , hud , h);
9    PicViewerMainPanel panel =
10     new PicViewerMainPanel(parser, modifier, h , o , hud, viewport);
11
12   EditableImage img =
13     new EditableImage(imageHandler , new CanvasTransformDefaultImpl()) {
14
15     @Override
16     public boolean isMoveable() {
17
```

```
18        return false;
19      }
20
21      @Override
22      public boolean isSelected( Point mouseClickScreenSpace ,
23                                 Point2d mouseClickCanvasSpace ,
24                                 CanvasViewport v ) {
25        return false;
26      }
27    };
28
29    modifier.addCanvasObject( img );
30
31    this.setSize( 400 , 400 );
32    this.add( panel );
33  }
```

Listing 5.1: Beispielhafte Initialisierung eines 2D-Canvas

Listing 5.1 zeigt den Konstruktor der Klasse `PicViewerMainWindow`. Die Klasse `ImageHandler` ist eine Hilfsklasse, welche ein `BufferedImage` kapselt. In den Zeilen 2 bis 8 werden alle notwendigen Objekte initialisiert. Es gibt bereits Standardimplementierungen für die wichtigen abstrakten Klassen, sodass diese gleich verwendet werden können. Die Klassen `GlobalOptions` und `HUD` sind Klassen zur Übermittlung globaler Parameter bzw. zum Anzeigen von Informationen auf dem Viewport. Nach der Initialisierung kann eine Instanz der Klasse `PicViewerMainPanel` erzeugt werden. Diese Klasse übernimmt die eigentliche Darstellung des Viewports und kann, da sie von `JPanel` ableitet, in jeder Swing-GUI ohne weiteres eingesetzt werden. Es ist sogar möglich, mehrere Instanzen zu initialisieren, um somit mehrere Sichten auf ein und dieselben Objekte zu erlauben.
In der Zeile 12 wird eine Instanz von `EditableImage` erzeugt, welche es ermöglicht, ein Bild auf dem Canvas anzuzeigen. An diese wird der Parameter `imageHandler` übergeben, sodass dieses Bild angezeigt werden kann. Das Überschreiben der beiden Methoden in den Zeilen 16 und 22 sorgt dafür, dass das Bild weder selektier- noch bewegbar ist.
Anschließend muss das Objekt noch auf dem *Canvas* platziert werden, hierfür wird die Methode `addCanvasObject()` der Klasse `CanvasModifier` verwendet. Die letzten Schritte in den Zeilen 31 und 32 setzen die Größe des Fensters und fügen das eben erzeugte `JPanel` hinzu.

```
1 public static void main(String[] args) {
2   PicViewerMainWindow w = new PicViewerMainWindow(
3     new ImageHandler("D:/work/test.jpg"));
4   w.setSize( 300 , 300 );
5   w.setVisible( true );
6 }
```

Listing 5.2: Implementierung der Methode `main`

Schlussendlich ist nur noch `PicViewerMainWindow` zu initialisieren und anzuzeigen, Abbildung 5.6 zeigt das Resultat der Implementierung, nachdem über eine main-Methode in Listing 5.2 ein Bild übergeben wurde.

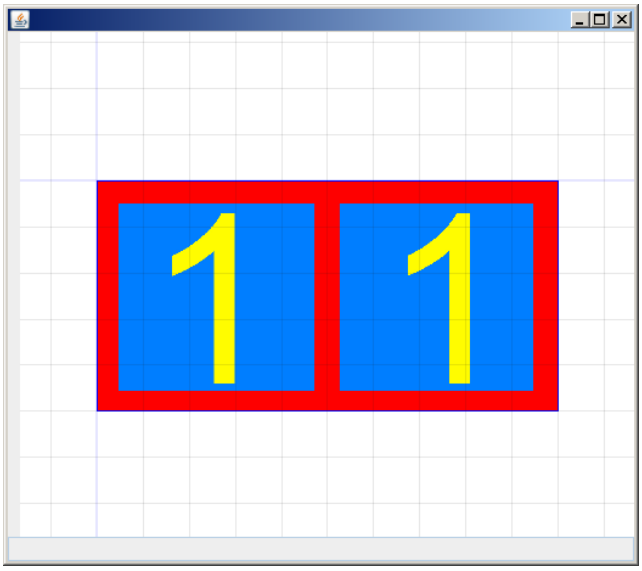


Abbildung 5.6: Beispiel einer einfachen Implementierung des Canvas

5.6 Fazit

Resultat der Überlegungen und anschließender Implementierung ist ein System, welches leicht einzusetzen ist und für alle 2D-Darstellungen von Grafiken genutzt werden kann. Darüber hinaus lassen sich leicht Erweiterungen implementieren, sodass die Funktionalität beliebig erweitert werden kann. Neben der Erweiterbarkeit ist die einfache Initialisierung ein wichtiger Punkt, was den 2D-Canvas zu einem vielseitigen und leicht einzusetzenden Werkzeug macht.

6 ArchiViz als Erweiterung von GroIMP

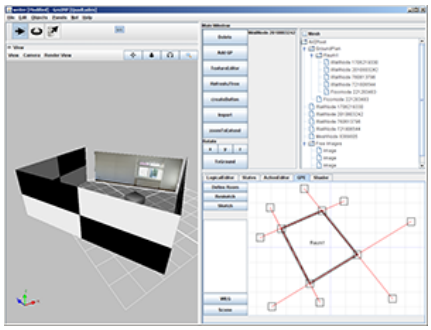


Abbildung 6.1: Das Plugin ArchiViz

Im Rahmen dieser Arbeit entstand ein Plugin für GroIMP mit dem Namen *ArchiViz*. GroIMP ist als ein auf Plugins basierendes System konzipiert, sodass das Hinzufügen weiterer Plugins zur Erweiterung der Funktionalität relativ einfach ist. Die folgenden Ausführungen basieren auf den in Kapitel 4 vorgestellten Konzepten. Auf diese wird an dieser Stelle nicht erneut eingegangen.
Ziel dieses Kapitels ist die Verdeutlichung der Mechanismen des Plugins *ArchiViz*, welches - mit GroIMP als Plattform - die in Kapitel 2.3 definierten Anforderungen umsetzt. *ArchiViz* ist ein wichtiger und elementarer Bestandteil dieser Arbeit. Es ist so konzipiert, dass es als Grundlage für weitere Arbeiten dienen kann.
Zunächst wird in Abschnitt 6.1 geklärt, welche softwaretechnische Struktur dem Plugin zugrunde liegt. Darüber hinaus nutzt ArchiViz einen relativ komplexen Szenegraph. Dessen Struktur wird in Abschnitt 6.2 näher beleuchtet. Die anschließenden Abschnitte befassen sich mit einigen ausgewählten Punkten des Plugins und erläutern entsprechende Details genauer.
Insgesamt betrachtet ist über das Plugin ArchiViz ein Großteil der in dieser Arbeit vorgestellten Konzepte umgesetzt worden. Es dient also als möglicher Ausgangspunkt für weitere Arbeiten und Konzepte, die sich unter anderem auf Anforderungen der Architektur beziehen. Derzeit ist ArchiViz ein Werkzeug, dass es erlaubt, einen gesamtheitlichen Arbeitsfluss durchzuführen: Angefangen bei der Bearbeitung von Fotografien zu Texturen für Innenwände (Texturedi-tor), über das Erzeugen von Innenwänden (Grundrisseditor), dem Zuweisen von Texturen (Shadereditor), Erzeugen und Zuweisen von

Aktionen und Reaktionen (Aktions-Reaktions-Editor), Verwalten und Erzeugen von verschiedenen Zuständen bis hin zur Durchführung der abschließenden Präsentation bauen alle Konzepte auf GroIMP als Plattform auf und werden über das Plugin ArchiViz eingebunden bzw. zur Verfügung gestellt. Einzig der Textureditor ist auch separat nutzbar, da die Anbindung an GroIMP bzw. ArchiViz lose gestaltet wurde.

6.1 ArchiViz - Basisarchitektur

Das Plugin ArchiViz wurde von Grund auf modular entwickelt. Das bedeutet, dass bestimmte Einzelteile separat entwickelt und getestet wurden.
Dennoch gab es eine Besonderheit an diesem Vorgehen: Einige Funktionalitäten, nämlich jene, die direkt Auswirkungen auf GroIMP haben, konnten ohne die Integration in GroIMP auch nicht implementiert werden. Darüber hinaus benötigten einige Module den Szenegraphen von GroIMP, was ebenso dazu führt, dass diese ohne GroIMP nicht implementiert werden konnten.
Somit wurden einige wichtige Elemente zuvor separat implementiert, bis eine Basisfunktionalität erreicht wurde. Anschließend wurden diese Elemente in GroIMP integriert und die Implementierung vervollständigt.

6.1.1 Module

ArchiViz besteht aus zwei Modi: dem Präsentationsmodus und dem Editormodus. Beide unterscheiden sich hinsichtlich der GUI und dem Verhalten GroIMPs, d. h. die Interaktion zwischen Plugin und GroIMP sind verschieden.
Abbildung 6.2 zeigt eine grobgranulare Übersicht der einzelnen Module des Plugins. Die Pfeile stellen nicht zwingend die Sichtbarkeit (A kennt B) dar, sondern eher den Verlauf des Steuerflusses. Es ist zu erkennen, dass beispielsweise der häufig genutzte Canvas und der Textureditor außerhalb liegen. Beide sind unabhängig von GroIMP nutzbar.
Zwischen den Modulen gibt es keine direkte Kommunikation. Alle beziehen ihre Daten direkt über den Kern (und somit aus dem Szenegraphen von GroIMP).

Zentrale Elemente sind GroIMP und der ArchiViz-Kern, bestehend

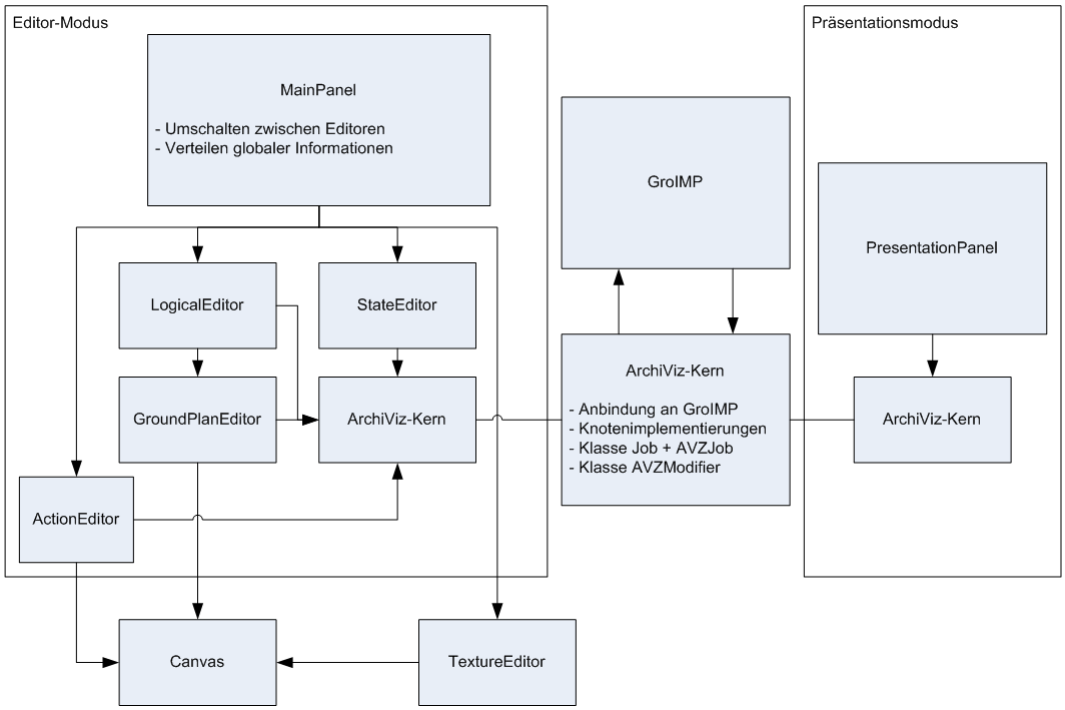


Abbildung 6.2: Module

aus mehreren Klassen. Es wurde darauf Wert gelegt, dass die einzelnen Module nie direkt mit GroIMP kommunizieren, sondern immer über den Kern. Dieser Kern besteht aus folgenden Klassen:

- `Job` und `AVZJob`
  Die Klasse `Job` ist bereits in Kap. 4.2.5 auf Seite 46 beschrieben worden. Die Klasse `AVZJob` erbt davon und implementiert notwendige Schritte, welche nach jedem `Job` ausgeführt werden müssen.

- `AVZModifier`
  Das Hinzufügen und Entfernen von Knoten sowie die Veränderung von Kanten findet ausschließlich über die Klasse `AVZModifier` statt. Diese kapselt einige Methoden von GroIMP, sodass der Zugriff auf ArchiViz zugeschnitten und somit vereinfacht wird. Dazu gehört, dass implizit bestimmte Klassentypen, z. B. `AVZNode` angenommen werden.

- `AVZParser`
  Der lesende Zugriff kann über diese Klasse stattfinden. Es ist gefiltertes Parsen des Graphen möglich, entsprechend Listing 4.1 auf der Seite 44, darüber hinaus sind einige speziellere Parsing-Funktionen enthalten, die ArchiViz benötigt.

- `Init`
  Diese Klasse meldet, ausgeführt über einen *Hook* (Vgl. 4.7.5, S. 62), die notwendigen Listener an.

- `AVZCore`
  Einige wichtige Instanzen von Klassen wie z. B. `AVZModifier` werden in dieser Klasse global gespeichert und stehen somit dem gesamten Plugin zur Verfügung.

6.2 Der ArchiViz-Szenegraph in GroIMP

Ein großer Teil der Logik in ArchiViz wurde auf den Szenegraphen in GroIMP ausgelagert. Das bedeutet, dass viele Entscheidungen des Programmes abhängig von bestimmten Knoten oder Kanten im Graphen sind. Es wurden eine Reihe von Knoten implementiert, welche verschiedene Aufgaben erfüllen. Alle Knoten wurden mit dem *enhance*-Skript (Vgl. Kap. 4.7.1 auf S. 58) verarbeitet, sodass die Variablenbelegungen aller Knoten in einer Szene gespeichert werden können.
Neben den implementierten Knoten nutzt ArchiViz das Kantenkonzept von GroIMP (Vgl. Kap. 4.2.1 auf S. 39), sodass auch über die Unterscheidung von Kanten durch den Graph iteriert werden kann. Diese Form des Parsens wurde allerdings weniger eingesetzt: Eine Unterscheidung über die Knotentypen und die Wahl von zwei Kantentypen, nämlich die *successor*- und eine Nicht-*branch*- bzw *successor*-Kante, erwies sich als ausreichend. Letztere wird im Folgenden als *AVZEdge* bezeichnet.
Weiterhin wurden *AVZEdges* dazu genutzt, besondere Beziehungen zwischen den Knoten zu modellieren: Somit besteht eine Innenwand aus Anfangs- und Endpunkt, welche selbst Knoten sind. Eine entsprechende Kante stellt diese Beziehung dann dar.

6.2.1 Wichtige Knotentypen

Basisknoten

Folgende Knotentypen bilden die Basis des ArchiViz-Szenegraphen:

- `AVZNode` ist die abstrakte Basisklasse, von der *alle* Knoten in ArchiViz erben. Unter anderem enthält die Klasse die Berechnung einer eindeutigen ID und die Zuweisung der Klassentypen (siehe weiter unten). Darüber hinaus bietet `AVZNode` einige lesende Zugriffe auf den Graphen, welche sich auf die direkten Nachbarn beziehen. Beispiele sind:

```
1 public <T extends AVZNode> AVZNodeList<T> getDirectChildrenByType( int type ) {
2   ...
3 }
```

```
1 public <T extends AVZNode> AVZNodeList<T> getParentsByType( NodeType type ) {
2   ...
3 }
```

  Weiterhin stehen Methoden zum Lesen und Schreiben der lokalen und globalen Transformation zur Verfügung.

- `AVZPointNode` ist die Basisklasse für die Speicherung und Darstellung einzelner Punkte. Diese ist notwendig, da beispielsweise die Schnittpunkte zwischen *Sketchinglines* als Punkt dargestellt und gespeichert werden. Damit diese editierbar bleiben, musste dafür eine Repräsentation im Szenegraphen implementiert werden. Ebenso konnte so eine direkte Verbindung zwischen dem Canvas und GroIMP geschaffen werden, sodass eine Veränderung des Punktes sich sowohl im 2D- als auch im 3D-Raum auswirken kann.

- `AVZRootNode` ist die Wurzel aller Knoten aus ArchiViz. Damit der ArchiViz-Szenegraph logisch vom Rest getrennt ist und sich somit z. B. das Parsen erleichtert, hängen alle Knoten in ArchiViz unter diesem Knoten.

- `AVZRoomNode` gruppiert mehrere Knoten zu einem Raum.

- `AVZNullNode` gruppiert mehrere Knoten vom Typ `AVZMeshNode`.

Bilder

Einige Module des Plugins haben mit der Bearbeitung von Bildern zu tun, ebenso soll bearbeiteten Bildern Geometrie zugewiesen werden. Hierfür bilden folgende Knoten die Basis:

- `AVZGroundPlanNode` beinhaltet die Referenz auf einen Grundriss als Bild und gruppiert zugleich Räume. Das enthaltene Bild kann direkt zugewiesen oder durch den Textureditor erzeugt werden.
- `AVZImageNode` enthält den Verweis auf einen Bild-Dateipfad. Der Pfad ist relativ, ausgehend von der Projektdatei.
- `AVZImageContainerNode` ist der Wurzelknoten neu importierter Bilder.
- `AVZImageSetNode` sammelt Image-Knoten unter einem gemeinsamen Knoten, damit diese als Gruppe in den Textureditor geladen werden können. Zugleich enthält dieser Container das Ergebnis des Textureditors.

Geometrie

Geometrieknoten enthalten die notwendigen Datenstrukturen, um Polygone (hier beschränkt auf Dreiecke) zu speichern. Es gibt zwei Arten von Geometrieknoten in ArchiViz: Entweder wird die Geometrie fest implementiert (z. B. bei importierter Geometrie) oder die Geometrie wird algorithmisch bestimmt und während der Laufzeit von GroIMP variiert.

- `AVZMeshNode` ist die Repräsentation eines polygonales Objektes. Um Objekte darstellen zu können, welche aus Polygonen bestehen (z. B. Wände oder importierte Objekte), aber auch um bestimmte ArchiViz-Daten zu halten, musste eine eigene *Mesh*-Klasse implementiert werden. Der Code besteht zu großen Teilen aus kopiertem GroIMP-Code, da alle Knoten in ArchiViz von `AVZNode` ableiten müssen. Einmal gespeicherte Geometrie wird während der Laufzeit von GroIMP nicht geändert.
- `AVZWallNode` enthält und berechnet die polygonale Geometrie einer Innenwand. Die Geometrie wird dynamisch zur Laufzeit neu berechnet.

- `AVZFloorNode` und `AVZCeilingNode` generieren polygonale Geometrie des Bodens bzw. der Decke eines Raumes. Die Geometrie wird dynamisch zur Laufzeit von GroIMP neu berechnet.

Aktionen und Reaktionen

- `AVZActionNode` und `AVZReactionNode` enthalten eine Sammlung definierter Aktions- bzw. Reaktionsobjekte, welche mit Aktionen bzw. Reaktionen in GroIMP bzw. Swing assoziiert sind.

Sketchingknoten

- `AVZIntersectionPointNode` stellt einen Schnittpunkt des Sketchings im Grundrisseditor dar und speichert diesen.
- `AVZSketchingLineNode` ist eine durch den Anwender gezogene *Sketchingline*.
- `AVZSketchingLinePointNode` definiert einen Anfangs- bzw. Endpunkt einer *Sketchingline*!Anfangs- und Endpunkt.
- `AVZSketchingWallNode` stellt einen Abschnitt einer *Sketchingline* zwischen zwei Schnittpunkten dar. Er gruppiert zwei `AVZIntersectionPointNode`.

Zustandsknoten

- `AVZStateServerNode` gruppiert alle Zustandsknoten.
- `AVZGlobalStateNode` stellt einen globalen Zustand dar.
- `AVZStateEntryNode` ordnet ein Objekt einem bestimmten Zustand zu und speichert die entsprechenden Attribut-Werte, die zu diesem Zustand gehören.

6.2.2 Knotenbeziehungen

Allgemein besteht der Szenegraph in ArchiViz aus zwei Ebenen: Zum einen handelt es sich um einen Baum, aufgespannt durch die *successor*-Kanten, und zum Anderen um einen Graphen, aufgespannt durch die restlichen *AVZEdges*. Zu beachten ist, dass in Wahrheit in GroIMP nur eine einzige Kante zwischen zwei Knoten existiert und

die Unterscheidung über die Kantenmarkierung geführt wird.
Darüber hinaus wird der ArchiViz-Szenegraph vom „normalen“ GroIMP-Szenegraph unterschieden, indem ArchiViz sich allein in einem Subgraph, mit `AVZRoot` als Wurzelknoten, aufhält.
Abbildung 6.3 zeigt die Struktur des ArchiViz-Szenegraphen. Die Kantenmarkierungen *s* und *e* deuten die Bit-Belegung der Kanten an: *s* steht für *successor*-Edge und *e* für *AVZEdge*. Insgesamt kann

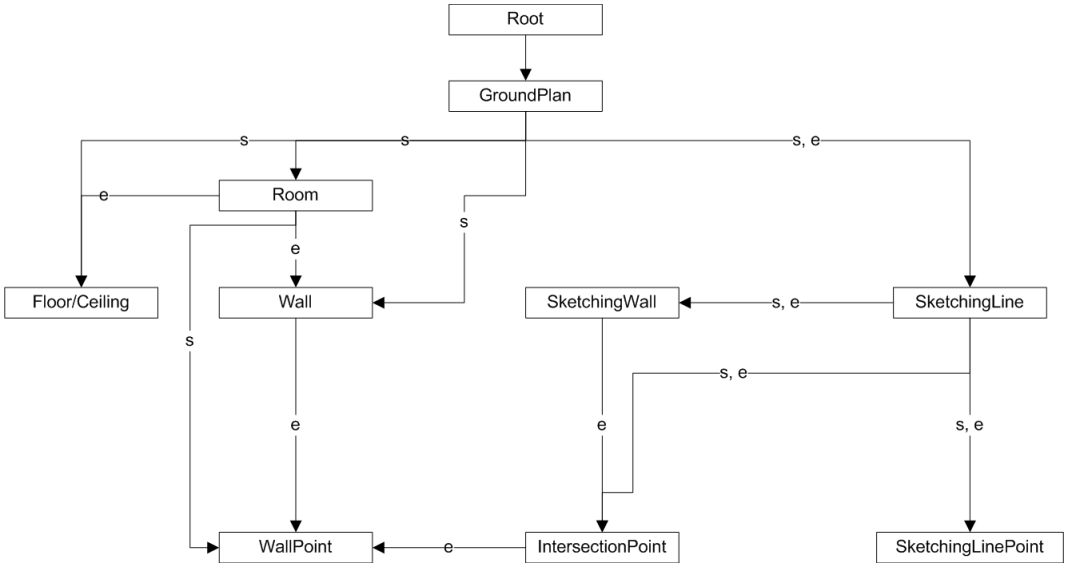


Abbildung 6.3: Struktur des Szenegraphen

auf Grund dieser Struktur der Szenegraph recht komplex werden. Jedoch muss der Anwender diesen nicht direkt handhaben, alles wird durch entsprechende Werkzeuge gesteuert.

6.2.3 Sichtbarkeit von Knoten

Eine etwas größere Schwierigkeit bestand darin, einzelne Objektknoten in GroIMP dem Renderprozess im Echzeitfenster zu entziehen, also, einfach gesprochen, zu verstecken. In GroIMP war bis zu diesem Zeitpunkt keine Funktionalität dafür vorgesehen, sodass eine Umgehung entwickelt werden musste.
Grundsätzlich spielt bei eben dieser Umgehung folgender Fakt die entscheidende Rolle: Es werden nur Knoten im 3D-Fenster angezeigt, zu denen genau ein Pfad aus *successor*-Kanten durch den Szenegraphen führt. Ist dieser Pfad nicht vorhanden, kann das Objekt

auch nicht angezeigt werden. Nun ist es möglich, Knoten nur über beliebige andere Nicht-*successor*-Kanten in den Graphen GroIMPs einzubinden. Genau dieser Umstand konnte genutzt werden, um Objekte im 3D-Fenster zu verstecken.
Hierbei wird die zu dem Objekt führende *successor*-Kante ersetzt durch eine *visibility*-Kante[1]. Somit bleibt das Objekt strukturell im Graphen erhalten, allerdings wird es beim Aufbauf der 3D-Szene übergangen und nicht dargestellt. Das Wiedereinblenden funktioniert entsprechend analog, wobei die *visiblity*-Kante wieder durch eine *successor*-Edge ersetzt wird.
An dieser Stelle muss betont werden, dass dieses Verstecken nur im Kontext von ArchiViz getestet wurde. Ein günstigerer Weg wäre ein anpassbares Attribut innerhalb eines jeden Knotens, dessen Wert die Sichtbarkeit für das 3D-Fenster festlegt.

6.3 Projektverwaltung

Ein wichtiger Punkt in der Zusammenarbeit zwischen ArchiViz und GroIMP ist die Projektverwaltung. Hierbei sind einige Dinge von Bedeutung:

- Die Speicherung eines Projektes muss vor dem Import von Bildern durchgeführt worden sein, damit der absolute Projektpfad bestimmbar ist. Somit kann ein relativer Pfad zur Projektdatei erzeugt und gespeichert werden.

- Bilder werden nur über ihren relativen Pfad zur Projektdatei referenziert und auch nur dieser Pfad in den entsprechenden Knoten gespeichert. Der Grund ist: In ArchiViz werden relativ viele Bilder zeitgleich verwendet, und diese zusätzlich in recht hoher Auflösung. GroIMP hat die Eigenschaft, dass Bilddateien, die mit dem Projekt verbunden werden (durch entsprechende Befehle in GroIMP), sofort in den Speicher geladen werden. Wenn man bedenkt, dass schon bei zwei Räumen mit jeweils vier Innenwänden und insgesamt zwei Zuständen 16 Texturen zu laden sind, ist bei der 32-Bit-Version von Java schnell die Grenze des adressierbaren Hauptspeichers erreicht. Um dieser Belastung aus dem Weg zu gehen, werden nur jene Texturen

[1] Die Namensgebung ist an dieser Stelle nicht von Bedeutung, grundlegend wird der Kante eine verschiedenartige Markierung hinzugefügt, sodass sie eindeutig identifizierbar ist.

in den Speicher geladen, die auch wirklich sichtbar sind. Ein relativer Pfad wird deshalb benötigt, damit das Projekt mitsamt seinen Bilddateien systemübergreifend weiter gegeben werden kann.

6.4 Interaktion zwischen dem ArchiViz-Canvas und GroIMP

6.4.1 Darstellung der Knoten in 2D

Da der *Canvas* für die Darstellung von 2D-Elementen mit einem eigenen Szenegraphen arbeitet und die Konzepte sich von denen von GroIMP unterscheiden, musste eine Verbindung zwischen Knoten des Szenegraphen von GroIMP und dem *Canvas* geschaffen werden. Diese Verbindung konnte relativ fest in die Knoten von ArchiViz integriert werden. Das geschah durch eine Erweiterung der Klasse `AVZNode`. Da alle anderen Knoten des Szenegraphen von ArchiViz von `AVZNode` ableiten, schuf dies zugleich eine Grundlage für alle anderen Knoten. Hierfür muss jeder Knoten eine Instanz vom Typ `CanvasObject` erzeugen können. Ist diese Instanz `null`, wird der Knoten beim Zeichnen ignoriert. Auf diese Art kann für jeden Knoten separat implementiert werden, in welcher Art und Weise der Knoten in 2D dargestellt wird. Ebenso stehen bei der Implementierung alle bereits implementierten Objekte des *Canvas* zur Verfügung. Um folgende Felder und Methoden wurde `AVZNode` erweitert:

```
1  //Feld zum Speichern des erzeugten CanvasObject
2  protected CanvasObject canvasObject;
3
4  //erzeugt ein CanvasObject initial
5  //kann leere Implementierung enthalten
6  protected abstract CanvasObject createCanvasObject();
7
8  //gibt ein CanvasObject zurück, kann null zurück geben
9  public CanvasObject getCanvasObject() {...};
10
11 //aktualisiert das Aussehen des CanvasObject in lokalen
12 //Canvas-Koordinaten
13 protected void updateCanvasObject( final CanvasObject o ){};
```

Listing 6.1: Erweiterung der Klasse `AVZNode`

Die ursprüngliche Verfahrensweise, wie der *Canvas* die Objekte zeichnet, musste erweitert werden, da nun die 2D-Objekte nicht di-

rekt in den *Canvas* eingefügt werden, sondern durch die Knoten in ArchiViz zur Verfügung gestellt werden. Es gab mehrere Ansätze:

- Bei jedem Zeichnen des *Canvas* wird der Szenegraph GroIMPs, beginnend beim Wurzelknoten von ArchiViz, komplett durchlaufen und - sofern nicht *null* zurück gegeben wird - über die Methode `getCanvasObject()` ein `CanvasObject` in den *Canvas* eingetragen.

- Es wird nur zu bestimmten Zeitpunkten der Graph durchlaufen, z. B. wenn ein *Listener* eine Änderung des Graphen meldet.

Letztendlich wurde Variante 2 implementiert, was einen erheblichen Effizienzvorteil bot. Insbesondere bei schnell hintereinander auftretenden Events reichte die Geschwindigkeit bei Variante 1 nicht aus, während bei Variante 2 der Graph bei weitem seltener durchlaufen werden muss.
Insgesamt spielen zwei Klassen eine wichtige Rolle:

- `GPEViewport`
  leitet von der Klasse `Viewport` des *Canvas* ab und erweitert diesen so, dass ein Repaint durch ein Mouse-Release-Event ausgelöst und in einem Job ausgeführt wird. Darüber hinaus werden Selektionen im Canvas an GroIMP übertragen.

- `GPECanvas`
  leitet von der Klasse `Canvas` ab und verwaltet die Verbindung zwischen GroIMP-Knoten und Objekten auf dem *Canvas*. Dies wird über zwei Hashmaps realisiert: Somit ist die Verbindung bidirektional[1]. Diese Hashmaps werden regelmäßig aktualisiert (Vgl. Kap. 6.4.2).

Die Abbildung 6.4 zeigt die Vererbungshierarchie. Durch die strikte Trennung des urspünglich implementierten Prinzips des *Canvas* und der Integration in GroIMP sind beide Prinzipien getrennt einsetzbar: Zum einen kann der *Canvas* in von GroIMP unabhängige Fenster oder andere Anwendungen integriert werden, zum anderen ist ebenso eine direkte Interaktion mit dem GroIMP-Szenegraphen möglich.

[1] Ein `CanvasObject` kann durch einen gegebenen Knoten bestimmt werden und umgekehrt

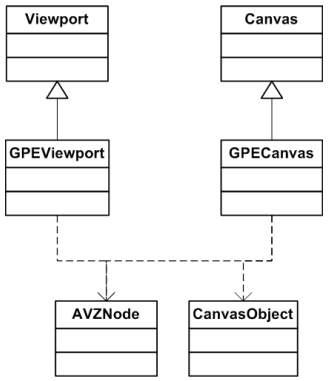


Abbildung 6.4: Klassendiagramm um die Klassen `GPEViewport` und `GPECanvas`

6.4.2 Bekanntmachung von Änderungen

Eine der größten Herausforderungen bestand darin, Änderungen des Szenegraphen in beiden Richtungen, also vom *Canvas* in ArchiViz zu GroIMP und umgekehrt, bekannt zu machen. Es soll gewährleistet sein, dass die in 2D dargestellten Objekte konsistent zum Szenegraphen von GroIMP sind. In erster Linie sollen allerdings direkte Veränderungen von ArchiViz-Objekten nur über den *Canvas* gestattet sein. Somit beschränkt sich die Richtung von GroIMP nach ArchiViz auf das Löschen und Hinzufügen von Knoten, während die andere Richtung zusätzlich Änderungen von Attributen (z. B. Transformationen) einbeziehen muss.
Abbildung 6.5 zeigt ein Sequenzdiagramm, beginnend bei einem *MouseRelease*-Event in einer Instanz von `GPEViewport` (nicht in Abbildung dargestellt). Das Vorkommen zweier Instanzen von `GPECanvas` soll andeuten, dass die Anzahl der `GPECanvas`-Instanzen keine Rolle spielt. Wichtig ist: Ein `GPECanvas` (in der Abbildung der linke) ist einem `GPEViewport` zugeordnet.

- `commitChanges()`
  überträgt die Änderungen von Positionen der 2D-Objekte an den Szenegraphen. Sie nutzt die in Kap. 6.4.1 erwähnten *HashMaps*, um diese Änderungen zuzuordnen. Dieser Aufruf findet nur auf dem `GPECanvas` des aktiven *Viewports* statt, denn nur dort können Veränderungen eingetreten sein[1].

[1]Dies wird implizit angenommen, da das *Mouse-Release*-Event nur von einem `GPEViewport` stammen kann.

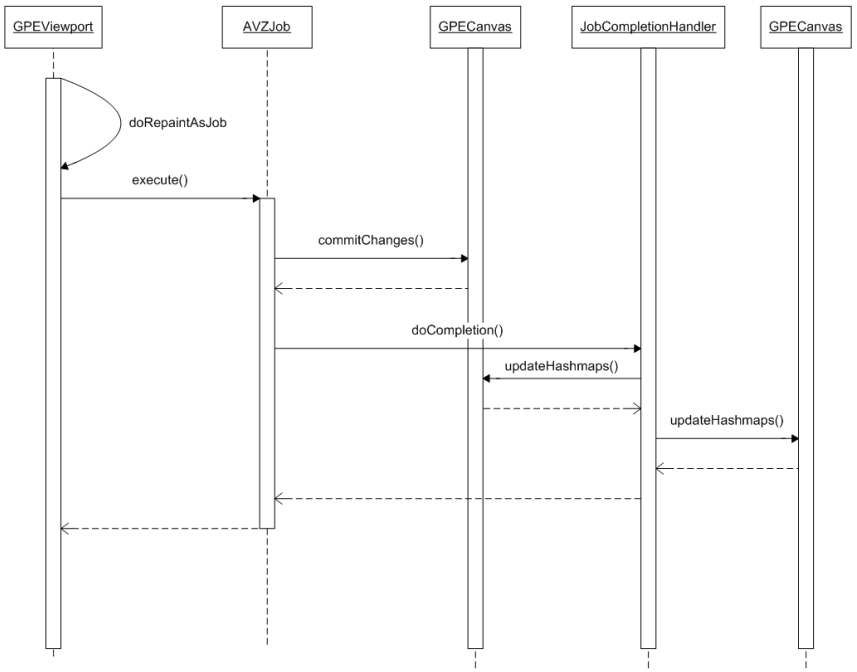


Abbildung 6.5: Sequenzdiagramm eines Repaints (vereinfacht)

- `updateHashmaps()`
  erneuert die *Hashmaps* aller Instanzen von `GPECanvas`. Somit sind alle auf `GPECanvas` basierende Module auf dem neuesten Stand.

Dieser gesamte Ablauf ist zusätzlich in einen `AVZJob` verpackt. Das stellt sicher, dass die durch `commitChanges()` durchgeführten Änderungen durch GroIMP registriert werden.
Die Wahl des *Mouse-Release*-Events als Auslöser der Aktualisierung hat den Grund der Verhinderung einer zu häufigen Ausführung der beschriebenen Ereigniskette. Das würde zu erheblichen Geschwindigkeitseinbußen führen. Insbesondere die zu häufige Ausführung eines Jobs kann GroIMP bis zu einem Absturz bringen (Vgl. Kap. 4.2.5).

6.4.3 Selektion

Von Anfang an hat GroIMP im Umgang mit Knoten vom Typ `MeshNode` eine entscheidende Lücke: Es ist nicht möglich, diese Knoten durch Anklicken im 3D-Fenster zu selektieren. Da es aber

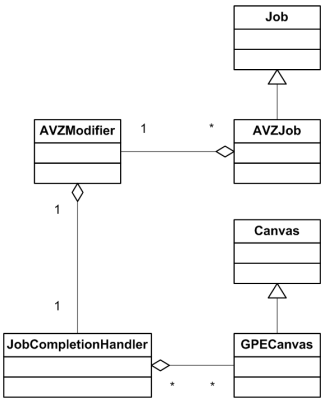


Abbildung 6.6: Klassendiagramm der Klasse AVZJob

zu einer grundlegenden Anforderung von ArchiViz gehört, auf Selektionen im 3D-Fenster reagieren zu können, war das Schließen dieser Lücke notwendig.
Da ArchiViz einen eigenen Mesh-Knoten hat, der sich äquivalent zum Typ `MeshNode` in GroIMP verhält, wurde die Implementierung zunächst in der Klasse `AVZMeshNode` vollzogen. Eine Übertragung des Prinzips auf `MeshNode` ist somit kein großer zusätzlicher Aufwand mehr, konnte allerdings nicht mehr umgesetzt werden.
Die Implementierung des Interface `Pickable` bringt folgende Methode mit sich:

```
1 public void pick( Object    node ,          //der zu untersuchende Knoten
2                   boolean   asNode ,        //ist node eine Kante oder ein Knoten
3                   Point3d   origin ,        //Ursprung in lokalen Koordinaten
4                   Vector3d  direction ,     //Sichtstrahlrichtung in lok. Koord.
5                   Matrix4d  transformation , //Transformationsmatrix lok.-> glob.
6                   PickList  list );         //Rückgabeliste mit Entfernung
7                                             //des nächsten Schnittpunktes
```

Listing 6.2: Methode `pick` des Interface `Pickable`

Zur Umsetzung der Selektion wurde ein Raytracing-Algorithmus nach (Möller und Haines, 1999, S. 305 ff.) umgesetzt.

6.4.4 Platzierung und Rotation

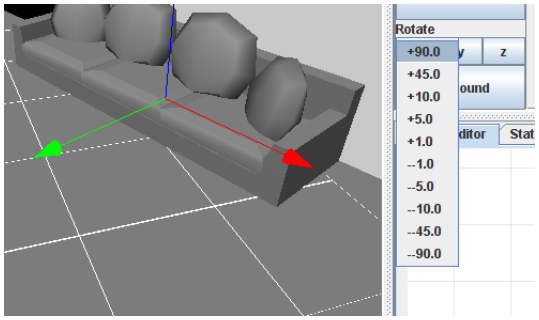


Abbildung 6.7: Das Rotationstool

Nach dem Importieren von Objekten ist es möglich, dass diese zum einen ein anderes Koordinatensystem und zum anderen einen anderen Ursprungspunkt haben. Dies ist sogar recht wahrscheinlich, da z. B. *OBJ* als Austauschformat üblicherweise die XZ-Ebene als Grundfläche hat, während GroIMP auf die XY-Ebene aufbaut. Dies hat zur Folge, dass ein importiertes Objekt zunächst um die x-Achse gedreht werden muss, sodass es konform zum GroIMP-Koordinatensystem ist. Diese Rotation wurde durch ein einfaches Werkzeug erweitert. Abbildung 6.7 zeigt das Rotationswerkzeug, womit in allen drei Achsen in bestimmten diskreten Schritten ein Objekt rotiert werden kann.
Ein weiteres Werkzeug stellt die Funktionalität bereit, ein Objekt im globalen Koordinatensystem, unabhängig von seiner Rotation, „auf den Boden zu stellen“. Das bedeutet, das Objekt wird mit jenem Vertex, der den kleinsten Wert der globalen Z-Koordinate hat, auf der XY-Ebene ausgerichtet. Abbildung 6.8 zeigt die Wirkung des implementierten Werkzeuges „To-Ground-Tool“.

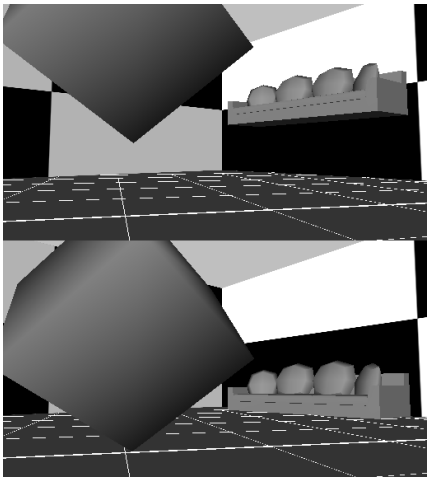


Abbildung 6.8: Das To-Ground-Tool vor und nach der Anwendung

6.5 Durchlässigkeit zwischen GroIMP und ArchiViz

Da der Szenegraph des ArchiViz-Plugins ein Teil des GroIMP-Szenegraphen ausmacht und ArchiViz recht strikt auf diesen Teilgraphen eingeschränkt ist, liegt es nahe, dass eine Übertragung von GroIMP-Knoten in den ArchiViz-Szenegraphen und zurück geschaffen werden sollte. Das erlaubt, dass z. B. durch XL modellierte Pflanzen in für ArchiViz verwendbare Geometrieknoten umgewandelt werden.
Ausgangspunkt des ArchiViz-Szenegraphen ist `AVZRoot`, dieser ist Wurzelknoten aller Knoten in ArchiViz. Durch XL erzeugte Objekte liegen üblicherweise unter einem Knoten „RGG-Root“. Diese zwei Teilgraphen sind voneinander getrennt. Alle Operationen von ArchiViz beschränken sich auf den ArchiViz-Graphen.
Eine Importierung von einzelnen RGG-Knoten oder ganzen Subgraphen konnte nur vorbereitend implementiert werden. Hierbei spielt folgender Fakt eine Rolle:
Nicht alle Geometrieknoten, welche durch XL erzeugt werden, wie z. B. `F`, stellen die erzeugten Polygone durch das Interface `Polygonizable` zur Verfügung. Eine Übertragung in ArchiViz würde bedeuten, dass dieser Knoten in `AVZMeshNode` überführt werden müsste, welcher direkt Polygone speichert und darstellt. Der Knoten `F` müsste also nachträglich das Interface `Polygonizable` implementieren. Diese Implementierung ist recht aufwändig, betrachtet man die Menge der Knoten, welche übertragen werden müssten.
Die vorbereitende Implementierung verhält sich so, dass ausgehend von einem selektierten Objekt aus XL alle unter diesem Knoten liegenden Kindknoten, welche das Interface `Polygonizable` implementieren, übertragen werden (durch Übertragung der Polygone in einen Knoten vom Typ `AVZMeshNode`). Andere Knoten werden darauf überprüft, ob sie eine Transformation enthalten (Vgl. Kap. 4.2.1). Falls ja, wird diese akkumuliert. Das bedeutet, dass mehrere hintereinander liegende Transformationsknoten zu einer Transformation zusammengefasst und auf den darüber liegenden Geometrieknoten übertragen werden. Das hat den Grund, dass nicht zwingend alle Transformationsknoten, die durch XL erzeugt bzw. durch RGG zur Verfügung gestellt werden, übertragen werden müssen.
Grundsätzlich ist es auch möglich, XL (bzw. das RGG-Plugin) par-

allel zu ArchiViz zu nutzen. Darüber hinaus werden alle Knoten in ArchiViz automatisch dem Layer 15 zugeordnet mit dem Vorteil, dass alle Knoten in ArchiViz zugleich ausgeblendet werden können. Da alle RGG-Knoten im Layer 1 erzeugt werden, können so, rein visuell im 3D-Fenster, die verschiedenen Darstellungen voneinander getrennt werden.
Zusammenfassend betrachtet ist die Implementierung nicht vollständig und wäre noch an die Gesamtapplikation anzupassen.

6.6 Persistenz

In der ursprünglichen Anforderung war X3D als Datenformat zur Speicherung der Szenen von ArchiViz und GroIMP vorgesehen. Doch es stellte sich heraus, dass die Komplexität des erzeugten Szenegraphen (siehe Abb. 6.3 auf Seite 90) eine Abbildung auf das Format X3D nur unter hohem Aufwand zuließ. Das lag darin begründet, dass die komplexen Knotenbeziehungen in X3D, welches auf die klassische Baumstruktur aufbaut, nicht ohne weiteres umsetzbar sind. Somit würde prinzipiell ein neues Format geschaffen, welches mit X3D an sich nichts mehr gemein hat, und somit auch nicht als Austauschformat hätte dienen können. Es wäre zwar während des Speicherns eine Vereinfachung des Szenegraphen auf das X3D-Format möglich gewesen (und auch das nicht ohne erheblichen Programmieraufwand), aber es sollten alle Informationen der Szene gespeichert werden, um ein lückenloses Fortsetzen der Arbeit nach dem Speichern und erneutem Laden zu ermöglichen.
Als Resultat der Überlegungen kam nur ein Format in Frage. GroIMP bietet als Datenformat ein internes, auf XML basierendes Szenenformat an, welches die gesamte Szene lückenlos serialisiert und abspeichert. Ein erneutes Laden stellt den gesamten Szenegraphen exakt so her, wie er gespeichert wurde.

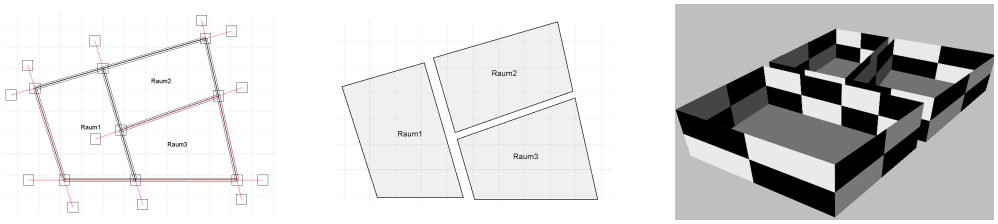
7 Grundrisseditor

Als Werkzeug zur Erzeugung der Innenraumgeometrie dient der Grundrisseditor. Hierbei kann eine vorhandene 2D-Zeichnung, als digitales Bild zur Verfügung gestellt, als Grundlage genommen werden. Grundlegend gibt es zwei Möglichkeiten der Definition der Innenraumgeometrie: Innerhalb einer zweidimensionalen Sicht auf die Szene (von oben) oder direkt in der dreidimensionalen Szene. Es steht fest, dass beide Varianten ihre Vorteile haben, doch ist es als Standard zu betrachten, dass solch eine Geometrie auf einer zweidimensionalen Ebene definiert wird und die dritte Dimension bei dieser Definition eine untergeordnete Rolle spielt. Das bedeutet, dass für die Innenraumgeometrie drei wichtige Attribute vorhanden sind: Anfangspunkt und Endpunkt einer Wand und die Wandhöhe. Es wird davon ausgegangen, dass Wände zur Erde senkrechte quaderförmige Körper sind. An dieser Stelle genügt eine Reduktion auf die Innenflächen, da im Allgemeinen die Innenwände von Interesse sind. Alles weitere kann vernachlässigt werden. Durch diese Reduktion bestehen in der dreidimensionalen Szene Wände nur aus Flächen, welche die Innenwände darstellen. Mehrere zusammenhängende Flächen stellen typischerweise einen Raum dar.

7.1 Sketching

In den ersten Überlegungen zur Funktionalität war folgende Wanderzeugung vorgesehen: Hierbei erzeugte ein Klick auf jeweils zwei Wandecken eine Wand. Allerdings unterscheidet sich dieser Workflow kaum von anderen Werkzeugen anderer Softwarepakete, ebenso ist dieser für große Pläne einfach zu langsam. Darüber hinaus ist Präzision wenig entscheidend, da korrekte Bemaßung keine Rolle spielt.
Aus diesen Überlegungen heraus galt es ein Werkzeug zu entwickeln, welches schnell und unkompliziert zur Wanderzeugung genutzt werden kann.

Das implementierte Sketching-Tool erfüllt diese Anforderungen. Mit ihm ist es möglich, sogar relativ präzise[1] mehrere Wände auf einmal zu erzeugen. Es basiert auf dem Prinzip von sich überschneidenden Hilfslinien, wobei zwei benachbarte Schnittpunkte zusammen eine sogenannte „*Sketching wall*“ bilden.
Eine *Sketching wall* stellt eine Wand eines Raumes oder von zwei Räumen dar, bildet also im letzteren Fall jeweils eine Innenseite eines anliegenden Raumes ab.



(a) Sketching (b) Logische Raumverteilung (c) Erzeugte Geometrie

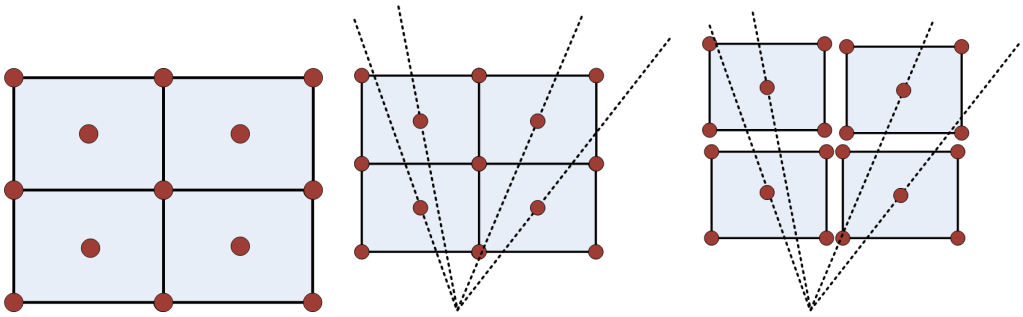
Abbildung 7.1: Vorgang des Sketchings

Abbildung 7.1 zeigt drei verschiedene Sichten auf ein *Sketching*. Die Definition der Schnittlinien und *Sketching-Walls* zeigt Abbildung 7.1a. Die Schnittlinien bestimmen implizit die Position der Innenwände. Das bedeutet, dass diese nicht die echte Position der Innenwände in der Szene darstellen, sondern deren Länge und Richtung definieren. Die Länge wird über den Abstand zweier benachbarter Schnittpunkte, die Richtung über die Schnittlinie selbst bestimmt. Würden alle Innenwände an der durch die Schnittlinie definierten Position verbleiben, würden sich Innenwände direkt benachbarter Räume überlappen. Erst wenn die Räume räumlich von einander getrennt werden, wird diese Überlappung verhindert. Die Räume selbst behalten dabei vollständig alle wichtigen Attribute: Flächeninhalt, Länge der Innenwände, Richtungsvektoren der Wände. Nur die Position der Räume selbst wird verändert. Abbildung 7.1b zeigt diese „Logische Raumverteilung“. Hierbei werden die Räume „auseinander gesprengt“ und somit überlappungsfrei dargestellt. Diese Raumverteilung ist nun die Grundlage für die dreidimensionale Szene: Abbildung 7.1c zeigt die erzeugte Geometrie.

[1] Präzision ist an dieser Stelle von mehreren Faktoren bestimmt: Zum Einen die Genauigkeit des gescannten Planes und zum Anderen die Notwendigkeit, überhaupt präzise zu sein. Oftmals hat eine Visualisierung nicht zwingend diesen Anspruch.

7.1.1 Erzeugung der logischen Raumverteilung

Das „Auseinandersprengen“ der Räume wird erreicht durch eine Verschiebung der einzelnen Räume, sodass Lücken zwischen diesen entstehen. Hierbei muss die Form und die Größe identisch bleiben. Realisiert wurde dies, indem zunächst von jedem Raum das Zentrum bestimmt wird. Dabei wird einfach der Durchschnitt über die Positionen einzelnen Eckpunkte des Raumes gebildet. Anschließend werden die bestimmten zentralen Punkte eines jeden Raumes über eine gleichförmige Skalierung von einem beliebigen Punkt des Raumes aus verschoben. Wichtig ist, dass die relative Position der einzelnen Eckpunkte eines Raumes zu seinem zentralen Punkt nicht verändert wird. Abbildung 7.2 zeigt die beschriebene Verfahrensweise.



(a) Anordnung der Räume nach dem Sketching und deren zentrale Punkte

(b) Ursprung der Skalierung

(c) Anordnung der Räume nach Verschiebung der zentralen Punkte

Abbildung 7.2: „Aufsprengung“ der Räume.

7.1.2 Bestimmung von Räumen

Nach der Definition der Innenwände über Schnittlinien fehlt noch der letzte Schritt: die Definition, welche Wand zu welchem Raum gehört. An dieser Stelle findet eine Umformulierung statt: Wände werden nur indirekt zugeordnet, und zwar über eine Auswahl von Schnittpunkten, die den Raum begrenzen und zugleich die Wandecken definieren. Optimalerweise findet diese Raumdefinition automatisch statt. Der Zeitrahmen der Beispielimplementierung ließ es allerdings nicht zu, diesen Automatismus zur Verfügung zu stellen.

Somit muss in der durchgeführten Implementierung die Selektion der Schnittpunkte zur Raumdefinition manuell durchgeführt werden. Es sei aber an dieser Stelle auf ein Konzept eines möglichen Automatismus eingegangen, welche aus Zeitgründen nicht implementiert werden konnte.

Die Vorbedingung ist, dass zwei Schnittpunkte, welche auf einer

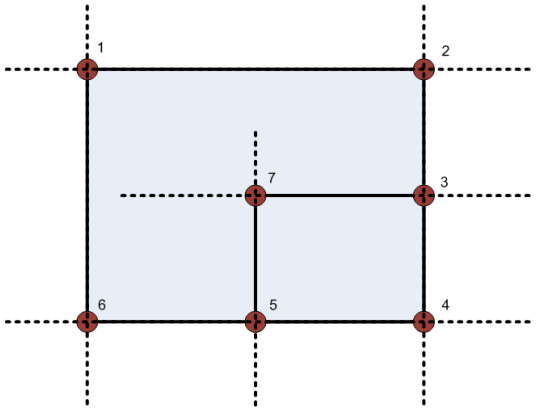


Abbildung 7.3: Zwei über Sketching definierte Räume

Schnittlinie direkte Nachbarn sind, miteinander in Beziehung stehen. Diese Beziehung kann im Szenegraphen durch eine spezielle Kante ausgedrückt werden. Die Besonderheit ist, dass die Beziehung bidirektional ist: Die Modellierung einer solchen Kante ist in GroIMP per se nicht vorgesehen, kann aber simuliert werden.

Nach Abbildung 7.3 stehen nun folgende Elemente in Beziehung: $(1,2),(2,3),(3,4),(4,5),(5,6),(6,1),(3,7),(5,7)$.

Nun müssen, ausgehend von einem beliebigen Knoten, die minimalen Zyklen über die Beziehungen der Schnittpunkte gefunden werden. Ein minimaler Zyklus bedeutet, in einer minimalen Anzahl von Schritten zum Ausgangspunkt zurück zu finden.

Am soeben genannten Beispiel wäre das, ausgehend vom Knoten 1: $1,2,3,4,5,6,1$ und $1,2,3,7,5,6,1$. Da nun zwei Zyklen gefunden wurden, muss über den Flächeninhalt der umspannten Fläche der kleinere gefunden werden. Dies ist dann eine zu einem Raum gehörende Kette von Schnittpunkten.

Der ganze Prozess muss für jeden Knoten wiederholt werden, damit alle Räume gefunden werden. Der Algorithmus zum Finden der minimalen Zyklen kann - bei einer eventuellen Implementierung - rekursiver Natur sein.

7.2 Fazit

Das in diesem Kapitel beschriebene Werkzeug stellt ein einfaches Verfahren zur Erzeugung der Wandgeometrie zur Verfügung. Es soll dem Anspruch der „einfachen Workflows“ entgegenkommen. Hierbei wird allerdings ein Kompromiss eingegangen: Präzision steht nicht an erster Stelle. Natürlich ist es möglich, relativ präzise zu arbeiten, allerdings können Schnittpunkte nur indirekt über die Schnittlinien platziert werden. Für exakte Maßeingaben ist dies nur bedingt nutzbar. Da der Anspruch dieser Arbeit zunächst darin besteht, das grundsätzliche Vorgehen zu klären, wurde diese Frage zunächst zurückgestellt.
Darüber hinaus ist es nicht möglich, die in Abbildung 7.4 dargestellte Wandkonstellation zu erzeugen: Drei Schnittlinien schneiden sich in einem gemeinsamen Punkt. Hierfür müssten erweiterte Konzepte implementiert werden, wofür der Zeitrahmen dieser Arbeit nicht ausreichte.

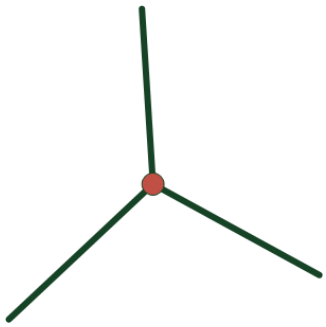


Abbildung 7.4: Drei Schnittlinien treffen aufeinander

Ebenso unterliegt der in Abschnitt 7.1.2 beschriebene Algorithmus zur Raumerkennung folgender Einschränkung: man stelle sich einen rechteckigen Raum vor, an welchen vier dreieckige Räume angrenzen. In diesem Fall kann der Algorithmus den viereckigen Raum nicht bestimmen, da alle vier Eckpunkte Teil eines anderen Raumes mit geringerer Eckenzahl sind. Dieser Spezialfall kann allerdings im vorgestellten Verfahren „*Sketching*“ nicht auftreten, da es nicht möglich ist, einen Schnittpunkt durch drei überkreuzende Linien zu erzeugen.

8 Textureditor

8.1 Motivation

Um ein realistisches visuelles Modell eines Innenraumes zu bieten, ist es als notwendig zu betrachten, dass die einzelnen Innenwände der definierten Räume mit Oberflächen zu versehen sind, welche möglichst der Realität entsprechen. In einer dreidimensionalen Darstellung bzw. Computergrafik ist solch eine Oberfläche beschrieben durch einen Shader bzw. eine Textur, wobei genau genommen eine Textur als ein spezieller Shader zu betrachten ist. Ein Shader ist ein (im softwaretechnischen Sinne) kleines Programm, welches Algorithmen zur Darstellung einer Oberfläche implementiert. Diese Algorithmen sind also Vorschriften zur Erzeugung der finalen Farbe eines Pixels auf dem Bildschirm[1].
Es gibt verschiedene Wege, eine passende und glaubwürdige[2] Textur für ein Objekt, in unserem Falle einer Innenwand zu erzeugen:

- Erzeugung „von Hand“
  In diesem Falle muss ein Paint-Programm, sei es nun vektor- oder pixelbasiert, eingesetzt werden, um durch den Einsatz verschiedener Werkzeuge das gewünschte Aussehen zu erzeugen. Dieses Verfahren ist oft schwierig, da eine realistische Abbildung einiges an Geschick und Zeit erfordert.

- Prozedurale Verfahren
  Hierbei werden Algorithmen zur Erzeugung von Mustern, z. B. Fraktale , genutzt, um das gewünschte Erscheinungsbild zu produzieren.

[1] Zu den hier verwendeten Grundlagen der Computergrafik sei auf Xiang und Plastock (2003) verwiesen.

[2] Der Begriff „glaubwürdig“ ist stark von der subjektiven Betrachtungsweise abhängig. Verschiedene Personen können, je nach Auffassung, ein Modell als glaubwürdig empfinden, andere wiederum setzen den Anspruch höher und definieren den Begriff entsprechend anders.

- Nutzung von Fotografien
  Es liegt nahe, Fotografien eines echten Objektes, in diesem Falle einer Innenwand, zu nutzen, um der Oberfläche ein realistisches Aussehen zu geben. Da die Oberfläche einer Innenwand oftmals nicht durch eine einzige Fotografie erfassbar ist, müssen mehrere Fotografien gemacht werden und in den Design-Prozess einfließen.

In der Realität wird oftmals eine Kombination aller drei Verfahren eingesetzt. Ausgangspunkt sind zumeist eine oder mehrere Fotografien, welche mit Hilfe von prozeduralen Verfahren bzw. Handarbeit so nachbearbeitet werden, dass sie dem gewünschten Erscheinungsbild entsprechen. Anschließend muss die erzeugte Textur auf das Objekt projiziert werden. Es gibt verschiedene Projektionsverfahren, wobei in Bezug auf die Projektion einer Textur auf eine (flache) Innenwand die Planarprojektion[1] zu wählen ist.
Zusätzlich besteht der Anspruch, den Zeitaufwand zur Erzeugung solch einer Textur für eine Innenwand eines Raumes möglichst gering zu halten. Erfahrungsgemäß schließt dieser Anspruch eine gänzlich manuelle Erzeugung aus, da dort der Aufwand im Gegensatz zu den anderen Methoden relativ hoch ist.
Des weiteren seien alle Innenwände ebene (also keine gewölbten

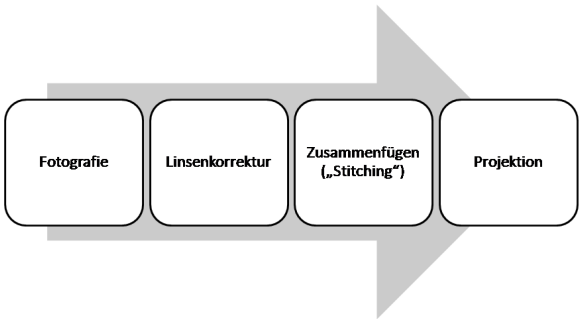


Abbildung 8.1: Arbeitsablauf von der Fotografie bis zur Projektion

oder verwinkelten) Flächen.

[1]Die Planarprojektion projiziert ein dreidimensionales Objekt auf eine ebene Fläche.

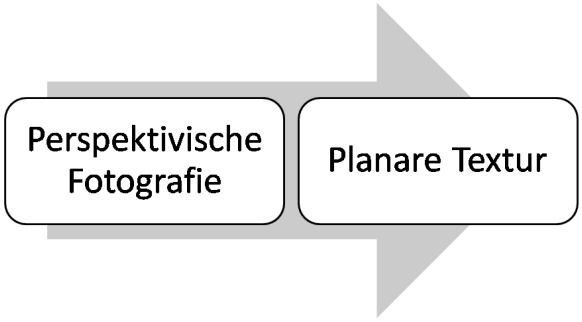


Abbildung 8.2: Informationstypen des Arbeitsablaufes

Optimal wäre eine gänzlich orthogonale Projektion[1] der Innenwand, da somit alle Verhältnisse der Abstände aller Punkte der Fläche zueinander erhalten blieben. Da jedoch das Aufnahmesystem einer Kamera näherungsweise einer perspektivischen Projektion entspricht, muss die Aufnahme in eine orthogonale Projektion überführt werden. Auf Grund des eingeschränkten Betrachtungswinkels der verfügbaren Kameras[2] und der geringen Nähe des zu fotografierenden Objektes (also der Innenwand) liegt die Anforderung nahe, dass mehrere Fotografien ein und der selben Innenwand angefertigt werden müssen. Angewendet auf die Aufgabe, Räume realistisch darzustellen, ergibt sich der in Abbildung 8.1 dargestellte Arbeitsablauf. Dieser ist pro Innenwand vollständig zu wiederholen, sofern sich die Innenwände im Aussehen nicht gleichen[3].
Ausgangspunkt bilden also eine oder mehrere Fotografien, welche in einer bestimmten Art und Weise erzeugt wurden und das ursprünglich planare Objekt perspektivisch abbilden. Die Kapitel 8.4 und 8.5 erläutern zwei Möglichkeiten, diese Fotografien zu erzeugen. Nach der Erzeugung müssen die verschiedenen Bilder oder Bildelemente (falls Details separat erzeugt wurden) zusammengefügt werden. Hierbei ist darauf zu achten, dass die Bildelemente möglichst wenig verzerrt werden. Dieser Vorgang wird im Allgemeinen *Stitching*

[1] Bei der orthogonalen Projektion liegen die Projektionslinien parallel zueinander.
[2] Es wird an dieser Stelle von handelsüblichen Digitalkameras ausgegangen.
[3] Ein Beispiel dieser Gleichheit wäre ein leerer Innenraum mit mehreren etwa gleich großen Wänden und einfacher homogener Oberfläche der selben Farbe.

genannt. Auf den Prozess des *Stitchings* als einen wichtigen Teil dieser Arbeit wird im Abschnitt 8.7 genauer eingegangen. Ergebnis dieser Schritte ist eine Textur, welche in einem bestimmten Bildformat vorliegt. Abbildung 8.2 zeigt die beiden Informationstypen, welche für den Prozess maßgeblich sind.
Anschließend folgt, wie in Abb. 8.1 bereits erwähnt, die Projektion der Textur auf die Geometrie, in diesem Falle die Innenwand. Es ist darauf zu achten, dass bei der Projektion die Wandecken und -kanten der Textur mit der Geometrie der Innenwand übereinstimmen. Kapitel 8.8 beschäftigt sich mit dieser Problemstellung.

8.2 Photogrammetrie als Grundlage

In Abb. 8.3 werden die in den vorherigen Abschnitten erläuterten Prozesse zusammengefasst dargestellt. Dies führt zu einer Abfolge von Prozessen. Um diese dargestellte Abfolge durchzuführen, sind

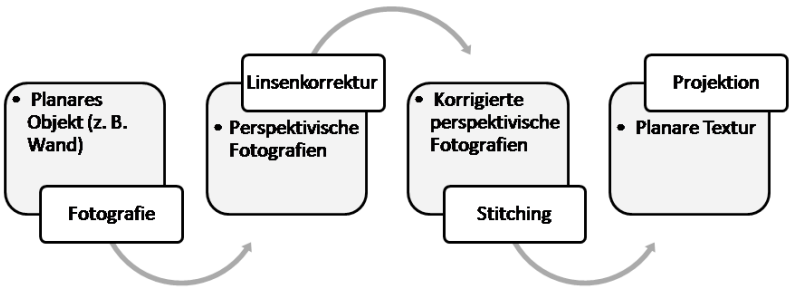


Abbildung 8.3: Gesamtablauf

Konzepte aus der Bildverarbeitung und - spezieller betrachtet - Pho-

togrammetrie notwendig. Laut ? ist die Definition für Photogrammetrie folgende:

> „*Photogrammetrie* befasst sich mit der Gewinnung und Verarbeitung von Informationen über Objekte und Vorgänge mittels Bildern, schwerpunktmäßig mit Bestimmung der Form, Größe und Lage von Objekten im Raum, vorzugsweise mittels photographischer Bilder als Informationsspeicher. Die Bilder werden durch photogrammetrische Aufnahmen gewonnen und durch photogrammetrische Auswertung verarbeitet. Werden digitale Bilder verwendet, so wird von digitaler Photogrammetrie gesprochen. - Als Luftbildphotogrammetrie mit Luftbildern, die in einigen Kilometern Höhe aufgenommen sind, - als terrestrische Photogrammetrie mit Bildern, die auf dem Erdboden aufgenommen sind. Nutzt in der Regel das Stereoprinzip, d.h. sich um bestimmte Bereiche überlappende Bilder desselben Objektes können dreidimensional ausgewertet werden."

Das gegebene Objekt, das die Informationen liefert, ist die Innenwand, die es zu verarbeiten gilt. Die zu gewinnende Information ist die Oberfläche der Innenwand als Bildinformation, damit diese auf die Geometrie der Innenwand projiziert werden kann. Hierbei spielen die in der Definition genannten Eigenschaften Form, Größe und Lage eine untergeordnete Rolle. Effektiv wird nur ein kleiner Teilbereich der Photogrammetrie benötigt, nämlich um aus den erzeugten Bildern ein Gesamtbild der Innenwand zu generieren. Dieser Teil ist die Nahbereichs-Photogrammetrie, welche sich mit Objekten von wenigen Zentimetern bis hin zu mehreren Metern Größe beschäftigt.

> „Die *Nahbereichsphotogrammetrie* ist ein Teilgebiet der Photogrammetrie,. . . [die] neben den Grundlagen aus Mathematik, Physik und Informatik . . . eng verknüpft [ist] mit Fachgebieten aus Grafik und Bildverarbeitung, z. B. mit Computergrafik, Computer Vision, Digitaler Bildverarbeitung, Computer Aided Design (CAD), Geoinformationssystemen (GIS) und Kartographie." (siehe (Luhmann, 2003, S. 6, Kap. 1.2.2, Abs. 1 und 2))
> Hierbei bezieht sich die Nahbereichsphotogrammetrie auf eine Aufnahmeentfernung von unter 300 m, wobei die Aufnahme von Objekten einer Größe von 1m bis hin zu ca. 200 m von Interesse ist.

Die Nahbereichsphotogrammetrie bedient sich verschiedener mathematischer Konzepte, um Informationen über ein Objekt zu gewinnen, das aus verschiedenen Richtungen fotografiert wurde. In erster Linie sollen solche Konzepte eine Rückgewinnung der Tiefeninformationen ermöglichen, sodass die räumliche Struktur des Objektes wiederhergestellt werden kann. Doch dieser Bereich ist für die definierte Aufgabe weniger sinnvoll, allein die Annahme, dass sich der Kamerastandort verändert, ist relevant. Wichtig für die Umsetzung des *Stitchings* (genauer: der Transformation eines Bildes) ist das Konzept der projektiven Geometrie (bzw. projektiven Abbildung). Abschnitt A.3.2 im Anhang beschäftigt sich mit dem Konzept genauer. Dies stellt in der Nahbereichsphotogrammetrie einen Spezialfall dar, da für ebene Objekte einige Vereinfachungen getroffen werden können. Die Besonderheit besteht darin, dass mehrere Aufnahmen dazu dienen sollen, ein planares Objekt aufzunehmen.

8.3 Wichtige Algorithmen

8.3.1 Target-To-Source

Ein häufiges Problem bei der Anwendung von Filtern, welche die Position von Pixeln verändern, ist, dass die Abbildungsfunktion nicht surjektiv ist. Das bedeutet, dass nicht alle Pixel des Zielbildes getroffen werden. Folge ist, dass das Zielbild durchsetzt ist von „schwarzen Löchern“, also schwarzen Pixeln.
Dieses Problem kann durch zwei Möglichkeiten umgangen werden. Die erste Möglichkeit ist die Nutzung eines Interpolationsalgorithmus, welcher die schwarzen Pixel aus den umliegenden, nicht schwarzen Pixeln berechnet. Allerdings ist zu beachten, dass das Originalbild durchaus bereits schwarze Pixel enthalten konnte. Somit muss während der Berechnung gespeichert werden, welche Pixelkoordinaten wirklich durch die Berechnungsfunktion „getroffen“ wurden. Dieser Ansatz ist durch die Einfachheit der zweiten Möglichkeit kaum gerechtfertigt.
Die zweite Möglichkeit besteht darin, über die Berechnung der inversen Funktion ausgehend von den Koordinaten der Zielpixel die dazu passenden Quellpixel zu berechnen. Listing 8.1 beschreibt das Verfahren als Pseudocode.

```
1  begin
2    Image source
3    Image target
4    f_inv = berechneInverse(f)
5    for (i=0; i<TargetZeilen; i++) do
6    begin
7      for (k=0; k<TargetSpalten;k++) do
8      begin
9        (m,n) = berechne(f_inv, (i,k))
10       target(i,k) = source(m,n)
11     end
12   end
13 end
```

Listing 8.1: Pseudocode des Verfahrens Target-To-Source

Üblicherweise ist die Funktion f die Multiplikation mit einer Abbildungsmatrix, die Invertierung der Matrix kehrt die Abbildungsrichtung (Target-To-Source) um.

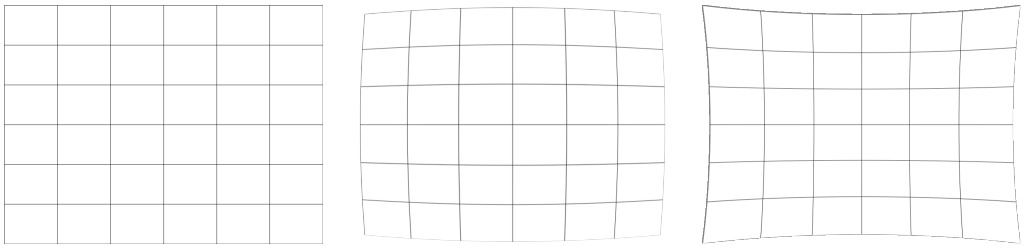
8.3.2 Linsenkorrekturfilter

Benötigt wird die (bereits inverse[1]) Funktion $lens^{-1} : canvas \times (\mathbb{R} \times \mathbb{R}) \rightarrow canvas$. Diese inverse Transformationsfunktion $p = lens^{-1}(p', (a_x, a_y))$ lautet nach Bourke (2002):

$$
\begin{aligned}
p_x &= \frac{p'_x}{1 - a_x || \frac{p'}{1 - a_x ||p'||^2} ||^2} \\
p_y &= \frac{p'_y}{1 - a_y || \frac{p'}{1 - a_y ||p'||^2} ||^2}
\end{aligned}
\tag{8.1}
$$

Die Parameter a_x und a_y liegen typischerweise zwischen 0 und 0.1. Abb. 8.4 zeigt die Wirkung der beschriebenen Abbildung. Hierbei ist das Foto bei einer üblichen Kamera ähnlich Abb. 8.4c verzerrt. Das bedeutet, dass diese Verzerrung durch die Linsenkorrektur ausgeglichen werden muss. Ziel ist es, dass das Foto dem Ideal (Abb. 8.4a) nahe kommt. Gleichung 8.1 geht von einer idealen Linse aus und vollzieht die in Abb. 8.4b dargestellte Transformation. Da es aber viele unterschiedliche Verzerrungsarten[2] gibt, hat dieser Ausgleich einen approximativen Charakter.

[1] Für eine Transformation Target-To-Source ist die inverse Transformation maßgeblich.
[2] Vgl. Kap 8.6, S. 116



(a) Ausgangsbild (b) Transformation nach Außen (c) Transformation nach Innen

Abbildung 8.4: Wirkung der Linsenkorrekturfunktion

8.4 Fotografie mit festem Kamerastandort

Dieser Ansatz ist in erster Linie in der Panoramafotografie zu finden. Ein Panorama ist ein Bild oder eine Fotografie mit einem sehr großen (horizontalen) Betrachtungswinkel. Dieser Winkel kann bis zu 360 Grad betragen, diese Panoramen werden entsprechend 360°-Panoramen genannt. Der Anwendungsbereich geht von interaktiven Präsentationen, bei denen sich der Betrachter um 360 Grad drehen, aber üblicherweise nicht den Standort wechseln kann, bis hin zu Landschaftsfotografien, welche einen besonderen Überblick bieten sollen. Ebenso ist es möglich, Aufnahmen mit besonders hoher Auflösung zu erzeugen. Durch Vergrößerung der Brennweite (also durch „Hineinzoomen“) kann der Bildausschnitt pro Foto verkleinert werden. Somit müssen zwar mehr Fotos erzeugt werden, aber die Auflösung steigt entsprechend proportional. Ziel ist es nun, mehrere sich überlappende Fotos einer Innenwand zu machen. Die Bewegung der Kamera wird hierbei auf horizontales Schwenken reduziert, der Standort bleibt konstant. Das bedeutet, dass die Transformation der Kamera von Bild zu Bild eine eindimensionale Bewegung ist, nämlich die Rotation um eine einzige Achse. Das vereinfacht das spätere Zusammenfügen (siehe Kap. 8.7), da die Parameter der Transformation nur aus einem einzigen Wert, nämlich der Rotation selbst bestehen.
Besonders vorteilhaft ist dies, wenn die Drehung automatisiert, d.h. computergesteuert stattfinden kann TBK (1997–2007). Dabei ist der Parameter (die Rotation) fest programmiert und ein spezielles Stück Hardware steuert die Drehung und das Auslösen der Kamera. Das bewirkt eine vollständige Kenntnis aller Kameraparameter, die für

ein Zusammenfügen notwendig sind. Als Resultat entsteht eine Kette von Bildern, welche nur noch geringfügigen Änderungen (Transformationen) unterzogen werden müssen.
Das Verfahren wird komplexer - aber auch wertvoller -, wenn statt nur einer Drehung um die horizontale Achse die vertikale hinzu genommen wird. Das bedeutet, dass beliebig viele Fotografien in beliebige Richtungen überlappend aufgenommen werden können, wobei nach wie vor der Kamerastandort konstant bleibt. Das ermöglicht eine Vergrößerung des Kamerablickwinkels auch in vertikaler Richtung.
Diese Aufnahmetechnik wird in erster Linie zur Erstellung von Landschaftsfotos oder Horizontbildern von Städten verwendet - es sind also Objekte, die in größerer Entfernung liegen, interessant.
Einer der größten Vorteile dieser Technik ist die Fülle an vorhandenen Lösungen, sei es in Form kommerzieller Softwarepakete oder freier Lösungen. Ebenso stehen mehrere Bibliotheken quelloffen zur Verfügung. Nachfolgend ist eine Auswahl an Lösungen[1] aufgelistet:

- Panorama Tools
  Photoshop-Plugin, quelloffen, C++, zu finden unter Dersch und Dersch (2001)

- hugin - Panorama photo stitcher
  GUI-Frontend zu den Panorama Tools, zu finden unter hugin

- Autostitch, eine Demoversion ist zu finden unter Brown (2008)

Diese Form des *Stitchings* hat für den geforderten Anwendungsbereich - nämlich die Texturerzeugung von Innenwänden - mehrere entscheidende Nachteile, welche dazu führen, dass das Verfahren, wie es vorliegt, nur eingeschränkt verwendbar wäre. Das Grundprinzip des Konzeptes, nämlich der konstante Kamerastandpunkt, begründet verschiedene Nachteile in Bezug auf die Anforderung. Abbildung 8.5 zeigt zwei dieser Probleme:

- Verdeckung sichtbarer Bereiche durch hervorstehende Objekte
  Hierbei werden durch hervorstehende Objekte Bereiche der dahinter liegenden Wand verdeckt, obwohl diese eigentlich bei einer Verlagerung des Standpunktes sichtbar wären. Dieses Problem verstärkt sich noch dadurch, dass man üblicherweise re-

[1] Auf Grund mangelnder Zeit konnten nicht alle Lösungen auf die vorhandene Funktionalität getestet werden.

lativ dicht vor der Wand steht, da der Platz durch die Raumgröße beschränkt ist. Je dichter diese Objekte vor der Kamera sind, desto mehr wird verdeckt.

- Verringerung der Auflösung zu den Wandecken hin
  Dies ist ein nicht zu unterschätzendes Problem. Je weiter die Wandecken im Verhältnis zur Wandmitte von der Kamera entfernt sind, desto geringer wird dort auch die Auflösung. So kann es schnell passieren, dass zu den Ecken hin nur noch die Hälfte der Pixel zur Verfügung stehen. Das ist besonders problematisch, wenn Nachbearbeitungen notwendig sind. Zwar haben moderne Digitalkameras mehrere Millionen Pixel Auflösung (die verwendete Digitalkamera liefert ein Bild mit einer Auflösung von 3872x2592), aber dennoch gehen bei einer zu starken Verringerung der Auflösung möglicherweise wichtige Details verloren.
  Diese Einschränkung kann aber vernachlässigt werden in Bezug auf die spätere Echtzeitpräsentation. Hierbei sind nur bestimmte Maximalgrößen der Texturen zugelassen, sodass eine Darstellung eines solch hohen Detailgrades von vornherein nicht möglich ist. Einzig und allein die Genauigkeit kann unter der verringerten Auflösung zu den Wandecken hin leiden.
  Ist aber die erzeugte Textur schlussendlich für andere Zwecke gedacht, z. B. zur Archivierung oder Klassifikation (wenn man z. B. von historischen Inhalten ausgeht), sollte der Detailgrad eine wichtige Rolle spielen.

8.5 Fotografie mit variablem Kamerastandort

Die Grundlagen zur Fotografie mit variablem Kamerastandort und deren Einsatz sind in erster Linie durch die Geodäsie (Vermessung der Erdoberfläche) erforscht worden. Hierbei sind beispielsweise hochauflösende Kameras an der Unterseite eines Flugzeuges montiert. Zur gleichmäßigen Vermessung der Oberfläche überfliegt das Flugzeug das zu vermessende Gebiet in gleich bleibender Höhe und löst die Kamera in äquidistanten Zeitabschnitten - abhängig von der Geschwindigkeit - aus. Dasselbe Prinzip funktioniert auch von einem Satellit aus, wobei dort die Kameras eine erheblich größere Auflösung besitzen müssen.

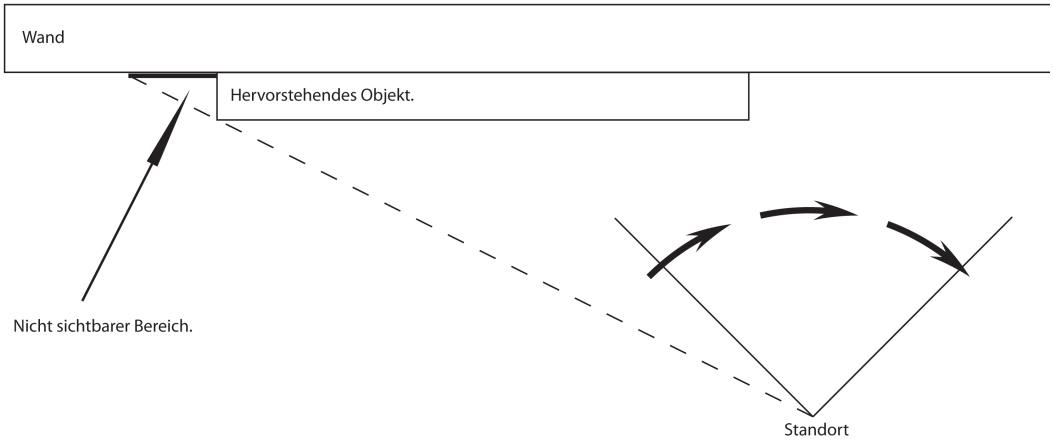


Abbildung 8.5: Verdeckung von Wandteilen durch hervorstehende Objekte

Ergebnis ist eine Kette von Fotografien, welche in die vorhandenen Karten einsortiert („registriert“) werden müssen. Hierbei sind auf den Karten bekannte und auf den Fotografien festgehaltene Merkmale zu überdecken (sog. „Passpunkte“). Bei der Erzeugung der Fotografien wird darauf geachtet, dass die Kamera sich möglichst lotrecht zur Erde befindet, da so die Verzerrungen minimiert werden. Je weiter die Kamera von der Oberfläche entfernt ist, desto geringer ist die Verzerrung.
Das beschriebene Grundprinzip lässt sich auf das Fotografieren der Innenwände anwenden: Vergleichbar mit einem Überflug, bei dem die Kamera nahezu senkrecht auf die Erde gerichtet ist, fotografiert der Anwender die darzustellende Wand. Die Randbedingung, möglichst senkrecht die Wand zu fotografieren, bleibt erhalten. Nach jedem Foto wechselt der Anwender den Standort und nimmt somit die gesamte Wand ähnlich einem Überflug, natürlich in viel geringerer Distanz, auf. Genau wie in der Panoramafotografie müssen sich die Fotos überlappen, damit gemeinsame Merkmale für die Registrierung gefunden werden können.

Diese Vorgehensweise löst einige Probleme der Fotografie mit festem Standort, erzeugt aber auch neue:

- Verdeckung sichtbarer Bereiche durch hervorstehende Objekte
  Durch die Aufnahme der Wand durch einen annähernd senkrechten Blickwinkel werden hervorstehende Objekte aus einer nahezu gleichen Perspektive wie die Wand selbst aufgenom-

men. Hierbei bestehen natürlich weiterhin die Probleme einer perspektivischen Verzerrung. Darüber hinaus werden, bei Überschreitung einer gewissen Größe der Objekte (genauer gesagt: Tiefe), Teile der Wand zunehmend verdeckt. Um dem entgegen zu wirken, müssen die Fotografien so gewählt werden, dass eben solche Verdeckungen möglichst gering werden.

- Verringerung der Auflösung zu den Wandecken hin
  Durch die gleichmäßige Aufnahme der Wand von einem Ende zum anderen hin und die Verwendung von in erster Linie zentralen Bildelementen der Fotografien sind die Abweichungen der Abstände der Wandelemente von der Kamera bei weitem geringer.

Hinzu kommt ein weiterer Vorteil:

- Es sind zusätzliche Detailaufnahmen durch Veränderung des Standortes möglich.
  Ist ein wichtiger Teil der zu fotografierenden Wand durch ein nicht direkt an der Wand stehendes Objekt verdeckt, kann durch Verlagerung des Kamerastandpunktes bzw. -winkels auch dieser Bereich erfasst werden.

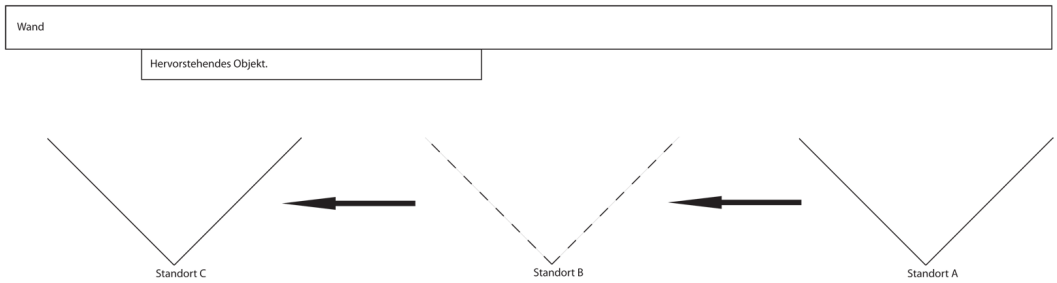


Abbildung 8.6: Durchführung der Fotografien ähnlich einem Satellitenflug

8.6 Linsenkorrektur

Jede Linse einer analogen oder digitalen Kamera erzeugt herstellungsbedingte Fehler, also Abweichungen vom optischen Ideal. Dazu gehören z. B.

- *chromatische Aberration*
  bezeichnet Farbfehler (z. B. Verschiebungen in den Rotbereich) entlang von Kanten verschiedener Kontraststufen.

- *Verzeichnung*
  ist eine Krümmung des Bildes nach innen oder außen.

- *Lens Vignetting*
  steht für eine Abdunklung der Bildbereiche zu den Ecken hin[1].

Entscheidend für ein genaueres Stiching ist die Minimierung der Verzeichnung - da diese auch der Fehler mit der stärksten Ausprägung bei heutigen Kameratypen ist.
Es gibt eine Reihe von verschiedenen Verzeichnungen, die durch die Beschaffenheit der Aufnahmelinse bestimmt werden. Es gibt „Radial-symmetrische Verzeichnung“, „Radial-asymmetrische und tangentiale Verzeichnung“ und „Affinität und Scherung“[2]. Typischerweise erzeugen Weitwinkelobjektive (im Extremfall Fischaugen-Objektive mit extrem großen Einfallswinkeln) eine Verzerrung nach außen, während Linsen mit kleinerem Bildwinkel eher Verzerrungen nach innen erzeugen.
Es gilt also eben diese Verzerrungen vor dem eigentlichen Stitching auszugleichen, um den Fehler bei der Zusammenführung der Fotografien zu verringern. Dies ist insbesondere deshalb wichtig, da die Überlappungen, die zur Registrierung notwendig sind, eben in den Bildbereichen mit der stärksten Verzerrung, nämlich am Rand des Bildes, liegen.
Die Umsetzung eines solchen Linsenkorrekur-Algorithmus wurde im Wesentlichen nach Bourke (2002) durchgeführt. Der Abschnitt 8.3.2 erläutert die Implementierung genauer.

8.7 Stitching (Registrierung)

Nach Erzeugung der notwendigen Daten, nämlich überlappenden Fotografien der Innenwand, sind diese zu einem großen Bild zusammenzufügen. Dieses Verfahren trägt viele Namen. Im Bereich der Geodäsie ist der Begriff „Registrierung“, in diesem Fall meist von Satellitenaufnamen, geläufig. Die Panoramafotografie hat den Begriff

[1] Vgl. (Luhmann, 2003, S. 118, Abschn. 3.2.2.6).
[2] Nach (Luhmann, 2003, S. 120 ff.).

„Stitching“ geprägt, während andere Quellen „Mosaik“ verwenden.
I.A. ist allen Begriffen ein zweistufiges Verfahren gemein, Möller (2005) beschreibt (in Abb. 1.3) die Erzeugung eines Mosaikbildes in ebendiesen Stufen:

- Registrierung
- Integration

Die Phase der „Registrierung“ beschreibt zunächst die Schätzung oder Eingabe der Transformationsparameter, um die einzelnen Bilder in ein gemeinsames Koordinatensystem zu überführen, und die Durchführung der Transformation. In der zweiten Phase, der „Integration“, werden die einzelnen Bilder so angepasst, dass sie sich in das Gesamtbild integrieren. Das kann z. B. durch Elimination sichtbarer Farbkanten an den Bildgrenzen oder Anpassung der Helligkeitsunterschiede geschehen.
Es gibt eine Reihe automatisierter Verfahren, welche das Stitching fast ohne Zutun des Anwenders durchführen können. Eines davon ist die Verwendung invarianter Merkmale[1], beschrieben in Brown und Lowe (2007). Hierbei werden Merkmale erkannt, die verschiedene Bilder jeweils paarweise gemeinsam haben. Allerdings wird hier angenommen, dass der Standort der Kamera konstant bleibt, das Verfahren wurde also für die Panoramafotografie entwickelt. Das ist auch der Grund, warum dieses Verfahren für die geplante Implementierung ungeeignet war. Die automatische Erkennung der Merkmale würde durch die verwendete Aufnahmetechnik fehlerhafte Werte produzieren, denn es würden Bildteile, die nicht zur Wand gehören, einbezogen. Bei der Panoramafotografie wiederum ist es durchaus möglich, alle Bildelemente einzubeziehen, da das perspektivische Verhalten eines Objektes von Bild zu Bild erhalten bleibt (sofern das Objekt keine Eigenbewegung besitzt).
Somit wurde vorerst auf eine manuelle Eingabe der Merkmalsparameter gesetzt. Darauf aufbauend wäre es möglich, eine Merkmalserkennung in einem gewissen Bereich um eine Benutzereingabe durchzuführen. Das würde bedeuten, der Anwender hat nur bestimmte Merkmale „ungefähr“ zu markieren. Anschließend würden entsprechende gemeinsame Merkmale von Bild zu Bild gefunden und einbezogen werden. Doch eine automatische Merkmalserkennung wie die beschriebene wäre mit einem hohen Implementierungs-

[1]Der hier verwendete Begriff „Merkmal“ wird im folgenden Kapitel 8.7.1 erläutert.

aufwand verbunden gewesen, welcher aus Zeitgründen nicht eingegangen werden konnte.
Die zweite Phase ist die Phase der Integration. Hierbei werden die geometrisch angepassten bzw. platzierten Bilder insbesondere an den Übergängen in ihren Helligkeits- und Farbwerten korrigiert („Gain Compensation“, siehe (Brown und Lowe, 2007, Kap. 6)). Die dafür verwendeten Fehlerfunktionen basieren auf den Helligkeitsdifferenzen der sich überlappenden Pixel. Da bei der Registrierung ein bestimmter Fehler vorhanden ist (nicht jeder Pixel in der überlappenden Region eines Bildes liegt genau über dem äquivalenten Pixel der entsprechenden überlappenden Region des anderen Bildes) wird zur Berechnung der Differenzen eine Summation mehrerer Pixel eingesetzt. Das macht die Fehlerberechnung resistenter gegen Registrationsfehler.

8.7.1 Formale Grundlagen des implementierten Verfahrens

Bevor das eigentliche Stitching durchgeführt werden kann, müssen einige Vorbedingungen erfüllt werden. Hierfür folgen einige allgemeingültige Definitionen:

$$C = \{c \mid c = (r, g, b) \text{ mit } r, g, b \in \{0; 1; \ldots; 255\}\} \tag{8.2}$$

Somit ist $c \in C$ ein RGB-Farbwert mit 24 Bit (8 Bit pro Kanal). Zusätzlich sind digitale Bilder oftmals mit einem *Alpha-Kanal*[1] ausgestattet, welcher hier zur Vereinfachung ignoriert wird. Die Menge C enthält alle möglichen Farbwerte ($256^3 = 16777216$ verschiedene Farbwerte). Ebenso ist es möglich, statt 8 Bit pro Kanal mehr oder weniger einzusetzen. Dies ist von Bildformat zu Bildformat unterschiedlich und auch abhängig vom Anwendungsbereich.
Nun sei B die Menge der zu bearbeitenden Bilder:

$$B = \{b_0, b_1, \ldots, b_n\}, n \in \mathbb{N}, \tag{8.3}$$

während ein Bild definiert ist als[2]:

$$b_i = \{p_{\mathbb{N}}^{k=0,1,\ldots,n_p}\},\ n_p \in \mathbb{N} \tag{8.4}$$

[1] Im Alpha-Kanal eines Bildes ist jedem Pixel ein zusätzlicher Graustufenwert zugeordnet. Dieser kann beispielsweise als Transparenzwert interpretiert werden.
[2] Nach (Nischwitz u. a., 2007, S. 570)

und ein Pixel $p^k_{\mathbb{N}}$ eines Bildes $b_i \in B$ definiert ist als:

$$p^k_{\mathbb{N}} = (x, y, c) \in \mathbb{N} \times \mathbb{N} \times C \tag{8.5}$$

Abbildung 8.7 zeigt eine solche Menge von Fotografien.



(a) Fotografie links (b) Fotografie Mitte (c) Fotografie rechts

Abbildung 8.7: Ein Beispiel einer möglichen Menge von Fotografien.

Zuerst müssen alle Eingabebilder in ein übergeordnetes Koordinatensystem übertragen werden, damit die einzelnen Pixel der verschiedenen Bilder im selben Koordinatensystem referenzierbar sind. Dieses (kartesische) Koordinatensystem wird im folgenden *Canvas*[1] genannt, in Anlehnung an die Implementierung, und ist definiert als:

$$canvas = \{p_c = (x_c, y_c) \mid x_c, y_c \in \mathbb{R}\} \tag{8.6}$$

Da insbesondere auch Bilder verschiedener Auflösung denkbar sind, muss eine Beziehung zwischen den Pixeln verschiedener Bilder herstellbar sein. Die eigentliche Übertragung in den *Canvas* besteht nun in der Umrechnung der lokalen Koordinaten der Pixel in Koordinaten des *Canvas*. Diese Koordinaten können anschließend noch in die normalisierte Form überführt werden[2] (d.h. der Wertebereich ist zwischen -1 und 1). Jedoch ist dies für den Algorithmus in der vorliegenden Form nicht notwendig. Es ist zu beachten, dass ein Pixel nach Übertragung in den *Canvas* keinen Punkt, sondern eine Fläche darstellt. Das bedeutet, eine Umrechnungsfunktion muss eine gegebene Koordinate des *Canvas* in ein konkretes Pixel des Bildes umrechnen und umgekehrt. Ebenso soll diese Umrechnung immer im lokalen Koordinatensystem des Bildes stattfinden. Als Resultat sind die Achsen des Bildkoordinatensystems parallel zu den Achsen

[1] vgl. 5.2
[2] vgl. (Szeliski, 2006, S. 3)

des *Canvas* ausgerichtet[1]. Zunächst sei ein Pixel im lokalen Koordinatensystem eines Bildes auf dem *Canvas* analog zu Definition 8.5 definiert:

$$p_{canvas} = [x; x+1) \times (-(y-1); -y] \times c \text{ mit } x \in \mathbb{R}^+, y \in \mathbb{R}^-, c \in C \quad (8.7)$$

Das Minus vor dem y im zweiten Intervall ist notwendig, weil der Ursprung des Bildes sich in der linken oberen Ecke befindet. Zu beachten ist, dass p_{canvas} sich auf ein Pixel im lokalen Koordinatensystem des Bildes bezieht. Das eigentliche Koordinatensystem des *Canvas* deckt natürlich die gesamten reellen Zahlen ab. Ein Pixel des Bildes mit den Koordinaten $(205, 172)$ entspricht also $[205.0; 206.0), (-173.0, -172.0]$ auf dem *Canvas* in lokalen Koordinaten des Bildes. Das Koordinatensystem des Canvas ist in Abbildung 8.8 genauer gezeigt.

Nun ist noch zu beachten, dass der Anwender die Bilder beliebig auf

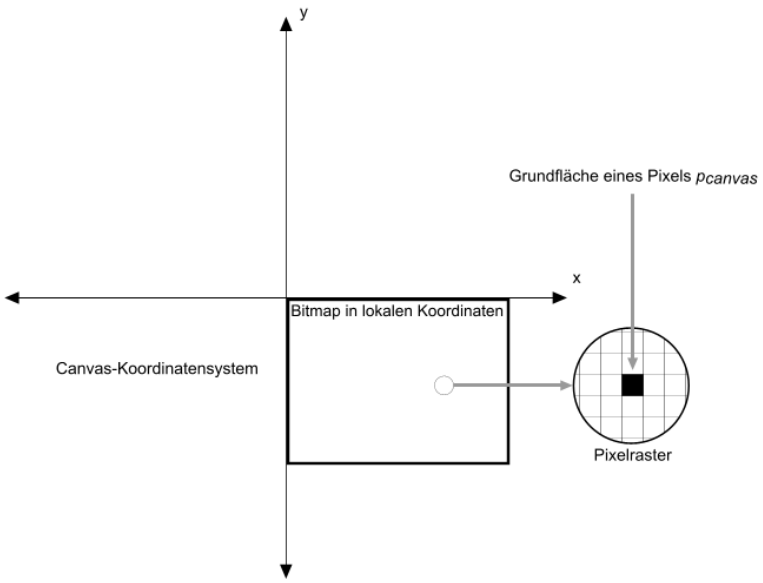


Abbildung 8.8: Das Koordinatensystem des Canvas

dem *Canvas* platzieren kann. Das bedeutet, dass jedem Bild $b \in B$ genau eine affine 3×3-Transformationsmatrix A_{aff} in homogenen Koordinaten zugeordnet ist. Dies muss beim späteren Stitching be-

[1] vgl. Abb. 8.8

achtet werden. Der Abschnitt A.2 des Anhangs erläutert die Begriffe affine Transformationsmatrix und homogene Koordinaten genauer. Somit sei:

$$M \subseteq \{(b, l) \mid b \in B, l \in L_{aff}\} \tag{8.8}$$

wobei

$$L_{aff} = \{A_{aff} | A_{aff} = (a_{ij})_{i=0,\dots,2, j=0,\dots,2} \wedge a_{ij} \in \mathbb{R} \wedge (a_{2k})_{k=0,1} = 0\} \tag{8.9}$$

L_{aff} ist die Menge aller affinen Transformations-Matrizen, während die Menge M alle Zuordnungen einer affinen Transformationsmatrix zu einem Bild enthält mit der Forderung:

$$\forall b \in B \; \exists l \in L_{aff} : (b, l) \in M \wedge l \; eindeutig \tag{8.10}$$

Das bedeutet, dass jedem Bild genau eine Matrix zugeordnet ist. Diese affine Transformationsmatrix l entspricht der globalen Transformationsmatrix, welche das Bild auf dem *Canvas* platziert.
Des Weiteren wird die entscheidbare Hilfsfunktion t zur Bestimmung, ob ein gegebener Punkt auf dem *Canvas* innerhalb eines Bildes liegt, benötigt:

$$t : M \times (\mathbb{R} \times \mathbb{R}) \to \{0; 1\} \tag{8.11}$$

und ist definiert als:

$$t((b, l), p) = \begin{cases} 1, & \text{falls } (l^{-1} \cdot p) \in b \\ 0, & \text{sonst} \end{cases}, \text{ mit } (b, l) \in M, \; p \in \textit{Canvas} \tag{8.12}$$

> Anmerkung:
> Die Aussage $p \in b$ der Definition 8.12 ist folgendermaßen zu verstehen: Der Punkt p auf dem *Canvas* wird zunächst in die lokalen Koordinaten des Bildes b umgerechnet, äquivalent zur Gleichung 8.13 (Berechnung von p_l). Befindet sich das Resultat innerhalb der Ausmaße des Bildes b auf dem *Canvas*, ist $p \in b$ wahr, sonst falsch.

Nun sei die Funktion $f : canvas \times M \to b_i$ zur Umrechnung einer homogenen Koordinate des *Canvas* $p = \begin{pmatrix} x/w \\ y/w \\ w/w \end{pmatrix}$ in ein Pixel $p' = \begin{pmatrix} x' \\ y' \\ c \end{pmatrix}$

eines Bildes definiert als:

$$f(p,(b,l)) = integer(l^{-1} \cdot p), (b,l) \in M, p \in Canvas \quad (8.13)$$

mit Farbkoordinate $c = \gamma(p_l)$

wobei l die dem Bild zugeordnete affine Transformationsmatrix ist. Zunächst wird der Punkt p über die inverse Matrix l^{-1} in das lokale Koordinatensystem des Bildes überführt (dies ist notwendig wegen der beliebigen Platzierung des Bildes auf dem *Canvas*). Anschließend wird über die Funktion $integer$ der Nachkommaanteil der beiden Komponenten des Resultates gestrichen. Somit stehen zwei natürliche Zahlen als Indizes zur Verfügung. Falls x' oder y' die jeweiligen Ausmaße des Bildes überschreiten oder kleiner 0 sind, liegt der Punkt p außerhalb des Bildes. Dies muss von Fall zu Fall geprüft werden.
Nun ist noch der Farbwert c über die Funktion γ zu bestimmen. Hierbei gibt es mehrere Möglichkeiten:

- Direktes Auslesen
  Falls sich p' innerhalb des Bildes befindet (d.h. die Funktion $t((b,l),p)$ aus Def. 8.12 ergibt 1), kann der Farbwert direkt vom korrespondierenden Pixel mit den Indizes x' und y' abgenommen werden.

- Interpolation
  In diesem Fall würde der Punkt p_l auf dem *Canvas* dazu genutzt werden, um die Entfernung vom Mittelpunkt des korrespondierenden Pixels zu bestimmen. Anschließend kann der Farbwert der Nachbarpixel - entsprechend gewichtet durch die Entfernung vom Mittelpunkt - in die Farbwertberechnung einbezogen werden. Anschließend würde die Funktion $integer$ auf den Punkt $l^{-1} \cdot p$ angewandt werden.

Anmerkung:

Es gibt folgende unterschiedliche Arten von Punkten:

- Pixel auf einem Bild $p^k_{\mathbb{N}} = (x, y, c)$
- Pixel als Fläche (zwei Intervalle + Farbwert) im lokalen Koordinatensystem des Bildes auf dem *Canvas* $p_{canvas} = (x_{canvas}, y_{canvas}, c)$
- Pixel als diskretes Wertepaar im lokalen Koordinatensystem des Bildes auf dem *Canvas*
- Ein einzelner Punkt auf dem *Canvas* $p_i = (x_i, y_i)$
- Ein einzelner Punkt auf dem *Canvas* in homogenen Koordinaten $p_{hom} = (x_{hom}, y_{hom}, w_{hom})$, wobei wir $w_{hom} = 1$ annehmen.

Abschließend ist noch zu klären, wie der Input insgesamt aussehen muss, damit das Stitching durchgeführt werden kann. Dabei können Teile des Inputs durch den Anwender vorgegeben werden, andere können durch Vorberechnungen zur Verfügung gestellt werden.

Eine *Überlappung* ist eine inhaltliche Überdeckung zwischen zwei Eingabebildern. Entscheidend ist nicht, welche Pixelwerte des einen Bildes auf dem anderen Bild ebenso zu finden sind, sondern welche Pixel zu welchem Objekt auf dem Bild und ob das Objekt zu dem hier erforderlichen planaren Objekt gehört. Es können natürlich mehrere Objekte zu dem planaren Objekt gehören, beispielsweise Fensterrahmen, Fenster und Wand. Die minimalen Höhenunterschiede (genauer gesagt: Tiefenunterschiede, betrachtet vom Kamerastandpunkt aus), z. B. die der Fensterbank, spielen in diesem Fall kaum eine Rolle. Zwei Bilder überlappen sich, wenn ein Teil ebendieses Objektes sich auf dem einen Bild befindet, und genau der selbe Teil auf dem anderen Bild.
Speziell in diesem Fall ist das planare Objekt von Interesse. Der Anwender muss, wie bereits im vorherigen Kapitel erläutert, identische, zu dem planaren Objekt gehörende Elemente zweier Fotografien selektieren. Die Selektion selbst stellt dann natürlich nur ein Pixel dar (bzw. ein Pixel auf jedem Bild jeder Überlappung).
Die *Überlappungsrelation* verbindet dem entsprechend zwei Bilder, die sich überlappen. Es ist jedoch zu bedenken, dass - obwohl zwei Bilder sich überlappen - diese beiden nicht notwendig Teil der letztlich verwendeten Unterrelation der Überlappungsrelation sind, denn es obliegt dem Anwender (oder einem automatisierten Algorithmus), welche Bilder einander zugeordnet werden.

S sei eine Menge von selektierten Punkten auf dem *Canvas*:

$$S = \{s_0, s_1, \ldots, s_n\}; n \in \mathbb{N} \tag{8.14}$$

wobei ein selektierter Punkt $s \in S$ äquivalent zu Definition 8.6 definiert sei. Die Menge $B_P \subseteq B \times B$ sei folgende Relation:

$$B_P = \{(b_u, b_v) \mid b_u, b_v \in B \ \wedge \ b_u \ ueberlappt \ b_v \wedge \ b_u \neq b_v\} \tag{8.15}$$

Die Relation B_P ist irreflexiv, symmetrisch und nicht notwendig transitiv[1]. Diese Relation kann als *Überlappungsrelation* bezeichnet werden. Sie ist speziell im gegebenen Fall durch den Anwender zu definieren.

An dieser Stelle seien die Eigenschaften Irreflexivität, Symmetrie und

[1] Definitionen befinden sich im Anhang, Kap. A.1

Nicht-Transitivität in Bezug auf die Menge B_P und deren jeweilige Notwendigkeit erläutert. Die Sicherstellung, dass diese Eigenschaften erfüllt sind, ist beispielsweise bei der späteren Implementierung möglich durch Prüfung des Inputs des Anwenders.

- Irreflexivität:
  Ein Bild kann nicht sich selbst überlappen. Wäre die Inputmenge B_P reflexiv (oder hätte auch nur ein reflexives Element), würde die spätere projektive Transformation das Bild selbst wieder ausgeben. Der Grund ist: Zwei gleiche Bilder haben die gleichen Merkmale. Ein Bild überführt in ein Bild mit den selben Merkmalen (der Anwender muss ja auf beiden Bilden die selben Merkmale markieren) würde dasselbe Bild ergeben. Dies ist ein nicht erwünschtes Verhalten und auch nicht sinnvoll. Ein Algorithmus, der auf der Menge B_P arbeitet, würde sinnlose Berechnungen durchführen.

- Symmetrie:
  Überlappt sich ein Bild mit einem anderen, überlappt sich dieses auch mit ersterem.

- Nicht zwingend transitiv:
  Überlappt sich b_1 mit b_2 und b_2 mit b_3, darf sich b_1 nicht mit b_3 überlappen ($b_1, b_2, b_3 \in B$). Dies ist allerdings nicht immer der Fall, da es durchaus sein kann, dass Transitivität bei der Überlappungsrelation vorliegt. Abbildung 8.9 zeigt ein Gegenbeispiel. Es ist also notwendig, aus B_P eine Unterrelation B_L auszuwählen und dem Input beizufügen.

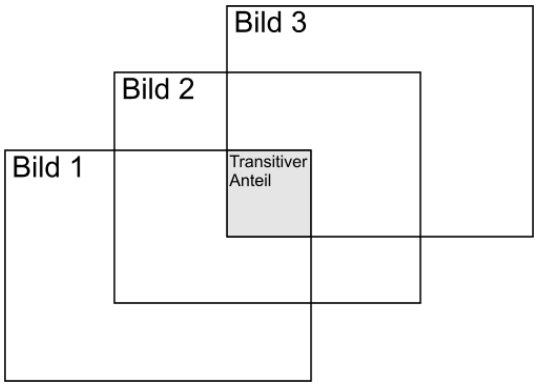


Abbildung 8.9: Transitivität in der Relation B_P

Die zusätzlich benötigte Unterrelation $B_L \subseteq B_P$ muss isomorph zu einer linearen Kette sein. Damit sind alle erforderlichen Eigenschaften (irreflexiv, symmetrisch und nicht transitiv) abgedeckt, auch wenn

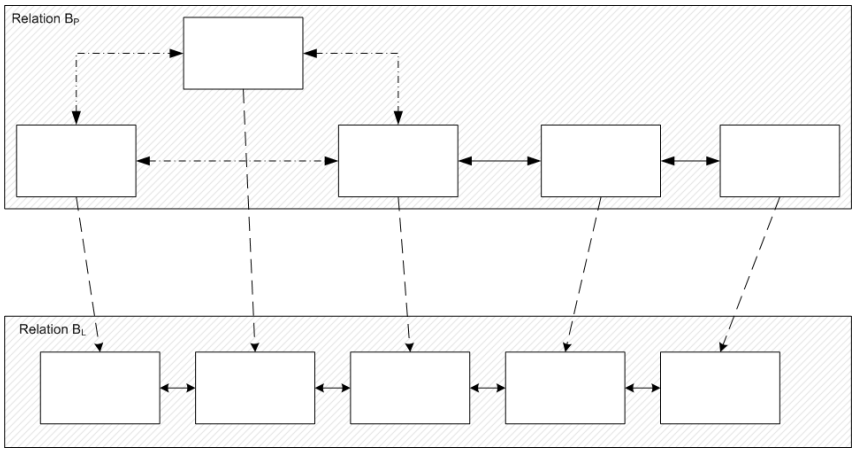


Abbildung 8.10: Bildung der Relation B_L als Isomorphie zur linearen Kette

sich nur zwei Bilder in der Bildmenge befinden. Abbildung 8.10 zeigt die Bildung der Unterrelation B_L und deren Isomorphie zur linearen Kette.
Die Relation B_L bestimmt ebenso implizit die Reihenfolge R, in der die Bilder dem Algorithmus zugeführt werden.
Abschließend sei die zweistellige Relation $G \subseteq S \times S$ definiert als:

$$G \subseteq \{(s_n, s_m) | s_n, s_m \in S\} \tag{8.16}$$

mit den Bedingungen:

$$\begin{aligned} &\forall (s_n, s_m) \in G : \exists b_1 \in B, \exists b_2 \in B \text{ sodass} \\ &(t(b_1, s_n) = 1 \wedge t(b_2, s_m) = 1) \Rightarrow (b_1, b_2) \in B_L \end{aligned} \tag{8.17}$$

und

$$\forall s \in S \; \exists (s_n, s_m) \in G \Rightarrow (s = s_n \vee s = s_m) \tag{8.18}$$

G ist irreflexiv, symmetrisch und, als Folgerung aus der Tatsache, dass die Menge B_L nicht transitiv ist, ebenso nicht transitiv. Die Menge G ist also die Zuordnung zweier selektierter Punkte zueinander, wobei jeweils beide Punkte auf zusammengehörigen Bildern der Überlappungsrelation liegen. Die Eigenschaft 8.18 besagt, dass jeder selektierte Punkt Teil der Relation G sein muss. Das hat zur Folge, dass es keine „überflüssige“ Selektion gibt, die keinem Bildpaar zugeordnet ist.

Der geforderte Input[1] für das Stitching ist nun definiert als:

$$I = \{B, B_L, S, t, G, M, \phi, R\} \tag{8.19}$$

Nun fehlen noch folgende Elemente:

- Die geometrische Transformationsfunktion $\phi : canvas \rightarrow canvas$. Das Kapitel 8.7.2 befasst sich mit der Funktion ϕ.
- Die Bestimmung der Reihenfolge der Anwendung der Transformationsfunktion, welche über die Folge R zu definieren ist. Dies ist im Kapitel 8.7.3 behandelt.

8.7.2 Stitching mit projektiven Abbildungen

Nachfolgend soll geklärt werden, warum eine projektive Abbildung als Transformationsfunktion ϕ zwischen zwei Bildern zu verwenden ist. Unter der Annahme, dass eine affine Transformation (siehe Kapitel A.3.1) den Anforderungen genügt, sei Abbildung 8.11c auf Seite 134 als Gegenbeispiel gezeigt. Dabei ist zu sehen, dass die Gitterlinien sich nicht überdecken.
Eine Eigenschaft der affinen Transformation ist, dass das Verhältnis des Abstandes mehrerer Punkte zueinander auf einer Gerade erhalten bleibt (siehe (Burger und Burge, 2006, S. 366)). Dies ist genau der Fakt, der die affine Abbildung für das *Stiching* unbrauchbar macht: Es ist damit nicht möglich, die perspektivische Verzerrung von Objekten (in diesem Fall des planaren Objektes) auszugleichen. Das bedeutet, dass mehrere Punkte, die auf einer Gerade im Raum liegen, nach dem Abfotografieren auf dem Bild nicht dieselben Abstände aufweisen wie im Raum zuvor. Nun sollen durch das Stitching eben diese Abstände wieder hergestellt werden, damit die Fotografien des planaren Objektes in eine orthogonale Projektion überführt werden können.
Aus diesem Grunde sei die Abbildungsfunktion ϕ definiert als:

$$p' = \phi(p) = M_{proj} \cdot p \tag{8.20}$$

wobei $p, p' \in canvas$ und M_{proj} die projektive Transformationsmatrix sei, aufgestellt nach Abschnitt A.3.2. Die Funktion ϕ transformiert

[1] Als Input müssen nicht nur Eingaben des Anwenders gelten, sondern auch Ergebnisse von Berechnungen, die zuvor durchgeführt wurden.

also einen gegebenen Punkt des *Canvas* mittels der projektiven Matrix.
Durch die Wahl einer projektiven Abbildung ergibt sich eine zusätzliche Forderung an die Menge S:

- Es müssen sich pro Bildpaar genau vier gemeinsame Punkte in der Selektionsmenge befinden.
- Jeweils drei dieser Punkte dürfen nicht gemeinsam auf einer Linie liegen.

8.7.3 Stitching-Reihenfolge

Die Wahl der Reihenfolge R des Stitchings bekleidet eine eher untergeordnete Rolle. Es ist denkbar, dass R implizit über die Reihenfolge der Benutzereingaben bestimmt wird. Zumindest entscheidet R, wie die Unterrelation B_L auszusehen hat. Somit stehen R und B_L in direktem Zusammenhang. Abbildung 8.12 auf Seite 135 zeigt zwei Möglichkeiten, in welcher Reihenfolge drei Bilder aneinander angepasst werden könnten. Somit ist die Folge R:

$$R = (b_0, b_1, \ldots, b_n), n \in N, b_i \in B_p \tag{8.21}$$

8.8 Herstellung der Orthogonalprojektion

Zur Umwandlung des zusammengefügten Bildes in eine Orthogonalprojektion - hierbei wird das Bild perspektivisch entzerrt - müssen zusätzliche Eingaben gemacht werden. Hierbei müssen die Ecken des planaren Objektes, also in diesem Fall die Wandecken, selektiert werden. Anschließend gibt es zwei Möglichkeiten:

- Definition der Texturkoordinaten der Wandgeometrie entlang der Wandkanten:
  Hierbei kann die Selektion der Wandecken direkt dazu genutzt werden, die Texturkoordinaten entsprechend der Selektion zu definieren. Wenn dann das Bild als Textur projiziert wird, wird die perspektivische Verzerrung durch die entsprechende Rendering-Plattform durchgeführt.

- Zuschneiden der Textur entlang der Wandkanten:
  Hierfür ist ein Zwischenschritt notwendig: Nämlich die Überführung der zusammengeführten Textur in eine rechteckige oder sogar quadratische Form. Der Anwender muss die Eckpunkte markieren, sodass erneut über projektive Transformation die abgebildete Wand in ein Rechteck überführt wird. Anschließend kann, ohne Zutun des Anwenders, das Bild beschnitten werden, indem die selektierten Punkte nach der Transformation als Bildgrenzen eingesetzt werden.

Abbildung 8.11e auf Seite 134 zeigt diese abschließende Transformation.

8.9 Zusammenfassung des Algorithmus

Nachfolgend sei der gesamte Algorithmus, ausgehend vom Input I, in verbaler Form aufgezeigt.

1. Überführe jeden Punkt s der Selektionsmenge S in das lokale Koordinatensystem des jeweilig zugeordneten Bildes b, indem:

   $$s' = l^{-1} \cdot s \tag{8.22}$$

   wobei l die globale Transformationsmatrix des Bildes b ist. Die Menge S' sei die Menge aller transformierten Selektionspunkte. Nun können die Bilder in lokalen Koordinaten bearbeitet werden.
   Diese Überführung jedes selektierten Punktes hat den Vorteil, dass sämtliche nachfolgenden Operationen in lokalen Bildkoordinaten stattfinden können.

2. Durchlaufe jede Überlappung $(b_u, b_v) \in B_L$ in der gegebenen Reihenfolge R.

   a) Seien $\{s'_0, s'_1, s'_2, s'_3\}$ die gegebenen Punktpaare, die dieser Überlappung zugeordnet sind. Bestimme die Transformationsfunktion ϕ über die gegebenen Punktpaare, wobei die Transformation die Richtung $b_v \rightarrow b_u$ habe.

   b) Zur Umsetzung von Target-To-Source (siehe Kap. 8.3.1), bestimme ϕ^{-1} nach A.3.3.

c) Bestimme die Ausmaße des transformierten Bildes.
Wende hierfür ϕ auf jeden Eckpunkt von b_v an. Die Ausmaße ergeben sich durch die Differenzen der Maxima und Minima der transformierten Eckpunkte in x und y zueinander.
Generiere ein neues Bild b_{result} mit der selben Pixelgröße auf dem *Canvas* wie b_v.
Ein Beispiel: Hat b_{result} eine Breite von 130 auf dem *Canvas* und b_v eine Breite von 100 bei 500 Pixeln, so hat b_{result} $\frac{130}{100} \cdot 500 = 650$ Pixel in der Breite.

d) Führe Target-To-Source-Transformation durch. Durchlaufe hierfür alle Pixel $p_i \in b_{result}$ und

i. wandle die Koordinaten des Pixels p_i in einen Punkt p_c des *Canvas* in homogenen Koordinaten um durch

$$p_c = f(p_i, (b_v, E)) \tag{8.23}$$

ii. transformiere:

$$p'_c = \phi^{-1}(p_c) \tag{8.24}$$

iii. falls $t((b_v, E), p'_c) = 0$ (t ist definiert in Def. 8.12), gehe über zum nächsten Pixel

iv. sonst bestimme Farbwert c entweder über direkte Bestimmung oder Interpolation (vgl. γ in Def. 8.13).

e) Transformiere jeden auf b_v liegenden Selektionspunkt mit der Funktion ϕ. Dies sorgt dafür, dass das nächste Bildpaar korrekt an das eben transformierte angepasst wird. Es wird also sicher gestellt, dass die selektierten Punkte sich entsprechend mit transformieren.

f) Ersetze b_v durch b_{result}.

3. Verschiebe alle resultierenden Bilder so, dass die nun transformierten Selektionspunkte sich überdecken.

4. Führe perspektivische Entzerrung in Orthogonalprojektion durch:

a) Erwarte vier selektierte Punkte, die die Wandecken markieren, als zusätzlichen Input.

b) Bestimme die Zielkoordinaten der Wandecken: Entweder als Quadrat oder als Rechteck, je nach Projektionsart.

c) Nutze korrespondierende Punktpaare (Wandecke und Zielkoordinate), um die projektive Matrix der Funktion ϕ neu zu bestimmen.

d) Führe Algorithmus äquivalent durch (Bilderzeugung, Target-To-Source-Transformation).

8.10 Fazit

Der Texturededitor ist ein wichtiges Werkzeug zur Überführung von Fotografien der Innenwände eines realen Raumes in die entsprechenden Orthogonalprojektionen. Das Resultat kann als Textur für die virtuelle Präsentation verwendet werden, indem das Bild der korrespondierenden Wand zugeordnet wird. Der Texturededitor stellt den höchsten Anspruch innerhalb der gesamten Arbeit dar: Sowohl die theoretischen Grundlagen als auch die Implementierung machten einen Großteil der Gesamtzeit aus. Insgesamt gesehen ist der Texturededitor eines der wichtigsten Bestandteile des gesamten Konzeptes. Mit ihm ist es möglich:

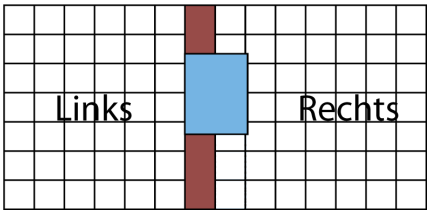
- Eine Linsenkorrektur für einzelne Fotografien innerhalb der Editorumgebung zu machen.
- Mehrere Fotografien anzuordnen.
- Ein manuelles *Stitching* durchzuführen, um mehrere Fotografien zusammen zu fügen. Dies ist auch über mehrere Zwischenschritte möglich, wobei jeweils ein Bild erzeugt wird, welches für den nächsten Schritt verwendet werden kann.
- Ein Bild in eine rechteckige Form zu überführen, indem vier Punkte durch den Anwender markiert und das resultierende Viereck durch das System in ein Rechteck überführt werden. Dies ist für die Überführung in die Orthogonalprojektion notwendig.
- Eine Beschneidung am finalen Bild durchzuführen, um nur die wichtigen Bildinhalte zu erhalten. Bei dem finalen Bild soll nur der zur Wand gehörende Anteil gespeichert werden. Dies kann durch die Beschneidung erreicht werden.

Es gibt jedoch einige Punkte, welche noch als Einschränkung gelten:

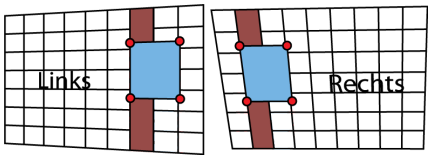
- Ungenauigkeit
  Durch die manuelle Platzierung der Markierungen können Ungenauigkeiten zu schlechten Ergebnissen führen.

- Fehlender Automatismus
  Eine Kombination aus einer automatischen Merkmalserkennung und einer manuellen Korrektur würde das Gesamtkonzept beschleunigen.

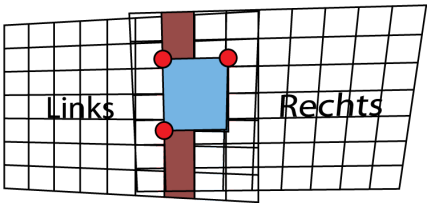
- Hauptfehlerquelle: Linsenkorrektur
  Die derzeit implementierte Linsenkorrektur arbeitet stark approximativ über eine lineare Funktion. Üblicherweise sind Linsen aber über eine nichtlineare Annäherung besser zu beschreiben, ebenso besitzt jede Linse bestimmte Eigenschaften, welche in diese Korrektur mit einfließen müssen, um ein bestmögliches Ergebnis zu erzielen.



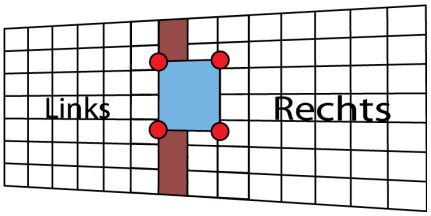
(a) Das Ausgangsobjekt in orthogonaler Projektion



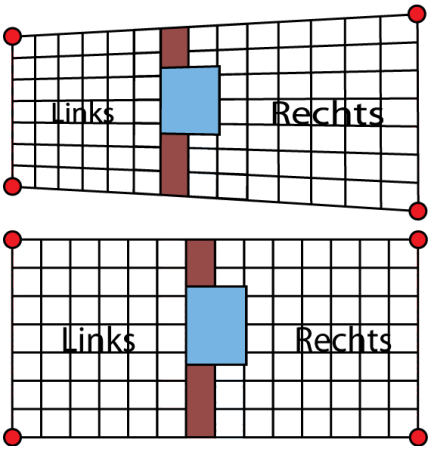
(b) Fotografien des Objektes



(c) Ergebnis nach affiner Transformation



(d) Ergebnis nach projektiver Transformation



(e) Rückführung in orthogonale Projektion

Abbildung 8.11: Auswirkungen der verschiedenen Transformationen.

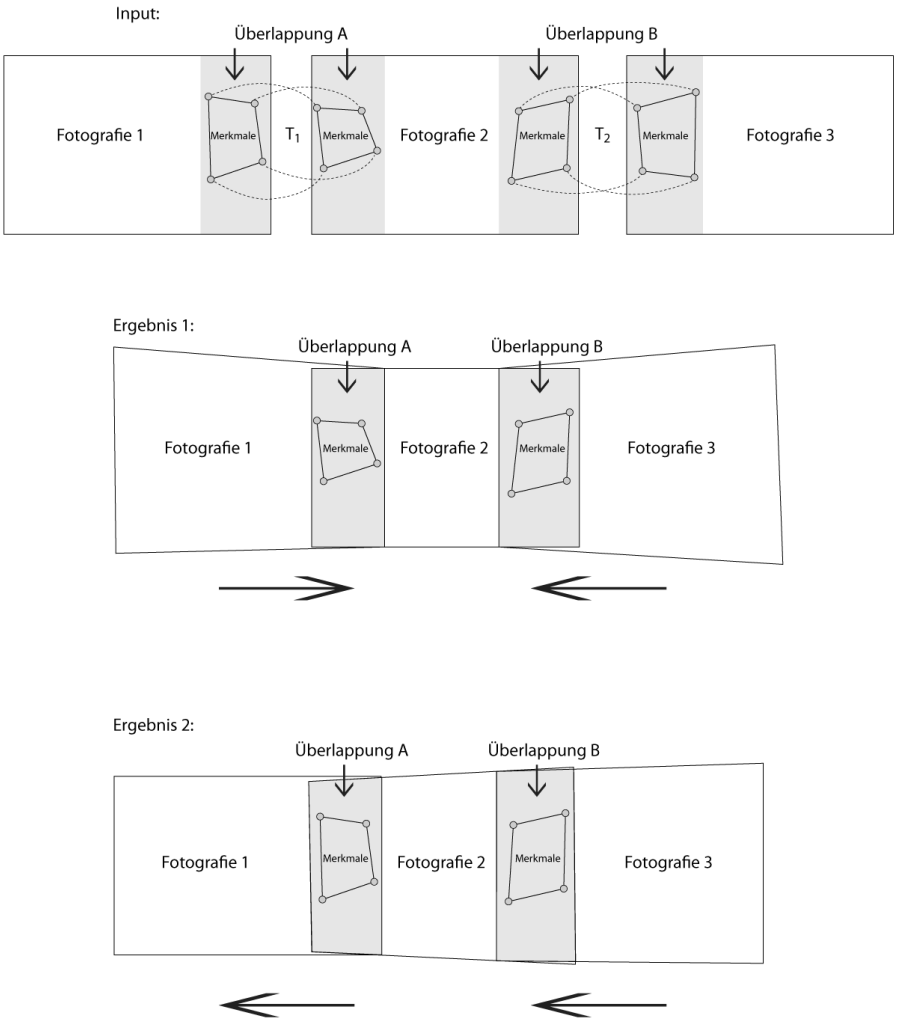


Abbildung 8.12: Zwei Varianten der Stitching-Reihenfolge

9 Aktionen und Reaktionen

9.1 Grundprinzip

Nach Kapitel 2.3.1 ist der Grad der Interaktivität einer Visualisierung ein wichtiger Bestandteil der Visualisierung selbst: Dieser Grad entscheidet darüber, wie viel ein späterer Betrachter in die Visualisierung eingreifen und mit ihr interagieren kann. Aus der Sicht des Erzeugers steht fest, dass ein hoher Grad an Interaktivität auch andere Anforderungen - meist eine höhere Komplexität - abverlangt. Um dem entgegenzuwirken, mussten im Rahmen der Entwicklung von ArchiViz Konzepte entwickelt werden, um die Erzeugung von Visualierungen mit hohem Interaktivitätsgrad möglichst einfach zu gestalten.
Zunächst seien beispielhaft einige mögliche Arten der Interaktion zwischen Betrachter und einer Visualisierung, in diesem Fall einer dreidimensionalen Szene, wie sie z. B. durch ArchiViz erzeugt wird, aufgezeigt:

- Bewegung der Kamera
  Dies bedeutet, dass der Betrachter z. B. durch ein Eingabegerät - z. B. die Maus - Richtung und Position der Kamera beeinflussen kann.

- Anwählen von Objekten
  Ein Mausklick auf ein Objekt kann zu einer Interaktion mit der Visualierung führen. Die entsprechende Reaktion darauf ist Teil der notwendigen Definition durch den Erzeuger.

- Bewegen von Objekten
  Es ist durchaus denkbar, dass dem Betrachter die Möglichkeit gegeben wird, Objekte zu verändern, z. B. zu bewegen oder zu drehen. Das bedeutet allerdings, dass der Betrachter die Struktur der Visualisierung anpassen kann. Dies erfordert erweiterte

Mechanismen, um die Visualisierung konsistent[1] zu halten.

Diese ersten beiden Punkte sind recht typisch für Visualisierungen hoher Interaktivität: Der Betrachter kann sich innerhalb einer virtuellen Szene frei bewegen (Steuerung der Kamera) und dabei Objekte auswählen. Der dritte Punkt, das Bewegen und Verändern von Objekten, gibt dem späteren Betrachter sehr viele zusätzliche Möglichkeiten zu interagieren, jedoch ist dies nicht immer erwünscht. Ist nur eine reine Darstellung gefordert, ist dieser Punkt nicht mehr notwendig und sollte ausgespart werden.
Nun ist solch eine Interaktion zweiseitig zu betrachten: Führt der Anwender eine bestimmte Aktion aus, kann es durch die Visualisierung zu einer Reaktion kommen. Diese Reaktionen müssen ebenso definiert werden wie die Aktionen, die diese auslösen. Hierbei gibt es keine Beschränkungen: Eine Aktion kann mehrere Reaktionen und mehrere Aktionen die selbe Reaktion auslösen.
Das nachfolgend beschriebene Konzept soll ein Werkzeug zur Definition solcher Aktionen und Reaktionen erläutern und anschließend auf eine Beispielimplementierung, durchgeführt im Rahmen des Plugins ArchiViz, eingehen. Hierbei wird darauf Wert gelegt, dass die Handhabung der Definition möglichst einfach gestaltet ist.

9.1.1 Aktionen und Reaktionen als Graph

Zunächst ist als Grundlage zu betrachten, dass jeder Reaktion mindestens eine Aktion zugeordnet ist, somit ist eben jene Aktion Auslöser der Reaktion. Dabei kann allerdings eine Aktion mehreren Reaktionen zugeordnet sein. Das hier vorgestellte Konzept geht noch einen Schritt weiter, wonach Reaktionen selbst andere Reaktionen auslösen können. Damit ist die Aktion initialer Auslöser ganzer Reaktionsketten.
Betrachtet man diese Ketten als Graph, sind die Aktionen und Reaktionen Knoten und die Beziehung „löst aus“ die jeweiligen Kanten. Dieses Auslösen kann seinerseits unterschiedliche Gründe haben:

[1] Konsistenz soll sicherstellen, dass die Interaktionen nicht zu unerlaubten Zuständen der Visualisierung führt. Hierbei ist eine Visualisierung mit einem hohen Grad an Interaktivität schwieriger konsistent zu halten, da die Anzahl der möglichen Zustände der gesamten Visualisierung schwerer zu überblicken ist.

- Automatisches Auslösen
  bedeutet, dass nicht der Betrachter entscheidet, dass die nächste Reaktion ausgelöst wird. Es ist dabei ein Algorithmus notwendig, der diese Entscheidung übernimmt (z. B. über die Auswertung bestimmter Bedingungen).

- Manuelles Auslösen
  gibt dem Anwender die Entscheidung in die Hand, wann die nächste Reaktion ausgelöst werden soll.

Der Unterschied zwischen den Begriffen „Auslösen“ und „Auswerten“ besteht darin, dass das Auslösen den Übergang an sich zwischen zwei Knoten beschreibt, während das Auswerten sich auf die Auswertung eines Knotens bezieht.
Es gibt also zwei Arten der Auswertung der verschiedenen Reaktionen eines Knotens:

- Sequentielle Auswertung
  Dies ist äquivalent zu einem Übergang von Knoten zu Knoten, welcher unmittelbar erfolgt. Es wird genau eine Reaktion ausgewertet.

- Parallele Auswertung
  Auswertung aller Reaktionen innerhalb eines Knotens.

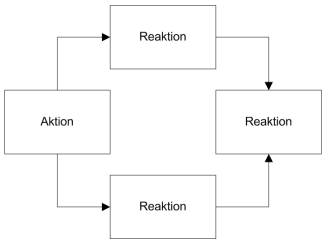
Abbildung 9.1 zeigt einige Ausprägungen eines Aktions-Reaktions-Graphen, diese seien nachfolgend kurz erläutert.

- Abbildung 9.1a zeigt das Grundprinzip, wie eine Aktion mehrere Reaktionen auslöst. Dabei können zwei Reaktionen zugleich eine weitere Reaktion auslösen. Jedem Reaktionsknoten ist genau eine Reaktion zugeordnet.

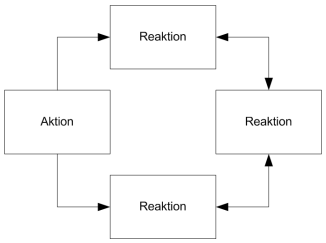
- Neben unidirektionaler Auswertung besteht die Möglichkeit der bidirektionalen Auswertung: es sind Übergänge in beide Richtungen erlaubt. Abbildung 9.1b zeigt eine Erweiterung des Graphen um bidirektionale Übergänge.

- Abbildung 9.1c erweitert den Graphen um weitere Knoten mit dem Hintergrund, dass eine Aktion mehrere Reaktionen in Folge auslöst. Dies führt zu einer sequentiellen Auswertung mehrerer Reaktionen in Folge.

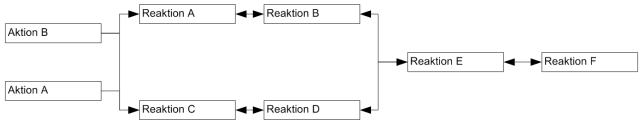
- Im vorherigen Punkt löst eine Aktion mehrere Reaktionen in sequentieller Folge aus. Dieses Verhalten kann jedoch noch darum erweitert werden, dass mehrere Reaktionen parallel ausgeführt werden. Somit werden einem Reaktionsknoten mehrere Reaktionen zugeordnet, welche zugleich ausgewertet werden. Die Kombination aus sequentieller und paralleler Abarbeitung ist in Abbildung 9.1d dargestellt. Knoten werden sequentiell hintereinander ausgeführt (Durchlaufen des Graphen über die Kanten) und zugleich werden mehrere Reaktionen parallel abgearbeitet (mehrere Reaktionen sind innerhalb eines Knotens definiert).



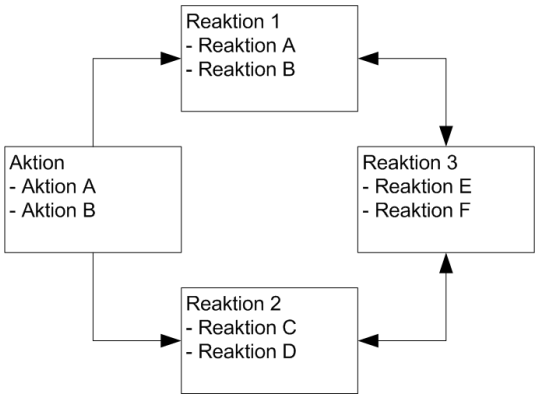
(a) Einfache Reaktionskette



(b) Kette mit Rückrichtung



(c) Sequentielle Auswertung



(d) Sequenzielle und parallele Auswertung

Abbildung 9.1: Verschiedene Ausprägungen von Reaktionsketten

9.1.2 Umsetzung für ArchiViz

Es liegt nahe, dem Erzeuger der Visualisierung ein grafisches Werkzeug zur einfachen Definition solcher Aktionen und Reaktionen in die Hand zu geben. Das ermöglicht, Menschen ohne Programmiererfahrung die Arbeit enorm zu erleichtern, denn es wäre auch möglich, Aktionen und Reaktionen z. B. über eine Skriptsprache[1] definieren zu lassen. Für ArchiViz wurde ein solches grafisches Werkzeug auf Grundlage des 2D-Canvas (siehe Kap. 5) umgesetzt. Dieses Werkzeug beinhaltet folgende Funktionalität:

- Erzeugung von Aktions- und Reaktionsknoten
- Verbindung von Aktionen und Reaktionen, also Erzeugung von Kanten zwischen den Knoten
- Zuordnung spezifischer Aktionen bzw. Reaktionen zu einem Knoten

Darüber hinaus war zu wählen, welche Aktionen und Reaktionen, in Bezug auf die Anforderung, sinnvoll sind. Als einzige mögliche Aktion wurde „Klick auf Objekt“ umgesetzt. Dies orientiert sich an den technischen Möglichkeiten GroIMPs. Bisher ist es nicht möglich, direkt Reaktionen auf bestimmte Ereignisse in GroIMP zu implementieren, dazu gehört die Reaktion auf Mausklicks im 3D-Fenster. Somit war es nicht ohne weiteres möglich, „Doppelklick“ oder „Rechtsklick“ als Aktionen umzusetzen.
Schlussendlich ist es das Selektionsereignis in GroIMP, das als „Klick auf Objekt“ interpretiert wird. Hierfür besitzt GroIMP Mechanismen, um auf eine Selektion reagieren zu können[2]. Prinzipiell wäre es aber möglich - mit erheblich mehr Aufwand - weitere Aktionen zu implementieren, auf die entsprechend reagiert werden kann. Es ist auch denkbar, dass bestimmte Buttons in GroIMP als Aktion dienen könnten.
Erheblich mehr Möglichkeiten boten sich bei der Implementierung verschiedener Reaktionen: Da diese Reaktionen in erster Linie ähnlich wie ArchiViz selbst realisiert werden konnten, war der Aufwand hierfür begrenzt. Folgende Reaktionen sind derzeit verfügbar:

- Zeige Text (Show Text)
  öffnet ein Textfenster, das einen zuvor definierten Text anzeigt.

[1] Z.B. Python oder Perl
[2] Siehe Kap. 4.7.5

- Zeige Bild (Show Image)
  öffnet einen 2D-Canvas, der ein zuvor der Szene hinzugefügtes Bild darstellt.

- Wechsle Zustand[1] (Change State)
  wechselt in einen zuvor definierten Zustand.

- Bewege Kamera (Move Camera)
  bewegt die Kamera an einen definierten Standort. Das ermöglicht z. B. das Wechseln der Räume während einer Präsentation, indem der Betrachter auf eine Wand klickt. Hierfür muss der Wand entsprechend eine Aktion zugeordnet worden sein.

Jeder Reaktion ist eine entsprechende Eingabemöglichkeit zur Verfügung gestellt: Z.B. ein simpler Texteditor oder ein Auswahlfenster für Bilder. Die definierten Aktionen bzw. Reaktionen und die zugehörigen Kanten werden zusammen mit der Szene gespeichert.

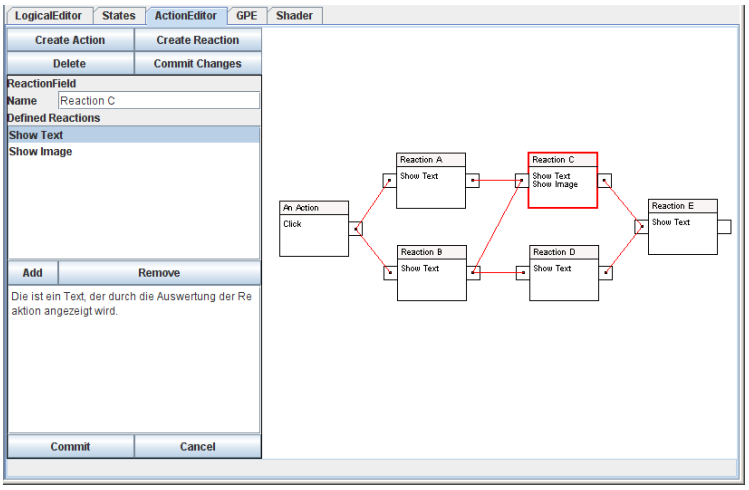


Abbildung 9.2: Grafisches Werkzeug „Action-Editor“ in ArchiViz

Abbildung 9.2 zeigt die implementierte Benutzeroberfläche. Die genaue Verwendung erläutert Kapitel 11 genauer.

[1]Das Zustandskonzept ist in Kapitel 10.1 erläutert

9.1.3 Verankerung in GroIMP

Die Struktur des Aktions-Reaktions-Graphen wurde eins zu eins auf den GroIMP-Szenegraphen abgebildet. Das bedeutet: Ein definierter Aktionsknoten oder Reaktionsknoten wird zu einem Knoten im Szenegraphen von GroIMP, darüber hinaus werden alle Kanten übertragen. Darüber wird der Aktions-Reaktions-Graph immer einem Objekt zugeordnet. Dies ist deshalb sinnvoll, da die verfügbare Aktion „Klick auf Objekt“ objektbezogen ist. Dieses Objekt, dem die Aktion zugeordnet ist, ist Vater des Aktionsknotens und somit Wurzelknoten des Aktions-Reaktions-Graphen. Es wird also pro Objekt solch ein Graph, falls benötigt, angelegt.
Insgesamt ist also der implementierte Editor eine Sicht auf einen Teil des GroIMP-Szenegraphen und editiert diesen. Bisher ist es in GroIMP nicht möglich, Knoten und Kanten direkt auf diese Art zu beeinflussen: Es gibt nur eine Ansicht des Szenegraphen als 2D-Graph.
Abbildung 9.3 zeigt den bereits in Abbildung 9.2 dargestellten Bei-

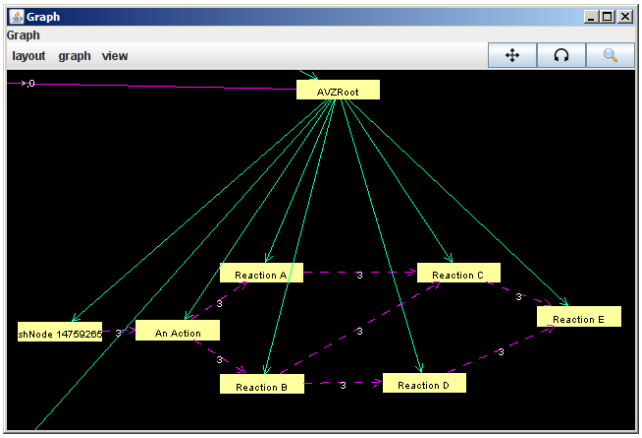


Abbildung 9.3: Ausschnitt des Szenegraphen in GroIMP

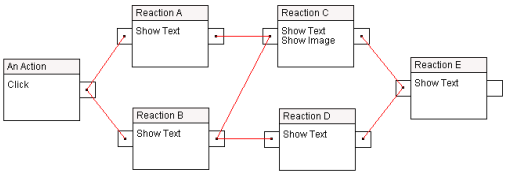
spielgraphen. Zwischen den Knoten werden spezielle Kanten verwendet, während jeder Knoten eine `successor`-Edge (vgl. Kap. 4) vom Wurzelknoten des ArchiViz-Graphen besitzt. Dieser Fakt ist in einigen Hindernissen in der Implementierung begründet und behindert das Gesamtkonzept nicht. Grundsätzlich ist die Struktur zu der im Editor definierten identisch, wobei im Editor nur die notwendigen Kanten angezeigt und editiert werden.
Anmerkung: Da der ArchiViz-Graph Maschen enthält, arbeiten die

in GroIMP implementierten Algorithmen zur automatischen Anordnung der Knoten in der Graph-Sicht GroIMPs fehlerhaft.

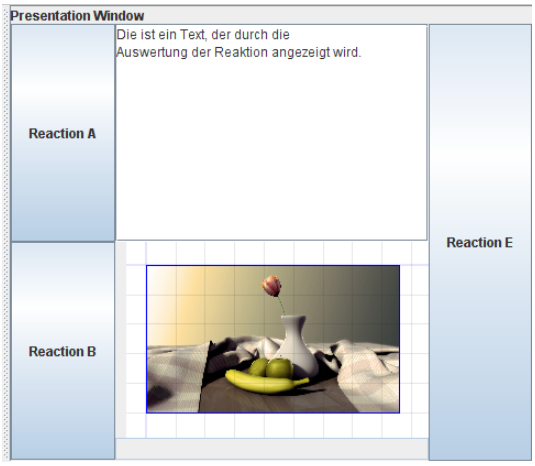
9.1.4 Auswertung im Präsentationsmodus

ArchiViz enthält einen in GroIMP verankerten Präsentationsmodus, welcher dazu entwickelt wurde, den Aktions-Reaktions-Graph auszuwerten. Das bedeutet: Wird eine Aktion wie „Klick auf Objekt“ ausgeführt, werden alle Knoten im ArchiViz-Szenegraph darauf überprüft, ob ihnen ein Aktions-Knoten zugeordnet ist, welcher die entsprechende Aktion enthält. Ist dies der Fall, kann der Aktions-Reaktions-Graph ausgewertet werden. Nachfolgend sei erläutert, wie die Übergänge zwischen Reaktionsknoten dem Anwender zugänglich gemacht werden können. Das bedeutet, dass der Anwender manuell entscheiden kann, welcher Reaktionsknoten als nächstes ausgewertet werden soll, während die Struktur des Graphen über die verfügbaren Optionen entscheidet. Somit ist jede Kante des Graphen solch eine Option, die dem Anwender zur Verfügung gestellt werden kann bzw. muss. Interpretiert man diese Optionen als simples grafisches Interface, ist jede Kante ein Button, dessen Betätigung durch den Anwender den Übergang auslöst. Bezieht man mögliche Reaktionen auf GUI-Funktionen, könnten Reaktionen wie „Zeige Text“ oder „Zeige Bild“ als entsprechende GUI-Elemente angezeigt werden. Eine definierte Reaktion wie „Zeige Text“ wird z. B. als ein anzuzeigendes Textfenster im Präsentationsfenster interpretiert. Sollen mehrere Reaktionen parallel ausgewertet werden, und diese beziehen sich auf GUI-Elemente (wie z. B. ein Text- oder Bildfenster), werden eben diese Elemente zugleich angezeigt. Natürlich sind dieser Auswertung einige Grenzen gesteckt: Zu viele Reaktionen, welche GUI-Elemente erzeugen, können nicht zugleich anzeigt werden. Es kann jedoch z. B. durch Scrollen Platz geschaffen werden.

Eine Auswertung eines Reaktionsknotens zeigt Abbildung 9.4, wobei Abbildung 9.4a einen Beispielgraph darstellt und Abbildung 9.4b die Auswertung des Knotens „Reaktion C“. Diesem Reaktionsknoten sind zwei parallele Reaktionen zugeordnet: „Show Text“ zeigt einen Text und „Show Image“ ein Bild an. Im Graphen ist zu sehen, dass der Reaktionsknoten zwei Vorgängerknoten, nämlich „Reaktion A“ und „Reaktion B“ und einen Nachfolger, nämlich „Reaktion C“ hat. Diese möglichen Übergänge werden als Buttons angezeigt und mit den entsprechenden Knotennamen beschriftet. Die Auswertung des



(a) Ein Graph



(b) Auswertung des Knotens C als GUI

Abbildung 9.4: Aktions-Reaktions-Graph wie für ArchiViz umgesetzt

Knotens „Reaktion C“ und dessen enthaltenen definierten Reaktionen „Show Text“ und „Show Image“ befinden sich in der Mitte der GUI.
Klickt der Anwender auf einen der Buttons, wird entsprechend der gewählte Reaktionsknoten ausgewertet.

9.2 Fazit

Das für ArchiViz umgesetzte Prinzip des Aktions-Reaktions-Graphen gibt dem Erzeuger einer Präsentation ein flexibles Werkzeug in die Hand, um gezielt Informationen darzustellen und verschiedenartige Reaktionen auf Aktionen des Anwenders auf äußerst einfache Art und Weise zu definieren. Darüber hinaus kann der spätere Betrachter selbst entscheiden, welche Knoten - sofern dies entsprechend definiert ist - für ihn interessant sind. Ebenso ist die Auswertung dieses Graphen eine einfache GUI-Logik, welche ohne Programmier-

kenntnisse erzeugbar ist.
Nachteil des gegenwärtigen Bearbeitungsstandes ist die Beschränkung auf nur eine Aktion, nämlich den Klick auf ein Objekt. Wenn man die Menge der möglichen Aktionen schrittweise erweitert und so die Funktionsvielfalt weiter erhöht, kann der Aktions-Reaktions-Graph und dessen Auswertung zu einem sehr flexibel einsetzbaren Werkzeug werden.

10 Zustandseditor

10.1 Zustände von Knoten

Ein grundlegender Gegenstand der Aufgabenstellung ist der Vergleich mehrerer Zustände eines Innenraums. Dies betrifft den Raum selbst (also deren Wandtextur) und die darin befindlichen Objekte.
Ein Zustand ist im Grunde genommen die Variablenbelegung aller Variablen eines konkreten Objektes im Speicher zu einem bestimmten Zeitpunkt. Da die Anforderung nur die Speicherung der Position, der Textur und der Sichtbarkeit vorsieht, kann der Zustand auf diese Elemente bzw. auf die dazugehörigen Variablen reduziert werden. Es ist also nicht notwendig, den gesamten Zustand zwischen zu speichern, sondern nur ausgewählte Attribute.
Da GroIMP noch kein Konzept zur Speicherung verschiedener Zustände eines Knotens hat, musste ein entsprechendes Prinzip entwickelt werden. Zur Vereinfachung geschah dies zugeschnitten auf ArchiViz, eine Verallgemeinerung auf GroIMP steht noch aus. Es ist damit zu rechnen, dass diese sehr zeitaufwändig wird, aber es ist zu vermuten, dass dies in prinzipiell gleicher Weise wie für ArchiViz bearbeitet werden kann.
Zunächst musste über eine grundlegende Arbeitsweise entschieden werden. Die folgenden Grundlagen bilden diese Arbeitsweise.
Gegeben ist ein beliebiges Objekt einer ArchiViz-Szene, repräsentiert durch einen Knoten im Szenegraphen GroIMPs. Dieses Objekt hat verschiedene Eigenschaften, wovon drei von besonderem Interesse sind:

- Transformation
  Die Transformation (vgl. Kap. A.11) im dreidimensionalen Raum beinhaltet die Position, die Rotation und die Skalierung. Diese drei Eigenschaften sind in der Transformation, die der Knoten innehat, gemeinsam gespeichert. Deshalb können diese auch als eine zusammenhängende Eigenschaft bezeichnet werden.

- Textur
  In GroIMP werden Texturen über Shader zugewiesen (Vgl. Kap. 4.2.1), sodass ein Bild z. B. über den Diffuse-Kanal als Textur eingebunden wird. Dieser Kanal wird zudem in der Echtzeitdarstellung von GroIMP via OpenGL dargestellt.

- Sichtbarkeit
  GroIMP stellt keinen Mechanismus zur Verfügung, um einzelne Objekte auszublenden. Es gibt zwar das Konzept der verschiedenen Layer, jedoch wäre dies für die hier betrachtete Aufgabenstellung keine gute Lösung: Der Anwender hat von außen vollen Zugriff auf die verschiedenen Layer und könnte durch seine Eingaben das Konzept stören.

Das zu entwickelnde Konzept soll die Möglichkeit schaffen, verschiedene Zustände für einzelne Objekte (also Knoten im Szenegraphen) zu definieren und zwischen ihnen umzuschalten. Ebenso soll der Anwender die Attributwerte innerhalb des Zustandes eines Objektes zuweisen bzw. verändern können.
Darüber hinaus müssen diese Funktionalitäten in einer GUI zur Verfügung gestellt werden, damit der Anwender volle Kontrolle über die verschiedenen Zustände erhält.

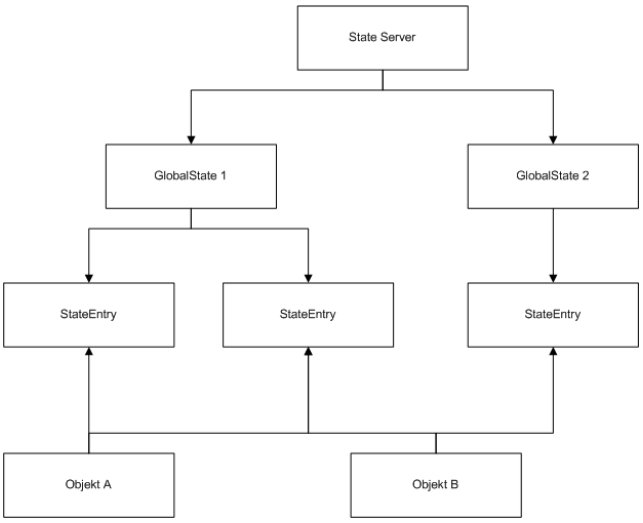


Abbildung 10.1: Schema des Zustandsgraphen

10.2 Verankerung in GroIMP

Das gesamte Prinzip wurde so entwickelt, dass es als Aufsatz zu dem vorhandenen ArchiViz-Szenegraphen funktioniert. Es hat keinerlei Einfluss auf bisherige Knotenbeziehungen in irgendeiner Form. Abbildung 10.1 zeigt beispielhaft, wie die verschiedenen Knoten und Kanten angeordnet sind. Zunächst müssen global[1] übergeordnete Zustände definierbar sein. Ein solcher globaler Zustand wird durch einen `AVZGlobalStateNode` repräsentiert. Diesen Zuständen können „normale“ Szenegraph-Knoten zugeordnet werden, sodass die Beziehung „Knoten A wird durch Zustand B beeinflusst“ modelliert wird. Ist dies für einen Knoten der Fall, werden alle für diesen Zustand relevanten Informationen gespeichert und dem Zustand (also dem `AVZGlobalStateNode`) entsprechend zugeordnet. Diese Speicherung und Zuordnung wurde über einen weiteren Knoten realisiert: `AVZStateEntryNode`. Wie der Abbildung 10.1 zu entnehmen ist, gruppiert ein globaler Zustand mehrere andere Knoten vom Typ `AVZStateEntryNode` unter sich. Ein Objekt, das nun einem globalen Zustand zugeordnet wird, bekommt in Wirklichkeit einen `AVZStateEntryNode` zugeordnet. Dieser speichert letztendlich die für diesen globalen Zustand relevanten Daten.
Der Knoten `AVZStateServer` gruppiert wiederum alle globalen Zustände, sodass alle Zustandsknoten einen separaten Teilgraphen des gesamten Szenegraphen bilden. In einer Szene gibt es nur einen einzigen `AVZStateServer`.
Bevor überhaupt andere globale Zustände definiert werden können, muss zunächst der initiale globale Zustand angelegt werden. Dieser repräsentiert den „Rohzustand“ der Szene, noch bevor der Szenegraph dem Zustandskonzept zugeführt wurde. Wird ein Objekt einem globalen Zustand zugeordnet, wird für dieses Objekt zunächst ein Knoten vom Typ `AVZStateEntryNode` angelegt und die aktuellen Werte initial gespeichert. Somit ist es möglich, das Objekt aus dem Zustandskonzept wieder zu entfernen und den Urzustand wieder herzustellen.
Im nächsten Schritt wird ein weiterer Knoten vom Typ `AVZStateEntryNode` angelegt und dem entsprechenden globalen Zustand zugeordnet. Dies geschieht über das Anlegen einer Kante von dem globalen Zustand zu dem neuen `AVZStateEntryNode`. Ebenso wird das Objekt mit dem `AVZStateEntryNode` verbunden.

[1] *Global* bedeutet hier: Szenenweit.

Nach diesen Schritten entspricht der Szenegraph dem in Abbildung 10.1 dargestellten Graphen. Abbildung 10.2 zeigt einen solchen Graphen beispielhaft in GroIMP. Nun ist noch zu erläutern, wie und

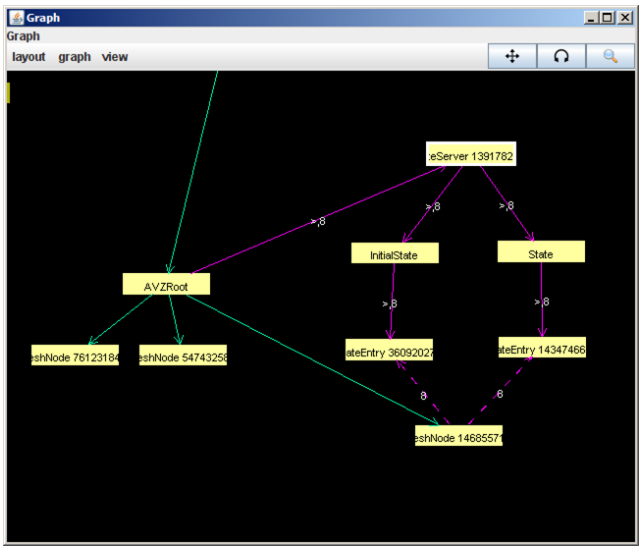


Abbildung 10.2: Beispielgraph von Zuständen in GroIMP

wann GroIMP die Daten speichert. An dieser Stelle gibt es mehrere Ansätze:

- Manuell
  Der Anwender muss selektieren, welches Attribut in welchem Zustand mit welchen Werten zu versehen ist.

- Automatisch
  Es wird ein Zustand als „aktiv“ erklärt (z. B. durch den Anwender). Wird nun ein Objekt, das diesem Zustand zugeordnet ist, verändert (einschränkend gesagt: eines der Attribute Transformation, Textur oder Sichtbarkeit), werden diese Änderungen automatisch aufgezeichnet und gespeichert.

Für *ArchiViz* wurde eine Mischung aus beiden Prinzipien umgesetzt. Zunächst wird ein Zustand als aktiv erklärt. Nun kann der Anwender manuell auswählen, welche der Attribute aufgezeichnet werden. Wird dann eines der ausgewählten Attribute verändert - z. B. durch Interaktion durch den Anwender oder als Folge anderer Interaktionen - , werden diese Veränderungen gespeichert.

Die Speicherung verläuft folgendermaßen: Es ist für das jeweilige Attribut ein *Listener* registriert. Bei einer Veränderung wird ein Event ausgelöst und der aktuelle Wert aus dem Knoten ausgelesen und im entsprechenden *Entry*-Knoten gespeichert.
Das Umschalten zwischen Zuständen hat folgenden Verlauf: Ein Iterieren über alle Knoten vom Typ `AVZStateEntryNode` eines globalen Zustandes liest die Attribute, die in eben diesem Knoten gespeichert sind, aus und schreibt diese in das Objekt. An dieser Stelle gibt es noch eine Besonderheit: Nicht jeder Wert muss in das Objekt geschrieben werden. Es ist möglich, bestimmte Attribute für einen Zustand zu deaktivieren (ohne sie dabei zu löschen).

10.3 Fazit

Das Zustandskonzept entwickelt und implementiert ein Prinzip, welches in GroIMP ursprünglich nicht vorgesehen war. Es ist zunächst auf die Bedürfnisse von ArchiViz zugeschnitten. Eine Erweiterung auf GroIMP ist grundsätzlich möglich, das Kapitel 10.1 beschäftigt sich mit dieser Frage. Insgesamt konnten die wichtigsten Elemente implementiert werden, sodass es möglich ist:

- globale Zustände zu definieren,
- einzelne Objekte in ArchiViz diesen zuzuordnen und
- die Attribute Transformation, Textur und Sichtbarkeit durch den Zustand zu steuern.

Der Zustandseditor ermöglicht es

- neue Zustände anzulegen,
- die Zuweisung der Objekte zu einem Zustand durchzuführen,
- Kontrolle über die überschriebenen Attribute auszuüben und
- zwischen diesen globalen Zuständen umzuschalten.

Was aus Zeitgründen nicht implementiert werden konnte, war die Einbeziehung der Kamera in das Zustandskonzept, sodass die Transformation (also die Position und Drehung) der Kamera durch einen globalen Zustand gesteuert werden kann.

In Kombination mit dem Aktions-Reaktions-Editor (vgl. Kap. 9.1) ist das Zustandskonzept auch im Präsentationsmodus nutzbar, indem der Designer im Aktions-Reaktions-Graph als eine Reaktion den Wechsel in einen Zustand einbezieht.

11 Fallbeispiel

Das nachfolgende Beispiel dient zur Darstellung eines kompletten Arbeitsablaufes. Insgesamt wird allerdings nur ein Ausschnitt aller verfügbaren Funktionalitäten eingesetzt, um das Beispiel übersichtlich zu gestalten.

Grundriss importieren

Der erste Schritt besteht darin, einen Grundrissknoten über den Button „Add GP“ zu erzeugen. Dies ist zwingend notwendig, da das spätere Sketching auf diesen Knoten aufbaut. Diesem Knoten kann eine Textur zugeordnet werden, in diesem Fall ein Grundriss.

Schnittlinien (Sketching lines) über Sketching

Zur Erzeugung der Wandgeometrie dient der Groundplan-Editor (GPE). Dieser zeigt als Hintergrund, sofern importiert, den Grundriss an. Über den Button „Sketching“ kann nun begonnen werden, die Schnittlinien zu platzieren. Hierbei verläuft pro Wand eine Schnittlinie. Ein Druck auf die Taste „Enter“ bestätigt das Ergebnis und lässt die Schnittpunkte berechnen. Abbildung 11.1 auf Seite 153 zeigt den importierten Grundriss mit den manuell erzeugten Schnittlinien. Resultat sind die Schnittpunkte, welche zur Raumerzeugung dienen können.

Raumerzeugung

Hierbei werden die erzeugten Schnittpunkte dazu verwendet, Räume zu definieren und die Wandgeometrie festzulegen. Ein Raum wird definiert, indem seine Eckpunkte im Uhrzeigersinn selektiert werden (durch Festhalten der Umschalt-Taste) und anschließend der Button „Create Room“ im GPE betätigt wird. Abbildung 11.2 zeigt beispielhaft zwei definierte Räume, im Grundriss befinden sich diese links oben.

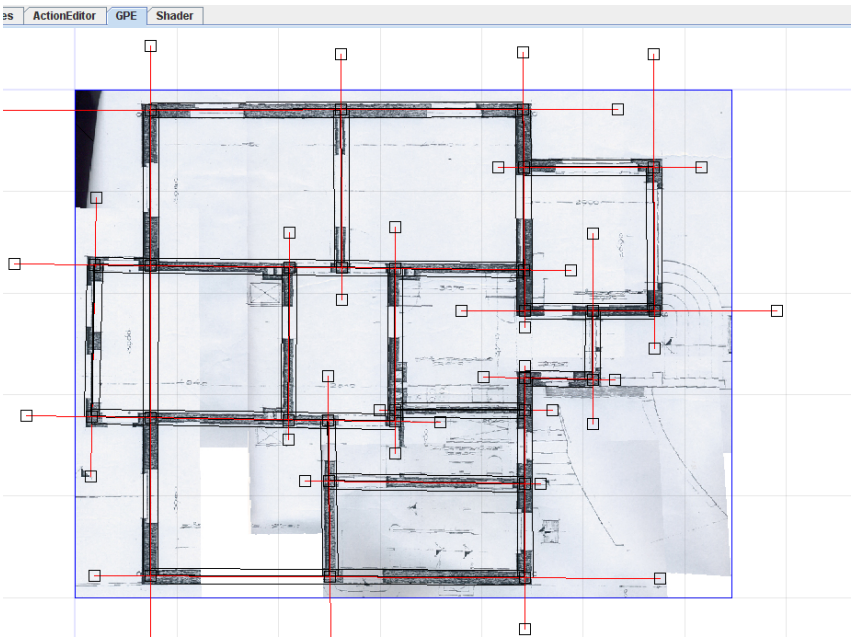


Abbildung 11.1: Erzeugung von Schnittlinien

Fotografien importieren Innerhalb dieses Fallbeispieles wird der Raum links oben im Grundriss weiter ausgebaut. Vor Erzeugung der Präsentation wurden Fotografien aller Innenwände angefertigt, einige davon mussten durch mehrere Fotografien, die sich überlappen, abgedeckt werden. Damit diese Fotografien dem System zur Verfügung stehen, müssen diese über die Shader-Übersicht (Karteireiter „Shader“) importiert werden. Ein Klick auf „Import“ öffnet das in Abbildung 11.3 gezeigte Menü. Nach Auswahl von „Picture“ und Klick auf „Okay“ öffnet sich ein Auswahlmenü, mit dem eine oder mehrere Bilder im Format „JPG“ importiert werden können. Diese erscheinen anschließend, wie in Abbildung 11.4 gezeigt, in der Liste ganz rechts. Dort werden alle Bilder aufgeführt, welche der Szene zugeführt wurden (außer dem Grundriss). Es ist zu beachten, dass ein importiertes Bild automatisch in den Szenenordner im Unterordner „Images“ gespeichert wird. Deshalb erscheint, sofern man das Projekt noch nicht gespeichert hat, eine Aufforderung, zunächst das Projekt inital zu speichern, damit der Projektpfad eindeutig ermittelbar ist.

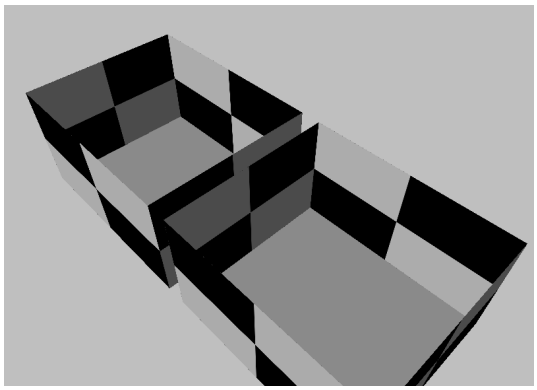


Abbildung 11.2: Erzeugung von Räumen



Abbildung 11.3: Import von Bildern

Stitching

An dieser Stelle soll beispielhaft eine Wand des Raumes, welche mit zwei Fotografien erfasst wurde, mit einer Textur ausgestattet werden. Zunächst müssen die beiden zusammengehörigen Fotografien importiert und gruppiert werden. Der Import findet statt wie bereits beschrieben. Alle initial importierten Fotografien werden automatisch unter der Gruppe „Free Images“ gruppiert. Für das neue Gruppieren muss eine Gruppe („ImageSet“) über den Button „New“ in der Shaderübersicht erzeugt werden. Der mittlere Teil der Übersicht zeigt, welche Bilder einer Gruppe zugeordnet sind. Ein Doppelklick auf die neu erzeugte Gruppe, in der Abb. 11.4 ist dies „wand2“, zeigt, dass diese noch leer ist. Durch Selektion der zugehörigen Bilder und Klick auf den Button „Pfeil nach links“ werden die Bilder unter dem ImageSet „wand2“ gruppiert.

> Anmerkung:
> Ein Doppelklick selektiert ein Bild oder eine Gruppe, ein Dreifachklick öffnet ein Vorschaufenster, welches das Bild anzeigt.

Nach der Gruppierung kann nun der Textureditor geöffnet werden. Hierfür muss eine gefüllte Gruppe selektiert sein (Doppelklick auf

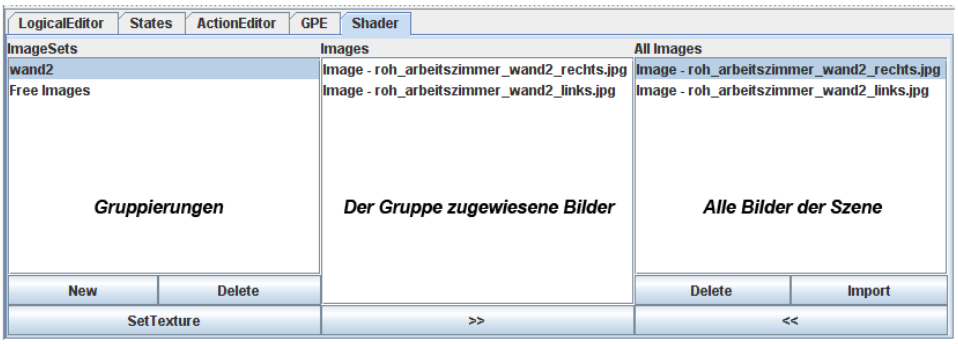


Abbildung 11.4: Importierte Bilder im Shader-Editor

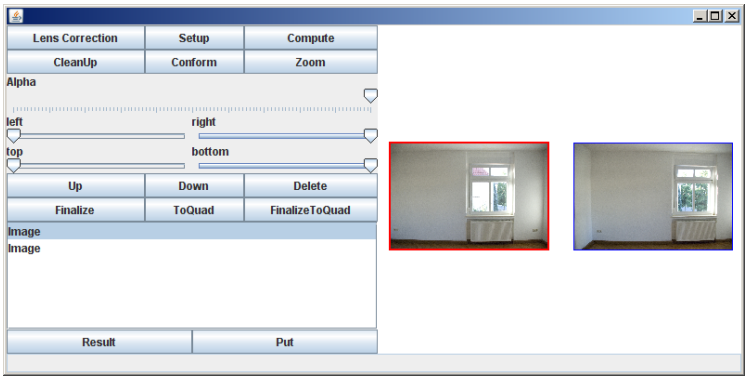


Abbildung 11.5: Der Textureditor

die Gruppe „wand“). Ein Klick auf den Button „Textureeditor“ öffnet den Editor und lädt alle in der Gruppe befindlichen Fotografien. Abbildung 11.5 zeigt den Textureditor nach dem Öffnen.

Der nächste Schritt besteht darin, dass die Linsenverzerrungen der einzelnen Fotografien ausgeglichen werden müssen. Hierfür gibt es einen eigenen Editor, der eine manuelle Korrektur ermöglicht. Geöffnet wird dieser, indem ein Bild selektiert wird (Linksklick auf das Bild) und der Button „Lens Correction“ angeklickt wird. Es öffnet sich der in Abbildung 11.6 gezeigte Editor. Die beiden Regler bilden die beiden Parameter der Transformationsfunktion ab (siehe Kap. 8.6), sodass das Bild separat in der X- und Y-Achse korrigiert werden kann. Ist „Auto“ aktiviert, aktualisiert die Software das Bild mit jeder Veränderung der Parameter, ansonsten kann dies auch über den Button „Compute“ geschehen. Mit einem Klick auf „Finish“ schließt sich der Editor und übergibt die Änderung an den Textureditor. Dieser Weg muss auch für die andere Fotografie durchgeführt werden.

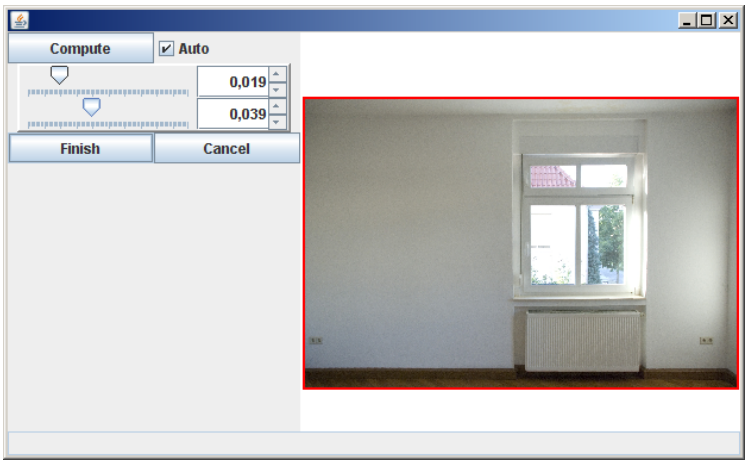


Abbildung 11.6: Editor zur Linsenkorrektur

Wurde diese mit der selben Brennweite fotografiert, können die selben Werte eingesetzt werden.
Nach dem Ausgleichen der Linsenverzerrung kann nun das manuelle Stitching durchgeführt werden. Begonnen wird mit dem Button „Setup“. Dieser initiiert das Stitching-Werkzeug. Nun müssen zunächst vier Merkmale auf dem ersten Bild und die vier korrespondierenden Merkmale auf dem zweiten Bild markiert werden (in der selben Reihenfolge). Abbildung 11.7 zeigt das Resultat. Sind alle acht Klicks durchgeführt (es muss darauf geachtet werden, dass die Klicks auf einem Bild stattfinden), beendet sich das Werkzeug automatisch. Die Software erstellt automatisch eine Zuordnung der an-

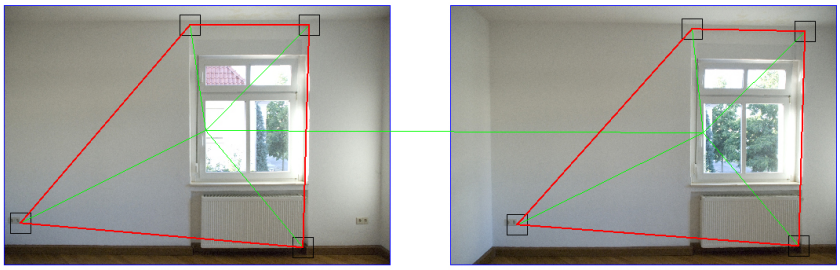


Abbildung 11.7: Markieren der gemeinsamen Merkmale

geklickten Fotografien. Das eigentliche Stitching wird durchgeführt nach einem Klick auf dem Button „Compute“. Resultat sind zwei weitere Bilder, die entsprechend der markierten Merkmale angeglichen und überlappend positioniert werden. Zugleich wird das zweite Bild

unter dem ersten gruppiert, sodass ein Verschieben des ersten Bildes ausreicht, um beide neu auf der Arbeitsfläche zu positionieren. Die Buttons „Up“ und „Down“ verändern die Darstellungsreihenfolge



Abbildung 11.8: Resultat des Stitchings

der Bilder: „Up“ schiebt ein Bild um eine Stufe in den Vordergrund, „Down“ in den Hintergrund. Somit kann manuell festgelegt werden, in welcher Reihenfolge die Bilder übereinander gelegt werden.
Nun müssen die Bilder zu einem Bild zusammengefügt werden. Hierfür müssen diese selektiert werden, indem entweder mit gehaltener Umschalt-Taste die Bilder nacheinander auf der Arbeitsfläche oder in der Liste links unten selektiert werden. Ein Klick auf „Finalize“ erzeugt ein einzelnes weiteres Bild, welches alle selektierten Bilder zu einem neuen Bild zusammenfügt. In Abbildung 11.8 wird das Ergebnis der verschiedenen Arbeitsschritte gezeigt. Der letzte Schritt



Abbildung 11.9: Überführung in eine rechteckige Form (Orthogonalprojektion)

besteht nun darin, die Orthogonalprojektion herzustellen, also die

Innenwand in ein Rechteck zu überführen und überflüssige Bildteile zu beschneiden. Ein Klick auf den Button „ToQuad“ startet das entsprechende Werkzeug. Es müssen nun nacheinander und im Uhrzeigersinn die vier Eckpunkte der Wand auf dem zusammengefügten Bild markiert werden. Das Werkzeug beendet sich nach dem vierten Klick automatisch. Die Berechnung wird mit einem Klick auf „FinalizeToQuad“ durchgeführt. Resultat ist ein bereits in ein Rechteck überführtes Bild.
Das Beschneiden findet über die vier Regler „left“, „right“, „top“ und „bottom“ statt. Diese müssen so eingestellt werden, dass die überflüssigen Bildteile ausgeblendet werden. Abbildung 11.9 zeigt den gesamten letzten Schritt.
Der gesamte Vorgang des Stitchings wird abgeschlossen, indem das finale Bild selektiert und der Button „Result“ angeklickt wird.

Anmerkung:
Es gibt noch drei weitere Buttons im Textureditor:

- *CleanUp* beseitigt die erzeugten Marker aus dem Stitching-Werkzeug,
- *Conform* schiebt zwei Bilder, welche Marker enthalten, so übereinander, dass der Marker links oben jeweils übereinstimmt. Dieses Werkzeug kann benötigt werden, wenn Detailaufnahmen innerhalb eines Stitchings eingefügt werden sollen.
- *Put* speichert das selektierte Bild als TIF auf der Festplatte ab. Dies kann z.B. als Zwischenergebnis dienen.

Das Resultat wird im zuvor ausgewählten Gruppenknoten (im Beispiel „wand2“) gespeichert.

Zuweisung der Texturen

Es kann nun einer Wand die erzeugte Textur zugewiesen werden. Hierfür muss entweder eine Gruppe oder ein Bild in der Shader-Übersicht selektiert werden. Anschließend kann eine Wand selektiert und die gewählte Textur über den Button „Set Texture“ zugewiesen werden. Sind mehrere Objekte in der Shader-Übersicht selektiert worden, öffnet sich ein Pop-Menü der selektierten Gruppen/Bilder, das eine Präzisierung ermöglicht. Das Ergebnis des Zuweisens zweier Texturen zeigt Abbildung 11.10.



Abbildung 11.10: Zuweisung der Texturen zu zwei Wänden

Import von Objekten

Nun kann die Szene mit verschiedenen Objekten ausgestattet werden. Hierfür kann über den Button „Import“ und anschließender Selektion von „Obj“ ein Datei-Browser geöffnet werden, um Obj-Dateien zu importieren. Wie diese Dateien entstanden sind, ob durch einen 3D-Scanner oder ein Modellierprogramm, spielt keine Rolle. Das Objekt wird in die Szene eingefügt, wie Abbildung 11.11. In die-

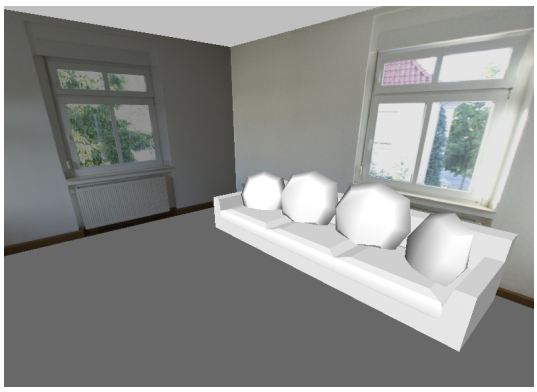


Abbildung 11.11: Import eines Objektes

sem Beispiel wurde das Objekt manuell modelliert.

Nun können weitere Objekte importiert und platziert werden. Abbildung 11.12 zeigt die vollständige Beispielszene. Es wurde dem Fußboden eine weitere Textur hinzugefügt, die in etwa dem realen Fußboden entspricht.

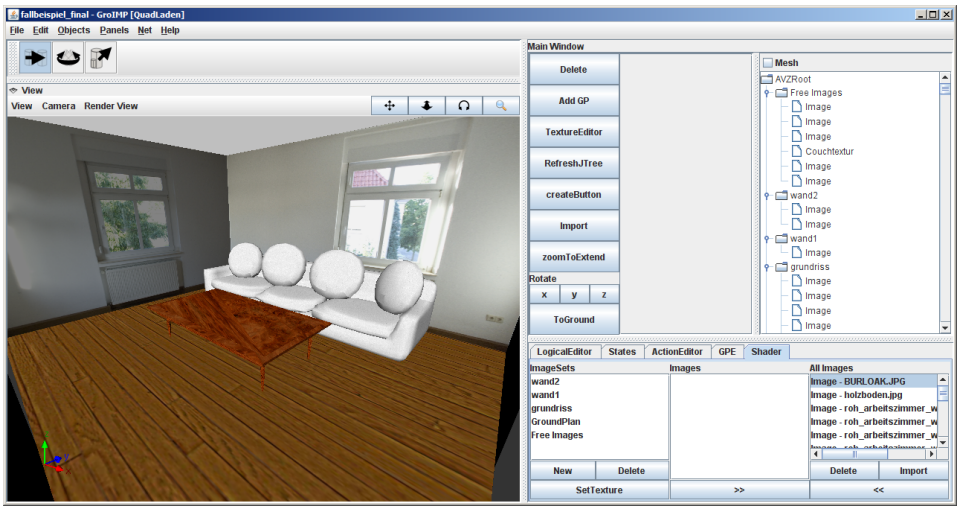


Abbildung 11.12: Fertige Beispielszene

Definition von Zuständen

Nun soll beispielhaft ein weiterer Zustand definiert werden, der sich vom initialen Zustand unterscheidet. Dies kann z.B. die Position der Objekte im Raum sein. Dies ermöglicht einen Vergleich.
Unter dem Reiter „States“ befindet sich der Zustandseditor. Ein Klick auf „Add State“ fügt einen neuen Zustand hinzu. Nun müssen die Objekte, die durch das Zustandskonzept veränderbar sein sollen, über eine Selektion und einen Klick auf „Add Object“ hinzu gefügt werden. Die Tabelle erweitert sich entsprechend, wie in Abbildung 11.13 gezeigt. Es gibt nun zwei Zustände: „Initial State“ und „Zu-

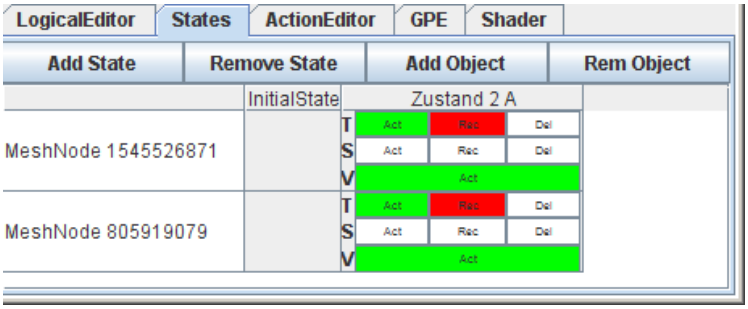


Abbildung 11.13: Zustandseditor mit zwei Zuständen („InitialState“(Urzustand) und „Zustand B“)

stand 2“ (durch einen Doppelklick auf den Tabellenkopf kann ein Zustand umbenannt werden). Das „A“ im Namen markiert den aktuellen Zustand.

Eine Zelle der Tabelle enthält nun sieben Buttons, welche jeweils „T“ (Transformation), „S“ (Shader) und „V“ (Visibility) zugeordnet sind. Der Button „Act“ aktiviert das Überschreiben der jeweiligen Eigenschaft. Ist dieser grün, ist das Überschreiben aktiv, d. h. es werden die Werte übernommen, die im jeweiligen Zustand gespeichert sind. Ist der Button deaktiviert (also nicht grün), werden die Werte des initialen Zustandes gewählt.
Der Button „Rec“ aktiviert den Aufzeichnungsmodus. Ist der jeweilige Zustand aktiv, werden alle Veränderungen entsprechend in diesem Zustand gespeichert. Wird also das Objekt bewegt mit aktiviertem Button „Rec“, wird diese Transformation entsprechend im Zustand gespeichert. Für die Eigenschaft Shader verläuft dies äquivalent.
Soll der in dem Zustand gespeicherte Wert gelöscht, also durch den initialen Wert überschrieben werden, muss der Button „Del“ betätigt werden.
Die Sichtbarkeit des Objektes wird über den Button „Act“ in der Zeile „V“ gesteuert. Ist dieser Button deaktiviert, wird das Objekt in diesem Zustand ausgeblendet. Abbildung 11.14 zeigt die Szene im

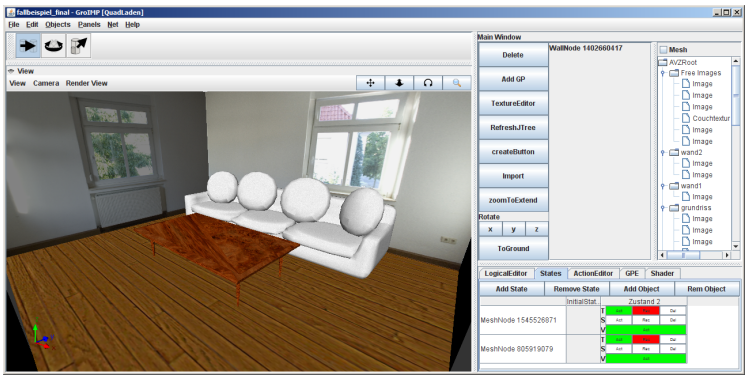


Abbildung 11.14: Szene im Urzustand mit einem zweiten definierten Zustand

Urzustand mit dem zweiten definierten Zustand. Innerhalb dieses Zustandes wurden beispielhaft die Positionen der beiden Objekte verändert, dies zeigt Abbildung 11.15, nachdem dieser Zustand aktiviert wurde.

Es ist auf diese Art und Weise möglich, die Texturen und Transformationen für ein Objekt durch verschiedene Zustände zu überschreiben und zwischen diesen Zuständen zu wechseln.

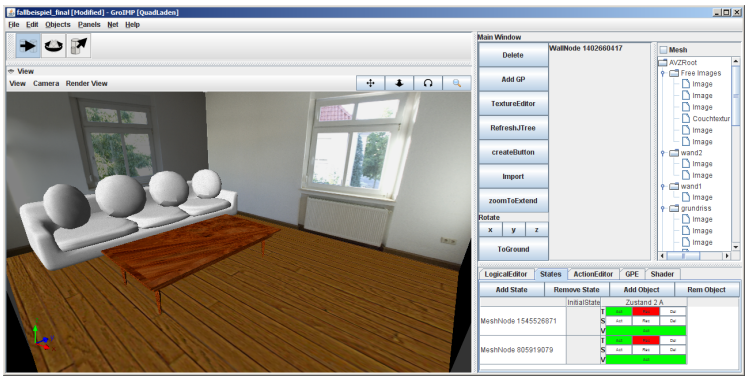


Abbildung 11.15: Szene nach Umschalten in den zweiten Zustand

Definition von Aktionen und Reaktionen

Als Abschluss der Präsentationserzeugung soll nun ein Aktions-Reaktions-Graph erschaffen werden, welcher nach einem Klick auf den Tisch einige Informationen anzeigt und den Wechsel des Zustandes ermöglicht. Unter dem Reiter „ActionEditor“ befindet sich der entsprechende Editor. Zunächst muss der Tisch selektiert werden, damit eine Aktion hinzugefügt werden kann. Dies geschieht mit einem Klick auf den Button „Create Action“. Nach einer Selektion des erzeugten Knotens und einem Klick auf „Add“ kann eine Aktion hinzu gefügt werden („Click“), ein Klick auf „Commit“ links unten bestätigt die Änderung. Nun können drei Reaktionsknoten erzeugt

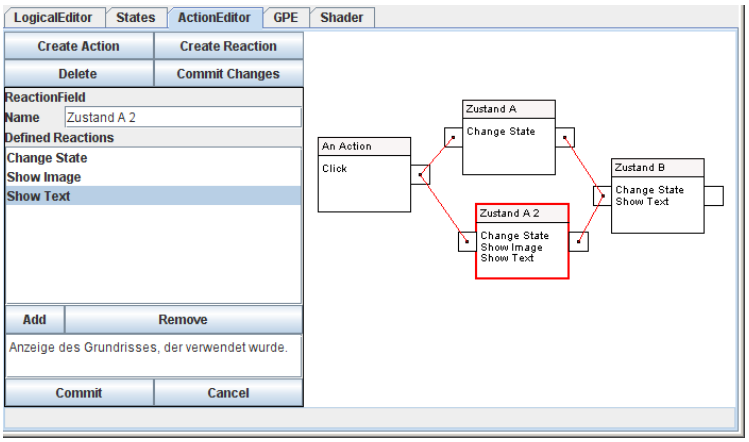


Abbildung 11.16: Definierte Aktion und verschiedene Reaktionen

werden über den Button „Create Reaction“ und entsprechend der Abbildung 11.16 platziert und verbunden werden. Eine Verbindung

wird geschaffen, indem durch „Klicken und Ziehen“ von dem kleinen Feld auf der rechten Seite eines Knotens zu einem kleinen Feld auf der linken Seite eines anderen Knotens eine Linie gezogen wird. Diese Linie besagt, dass ein Reaktionsknoten als Folge eines anderen Knotens dienen kann.
Nun können den Reaktionsknoten verschiedene Reaktionen über den Button „Add“ hinzugefügt werden. Ein Doppelklick auf die Reaktion auf der linken Seite öffnet einen passenden Editor. In Abbildung 11.16 wird ein Textfeld links unten angezeigt, um der Reaktion „Show Text“ einen entsprechenden Text zuzuordnen.
Eine wichtige Reaktion ist „Change State“. Das ermöglicht, bei Auswertung eines Reaktionsknotens den Zustand zu wechseln. In diesem Beispiel wechseln die Knoten „Zustand A“ und „Zustand A 2“ in den initialen Zustand, „Zustand B“ in den zweiten zuvor definierten Zustand. Die gesamten Änderungen werden durch den Button „Commit Changes“ in den Szenegraphen übertragen.
Dem Reaktionsknoten „Zustand A 2“ wurden insgesamt drei Reaktionen zugeordnet.

- „Change State“ aktiviert den initialen Zustand,
- „Show Image“ zeigt den importierten Grundriss und
- „Show Text“ zeigt einen simplen Text an.

Dem Reaktionsknoten „Zustand B“ wurden zwei Reaktionen zugeordnet.

- „Change State“ aktiviert den zweiten Zustand und
- „Show Text“ zeigt einen simplen Text an.

Durchführung der Präsentation

Um die Präsentation nun durchzuführen, muss in GroIMP in die Ansicht „ArchiViz Presentation Layout“ (Menü „Panels“, „Set Layout“) gewechselt werden. Es sollte zuvor sicher gestellt werden, dass sich das System im initialen Zustand befindet. Ein Klick auf den Tisch wertet den angefügten Aktions-Reaktions-Graph aus, es werden zunächst zwei Buttons, die in den Reaktionsknoten „Zustand A“ bzw. „Zustand A 2“ führen, angezeigt. Ein Klick auf „Zustand A 2“ wertet den Reaktionsknoten aus, Abbildung 11.18 zeigt das Ergebnis dieser Auswertung. Es wird entsprechend der Definition der

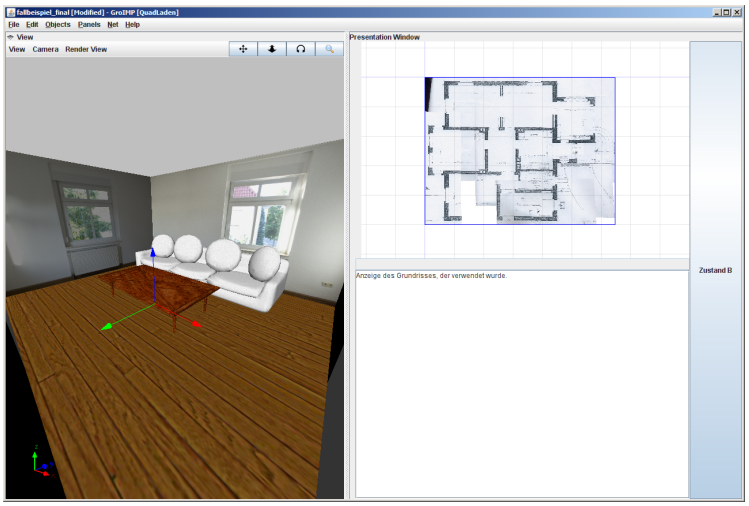


Abbildung 11.17: Auswertung des Knotens „Zustand A 2“ entsprechend der Definition

initiale Zustand aktiviert, der Grundriss und ein simpler Text angezeigt. Ein Klick auf den Button „Zustand B“ auf der rechten Seite

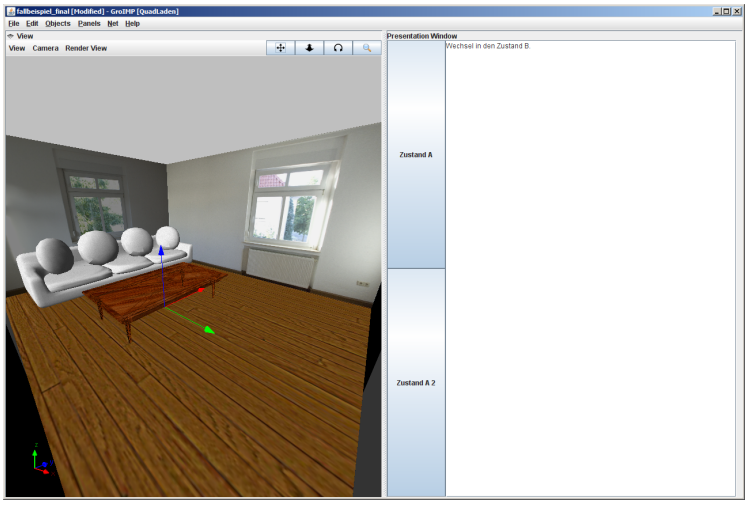


Abbildung 11.18: Auswertung des Knotens „Zustand B“ entsprechend der Definition

wertet den Reaktionsknoten „Zustand B“ aus, wobei in den zweiten Zustand gewechselt und ein simpler Text angezeigt wird.

11.1 Fazit

Dieses einfache Beispiel zeigt den gesamten Arbeitsgang der Plugins ArchiViz. Darüber hinaus wird ersichtlich, dass ausgehend von diesem Beispiel noch viel mehr Möglichkeiten gegeben sind. So könnten Stück für Stück die anderen Räume texturiert und möbliert, mehr Details hinzugefügt und die Präsentation entsprechend erweitert werden. Doch dies würde den Rahmen dieser Arbeit überschreiten. Die hier gezeigten Arbeitswege sollen einen Überblick über die gesamte verfügbare Funktionalität bieten.

12 Ausblick

12.1 Weitere Grundrisswerkzeuge

Nachfolgend seien einige Punkte erläutert, welche sich auf eine mögliche Funktionalitätserweiterung des Grundrisseditors beziehen:

- Bemaßung und Skalierung
  Hierbei ist das Prinzip „Measure by example“ günstig: Der Anwender zieht eine Linie mit bekannter Länge und Lage (horizontal oder vertikal). Anschließend wird der zugrunde liegende Grundriss automatisch gedreht und skaliert, sodass die Ausrichtung der Szenenachsen und die Skalierung der vorliegenden Szenenskalierung (z. B. 1 Einheit = 1 m) entspricht.

- Automatische Raumdefinition
  Wie bereits in Kapitel 7.1.2 vorgestellt, kann eine automatische Raumerkennung sehr behilflich sein.

- Ausrichtung und Fixierung von Objekten an einer Wand
  Es ist besonders für Wandbilder recht schwierig, diese an den entsprechenden Wänden korrekt zu platzieren. Ein zusätzliches Werkzeug, welches ein beliebiges Objekt an einer Wand ausrichtet, kann gegenüber dem manuellen Ausrichten eine große Zeitersparnis darstellen.

12.2 ArchiViz unter Verwendung von XL

An dieser Stelle soll kurz darauf eingegangen werden, wie und in welcher Form XL als ein sehr wichtiger Bestandteil von GroIMP (bzw. die umfangreiche Anbindung über das Plugin RGG) als Erweiterung und Unterstützung von ArchiViz dienen könnte.
Die Vorteil von XL liegt eindeutig im sehr präzisen Umgang mit Graphen: Über nur wenige definierte Regeln lassen sich komplexe

Veränderungen am Szenegraphen durchführen. Darüber hinaus ist die Möglichkeit, generisch Geometrie zu erzeugen, eine weitere Basis zum Aufbau auf XL.
Jedoch ist der Umgang mit diesem wachstumsbasierten Paradigma an mancher Stelle nicht ganz einfach. Die Komplexität der Regeln kann mit den Anforderungen recht schnell ansteigen. Insgesamt betrachtet wäre es ein relativ neues Einsatzgebiet für XL.
Nachfolgend sei ein kleines Beispiel beschrieben, in welcher Form XL zur Anwendung gebracht werden könnte. Zunächst bleibt die Erzeugung des Grundrisses äquivalent, dies sollte nach wie vor interaktiv in 2D stattfinden. Der Unterschied liegt aber darin, wie die Eingaben ausgewertet werden. Ein Klick eines Anwenders, der einen Knoten erzeugt, startet ein XL-Programm, ausgestattet mit den entsprechenden Regeln. Zeitgleich kann eine Verifikation stattfinden: Darf der Knoten überhaupt platziert werden? Ist der Zustand des Szenegraphen konsistent[1]? Für diese Fragen ist XL prädestiniert.
Zugleich könnte XL Zugang zum Kern von ArchiViz haben, indem entsprechende Methoden eingebunden werden. Somit könnte z. B. Zugriff auf die Bilderverwaltung gestattet werden.
Interessant wäre besonders das Konzept der Zustände von Knoten (vgl. Kap. 10.1) angewendet auf XL. Da in ArchiViz dieses Konzept als Subgraph umgesetzt ist, der über zusätzliche Kanten und Knoten Änderungen an Attributen festhält und nach einem entsprechenden Kommando an den Knoten übergibt, kann XL genau an dieser Stelle eingreifen. Entsprechende XL-Programme könnten die Verwaltung des Subgraphen vollständig übernehmen mit dem Vorteil, dass das Konzept auch außerhalb von ArchiViz greifen könnte.

12.3 Zustände von Knoten

Wie im Abschnitt 12.2 bereits erwähnt, könnte XL die Verwaltung von verschiedenen Zuständen übernehmen. Das bezieht sich aber darauf, dass das Konzept auf einen Subgraphen aufbaut. Dies ist allerdings nicht die einzige Lösung, mehrere Zustände zu definieren und zu verwalten. Nachfolgend sei kurz ein Konzept erläutert, wie solch eine Zustandsverwaltung noch aussehen könnte. Im Zuge der

[1] Da XL innerhalb eines Graphen bestimmte Muster suchen kann, könnte XL dafür genutzt werden, unerlaubte, also inkonsistente Zustände zu ermitteln und eventuell zu korrigieren. Das würde dazu führen, dass die Software insgesamt fehlerresistenter wird.

Entwicklung von ArchiViz kam dies nicht in Frage, da dies zu tiefe Eingriffe in die Funktionsweise von GroIMP zur Folge hätte.
Dreh- und Angelpunkt des Konzeptes sind die Knoten im Szenegraph, jeweils repräsentiert durch die Klasse `Node`. GroIMP hat selbst bereits die Möglichkeit, auf alle Attribute eines Knotens zuzugreifen (über eine Bearbeitung durch das *enhance*-Skript, vgl. Kap. 4.7.1). Ein Vorschlag wäre, diesen Zugriff vollständig zu zentralisieren. Kein Knoten könnte mehr seine Attribute selbst lesen und schreiben, sondern nur über den „Umweg“ einer zentralen Verwaltung. Abbildung 12.1 zeigt dieses Konzept stark vereinfacht. Hierbei

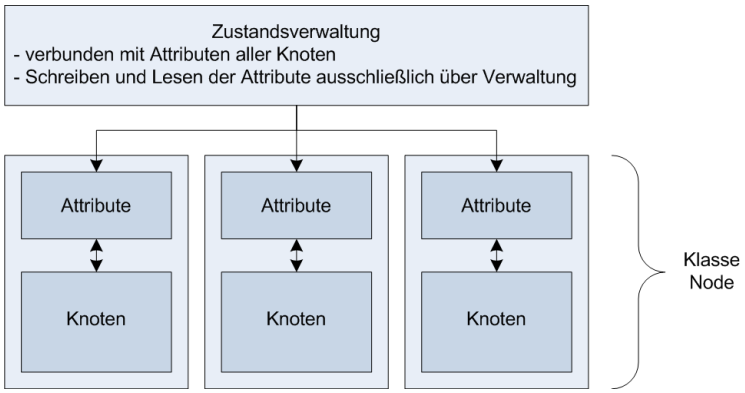


Abbildung 12.1: Zentralisierung der Attributverwaltung

sind alle Attribute „herausgelöst“, damit der Zugriff durch den Knoten selbst eingeschränkt werden kann. Die Zustandsverwaltung hat Zugriff auf diese Attribute über dieses separate Objekt.
Nun ist zu behaupten, dass dies ein enormer Eingriff in die bisherigen Konzepte von GroIMP wäre. Doch dem ist entgegen zu bringen: Ohne tiefe Eingriffe ist ein allgemeines Zustandskonzept kaum zu realisieren. Aber der Vorteil eines zentralen Systems wäre noch ein anderer: Neben dem Konzept verschiedener Zustände könnten auch Animationskonzepte oder andere Techniken umgesetzt werden, wobei diese sich zugleich auf alle neuen Knoten, die hinzugefügt werden, auswirken. Ebenso kann realisiert werden, dass verschiedene Zustände sich nur auf bestimmte Attribute auswirken.
Komplizierter wird es, wenn das Zustandskonzept nicht nur auf Attribute von Knoten wirken soll, sondern gleich auf ganze Graphen und deren Topologie: Hierbei müssten die Kanten selbst und Konzepte zur Sichtbarkeit von Knoten und Kanten mit einbezogen werden.

Eine weitere Anwendbarkeit des Konzeptes wäre die Modellierung dynamischer Prozesse über die Zeit. Ist das Prinzip verschiedener Zustände von Knoten und Kanten in Bezug auf deren Attribute und Sichtbarkeit realisiert, kann dieses auch ausgeweitet werden. Grundsätzlich gesehen ist jeder Zeitpunkt ein Zustand des Gesamtsystems (oder auch nur eines Teils, das kommt auf die Konzeption an). Die Definition bzw. Modellierung dieser dynamischen Prozesse kann hierbei durchaus kontinuierlichen Charakter haben: jedoch findet die Auswertung üblicherweise in diskreten Zeitabständen statt.
Als Anwendung in der Architektur könnte die Modellierung von Konstruktionsprozessen über die Zeit sein: ein Gebäude oder ein Innenraum würde nicht nur als fertiges Objekt darstellbar sein, sondern es könnten auch seine verschiedenen Konstruktionsstufen dargestellt und ineinander überführt werden.

12.4 Aktionen und Reaktionen

Das in Kapitel 9.1 beschriebene System zur Erzeugung von Aktionen und Reaktionen hat einen eher auf ArchiViz zugeschnittenen Charakter, jedoch ist das Grundprinzip durchaus nutz- und erweiterbar. Es wäre denkbar, als eine weitere Möglichkeit XL-Regelausführungen als eine mögliche Reaktion zu definieren. Denkt man dieses Konzept konsequent weiter, könnte auch die Interaktion mit XL an sich davon profitieren. Bisher wird jede Methode, die in einem XL-Programm als `public` deklariert ist, als ein Button in der RGG-Toolbar GroIMPs dargestellt. Dies ist allerdings ein recht lineares Vorgehen: Diese Methoden werden einmalig deklariert und die Buttons werden nach erfolgreicher Kompilation einmalig erzeugt. Nun wäre es aber möglicherweise sinnvoll, diese Auswahl von Methodenausführung zu Methodenausführung neu aufzubereiten. Dies könnte der Ausführung von XL-Code einen interaktiveren Charakter geben. Der Programmierer kann den späteren Anwender (oder vielleicht sich selbst für spätere Präsentationzwecke) gezielt in seinen Entscheidungen steuern. Nachfolgend sei ein kurzes Beispiel gezeigt.

```
1 module meristem extends Sphere;
2 float inputData = 0;
3
```

```
4  public void run1 ()
5  [
6    //eine Methode wie diese:
7    Axiom ==> [RU(15) meristem] [RU(45) meristem] [RU(70) meristem];
8    meristem ==> F meristem;
9  ]
10
11 public void run2 ()
12 [
13   Axiom ==> [RU(15) meristem] [RU(70) meristem];
14   meristem ==> RU(15) F F F meristem;
15 ]
16
17 boolean condition()
18 {
19    return !empty((* F F F meristem *));
20 }
21
22
23 protected void input ()
24 {
25   inputData = showNumericalInput();
26 }
27
28 public void run3 ()
29 [
30   meristem ==> [RU(inputData)] Sphere;
31 ]
```

Listing 12.1: Einfaches XL-Programm

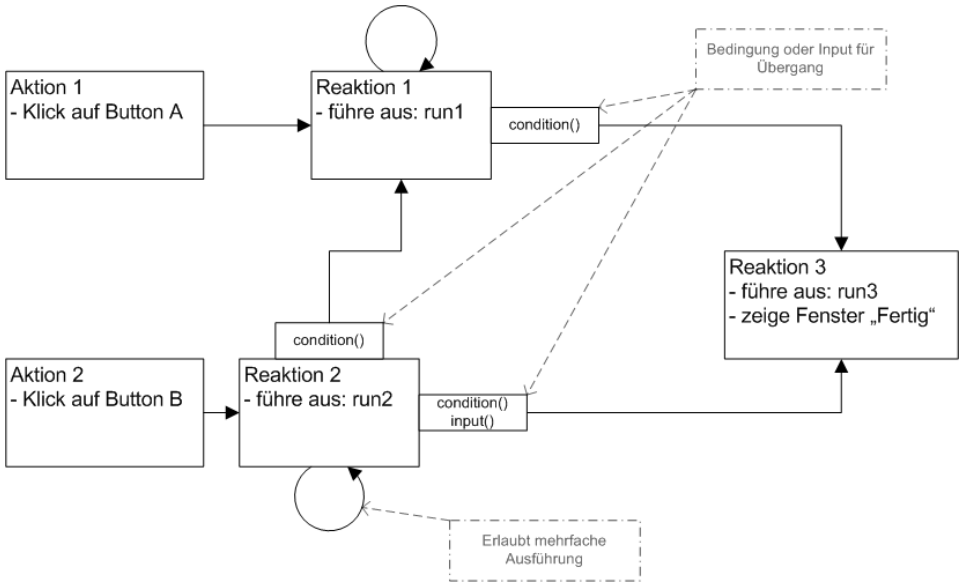


Abbildung 12.2: Aktionen und Reaktionen als Graph

Das Listing 12.1 sei das zum in Abbildung 12.2 abgebildeten Aktions-Reaktions-Graphen zugeordnete XL-Programm. Zu beachten ist, dass die Methode `showNumericalInput()` der Zeile 25 ein Eingabefenster für einen numerischen Wert anzeigt und den Input als Rückgabewert zurück gibt. Dies ist eine rein konzeptionelle Darstellung: eine solche Methode ist derzeit nicht verfügbar, könnte aber leicht hinzugefügt werden.
Äquivalent zu der Umsetzung für ArchiViz besitzt der Graph Aktions- bzw. Reaktionsknoten und Kanten. Als Auslöser sind zwei verschiedene Aktionen definiert: Ein Klick auf einen Button A bzw. B. Die in den Knoten Reaktion 1, Reaktion 2 und Reaktion 3 definierten Reaktionen führen jeweils die Methode run1, run2 und run3 des Listings 12.1 aus. Eine Erweiterung des Konzeptes bilden aber die an die Reaktionen gekoppelten Bedingungen: Diese können beispielsweise Inputs oder die Auswertung von Bedingungen sein und steuern, ob ein Übergang zur nächsten Reaktion möglich ist, oder sie führen anderen beliebigen Code aus. Sind mehrere Bedingungen gegeben, müssen alle ein `true` zurück geben, damit die nächste Reaktion ausgeführt wird. Ebenso sollte eine Reihenfolge bestimmbar sein.
An dieser Stelle sei das Sprachkonstrukt der Methode `condition()` kurz erläutert. Hierbei wird ein Pattern-Matching durchgeführt (Vgl. (Kniemeyer, 2008, S. 141 ff.)), wobei geprüft wird, ob es den übergegebenen Teilgraphen gibt, an dieser Stelle `F F F meristem`. Die Funktion `empty()` gibt `true` zurück, falls die ermittelte Liste von Vorkommen nicht leer ist, ansonsten `false`.
Das gegebene Beispiel würde nun folgenden Ablauf haben:
Mit einem Klick auf den Button B, welcher in der GroIMP-GUI angezeigt[1] wird, startet die Auswertung des Aktions-Reaktions-Graphen. Ein Vergleich der verschiedenen Aktionsknoten und der darin definierten Aktionen ermittelt den aktuellen Knoten: Dieser ist der Knoten „Aktion 2“. Entsprechend dem bisherigen Prinzip zeigt eine GUI nun für jede weiterführende Kante einen Button, den der Anwender anklicken kann.
Da der Knoten nur eine Nachfolgerkante hat, wird nur ein Button angezeigt. Der Übergang wird nach einem Klick auf diesen Button ausgeführt und der nächste Knoten, „Reaktion 2“, ausgewertet. Der Knoten enthält nur eine Reaktion: Die Ausführung der Methode `run2()` in Zeile 11. An dieser Stelle wird die erste Regel der Metho-

[1] Hierfür müsste eine Auswertung der verschiedenen Aktions-Knoten stattfinden und entsprechend eine Toolbar gefüllt und in GroIMP angezeigt werden.

de ausgeführt, nachdem beide Regeln gemäß dem Prinzip von XL ausgewertet worden sind. Diese Regel ersetzt das Axiom[1] durch zwei Transformationsknoten vom Typ `RU`[2]. Hier hat der Anwender drei Möglichkeiten:

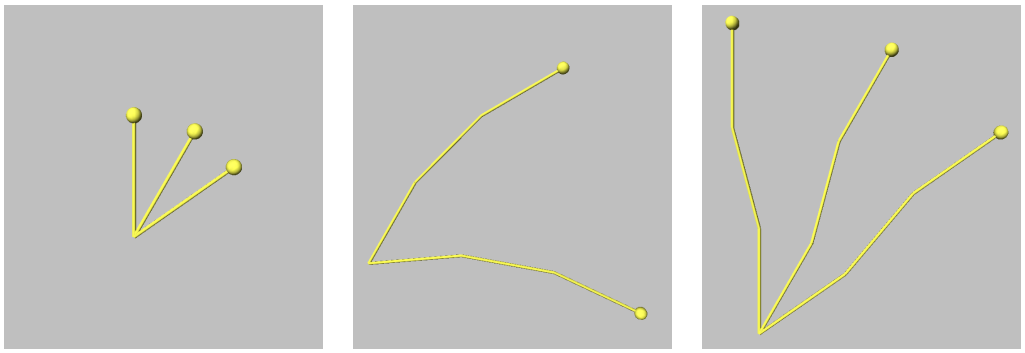
- Übergang in den Knoten „Reaktion 1“. Hierfür würde zunächst die Methode `condition()` ausgeführt und das Ergebnis ausgewertet werden. Ergibt die Methode den Wert `true`, ist der Übergang möglich.
- Übergang in den Knoten „Reaktion 3“. Neben der Ausführung der Methode `condition()` wird nun zusätzlich eine Eingabe eines numerischen Wertes erwartet. Ergibt die Methode `condition()` den Wert `true`, kann der Übergang stattfinden. Ebenso ist zu beachten, dass die Reihenfolge auch darüber entscheidet, in welcher Reihenfolge die Bedingungen ausgeführt werden. Würde die Methode `input()` noch vor der Methode `condition()` ausgeführt werden, hätte sich der Wert der Variable `inputData` nach der Benutzereingabe verändert. Solche Seiteneffekte müssten bei einer Präzisierung des Konzeptes mit einbezogen werden.
- Übergang zurück in den Knoten „Reaktion 2“. Hierbei würde die in diesem Knoten enthaltene Methode erneut ausgeführt und die enthaltenen Regeln ausgewertet werden.

Nun möge der Anwender sich für die zweite Option entscheiden. Dies schlägt allerdings fehl, weil die Methode `condition()` den Wert `false` zurück gibt. Somit bleibt „Reaktion 2“ der aktuelle Knoten, die darin definierten Reaktion werden nicht erneut ausgewertet. Nun bleibt dem Anwender nur die erste oder dritte Option: Er möge sich für die dritte Option entscheiden. Dies führt zu einer erneuten Auswertung der definierten Reaktionen: Es kommt die Regel in Zeile 14 zur Anwendung. Nun hat der Anwender erneut die Wahl, welche ebenso auf die dritte Option falle. Diesmal allerdings ergibt die Auswertung der Methode `condition()` den Wert `true` und die zweite Bedingung kann eintreten: Ein Fenster für die Eingabe eines numerischen Wertes öffnet sich. Nach Eingabe des Wertes 45 findet der

[1] Das Axiom ist ein Basisknoten, der zuvor automatisch erzeugt wurde und üblicherweise als Ausgangspunkt dient.

[2] Eine Einführung in XL und die L-System-Befehle ist zu finden in (Mannl, 2006, S. 7 ff.), (Bischof, 2006, S. 28 ff.) und Kniemeyer (2008).

Übergang statt und der nächste Knoten, „Reaktion 3“, kann ausgewertet werden. Dieser enthält zweierlei: Die Ausführung der Methode `run3()` und das Anzeigen eines Informationsfensters, das die Textzeile „Fertig“ anzeigt. Dieser Knoten stellt zugleich einen Abschluss dar, da von ihm keine Kante wegführt.
Abbildung 12.3a zeigt das Resultat des beschriebenen Ablaufes. Abbildung 12.3b und 12.3c zeigen noch zwei weitere mögliche Abläufe. Nachfolgend seien noch einige Ideen zu den verschiedenen Ele-



(a) Reaktionsknoten 2-2-3 (b) Reaktionsknoten 1-1-1-3 (c) Reaktionsknoten 2-2-2-1-3

Abbildung 12.3: Resultate drei verschiedener Pfade durch den Aktions-Reaktions-Graphen.

menten des Aktions-Reaktions-Graphen im Zusammenspiel mit XL erläutert. Viele davon sind selbst erklärend.

- Mögliche Aktionen
  - Selektion/Deselektion eines Objektes
  - Eingabeabfrage
    Als Input könnten Zeichenketten und numerische Werte bestimmter Formatierungen gelten.
  - Klick auf Button
    Diese Buttons könnten durch vorherige Auswertung der verschiedenen Aktionsknoten implizit bestimmt und angezeigt werden. Das bedeutet: Haben zwei Aktionsknoten A und B jeweils die Aktion „Klick auf Button *Name*“ definiert, werden zwei Buttons mit den jeweiligen Einträgen für *Name* automatisch erzeugt. Ein Klick darauf würde die Aktion auslösen. Heißen beide Einträge gleich, werden auch beide Aktionen zugleich ausgeführt.

- Mögliche Reaktionen:
  - Zeige Information (Text/Bild)
    Als Erweiterung dessen könnte HTML und CSS dargestellt werden.
  - Führe XL-Methode (mit enthaltenen Regeln) aus
  - Markiere ein Objekt (z. B. durch Einfärbung)
  - Bewege Kamera
- Mögliche Bedingungen:
  - Führe XL-Methode aus
  - Werte Bedingung aus (z. B. über Methodenaufruf mit Rückgabewert)
  - Zeige ein Fenster mit
    * numerischer Eingabe (deren Ergebnis z. B. eine globale Variable füllt)
    * Sicherheitsabfrage („Sicher?“ als Hinweis und „Ja“ bzw. „Nein“ als Buttons)
    * Warnung (mit Quittierung über „Okay“-Button und Eintritt in die nächste Reaktion)
    * Fehler (mit Quittierung über „Okay“-Button und verbleib in derselben Reaktion, ohne diese erneut auszuführen)
    * Prüfe Position der Kamera
- Struktur des Graphen:
  - Zyklen
  - Maschen
  - Unidirektionale Kanten
  - Bidirektionale Kanten
  - Kanten zum gleichen Knoten.
    Darüber hinaus können, äquivalent zum vorhandenen Konzept, diese Übergänge automatisch wiederholt ausgeführt werden, solange die entsprechenden Bedingungen erfüllt bleiben.

Abschließend seien noch einige neue Möglichkeiten, die sich damit offenbaren würden, aufgelistet:

- Auslagerung komplexer Entscheidungsstrukturen aus XL in einen relativ einfach zu erzeugenden „Logikgraphen".
  Das gezeigte Prinzip würde einige Teilbereiche von XL simplifizieren: Insbesondere das Einbeziehen von Interaktionen mit dem Anwender könnte auf diese Weise stark vereinfacht werden. Ebenso könnte in bestimmten Fällen die Ausführungsreihenfolge von Regeln klarer strukturiert und auf bestimmte Reihenfolgen, ohne komplizierte Abfragen in XL programmieren zu müssen, eingeschränkt werden.

- Präsentation von Modellen
  An dieser Stelle würde sich eventuell ein Modus GroIMPs äquivalent zu dem Präsentationsmodus in ArchiViz eignen. Die GUI könnte auf wichtige Elemente reduziert und die Interaktion mit dem Betrachter eingeschränkt werden.

- Zeigen und Kommentieren der schrittweisen Entwicklung des Modells
  Es kann präzise und dennoch mit einfachen Mitteln eine Abfolge von Schritten modelliert und zugleich dokumentiert werden.

- Entwicklung von Tutorials für XL
  Dies ist eine nicht zu unterschätzende Option: Da XL ein relativ komplexes Sprachgebilde ist, könnten mit dieser Technik geführte Tutorials entwickelt werden, die Ursache und Wirkung bestimmter Regeln demonstrieren. Durch die Nutzung bidirektionaler Kanten (und natürlich korrekter Implementierung der Übergänge) könnte der Anwender Schritte vorwärts und rückwärts ausführen. Darüber hinaus könnten zusätzliche Informationen wie Text und Bilder eingebracht werden. Es würde sich, wie im Punkt „Präsentation von Modellen" bereits erwähnt, ein Präsentationsmodus in GroIMP lohnen.

12.5 Dateninput durch Scanning

Durch die Nutzung eines 3D-Scanners wäre es möglich, komplexe 3D-Objekte ArchiViz zuzuführen, ohne diese von Hand modellieren

zu müssen. Es gibt bereits weit fortgeschrittene Techniken. Eine davon ist die Nutzung der Photogrammetrie, um dreidimensionale Objekte samt Textur aus einer Serie von Fotos zurückzuführen. Hierbei wird ein durch einen Motor gesteuerten Drehteller genutzt, um das Objekt um eine Achse in diskreten Schritten zu rotieren. Bei jedem Schritt wird durch eine Digitalkamera von einem festen Standort aus ein Foto des Objektes gemacht. Dem gesamten Vorgang geht die Kalibrierung der Kamera voraus, damit Brennweite, Linsenparameter, Standort und Blickrichtung eindeutig identifizierbar sind. Zudem muss das Objekt gut ausgeleuchtet und durch einen blauen Hintergrund separiert werden. Der Drehteller ist ebenso mit einer blauen Matte ausgestattet. Dies ist notwendig, damit das Objekt durch eine Software vom Hintergrund getrennt werden kann[1]. Das Resultat ist ein komplettes, aus Polygonen bestehendes 3D-Objekt, welches vollständig texturiert ist. Dies ist nur eine beispielhaft gewählte Technik, es gibt noch viele weitere vergleichbare (siehe (Luhmann, 2003, S. 477 ff.)).
Der Vorteil solch einer Technik besteht darin, dass das Objekt nahezu naturgetreu als 3D-Objekt dargestellt werden kann. Ist solch eine Messvorrichtung darüber hinaus transportabel, können z. B. alte Gegenstände, welche zu einem früheren Zeitpunkt zur Einrichtung gehörten, digitalisiert und in die Szene importiert werden, ohne dass der Gegenstand selbst transportiert werden muss.

[1] Dieser Vorgang wird bezeichnet als *Keying*.

13 Diskussion

Dieses Kapitel setzt sich in erster Linie mit der Frage auseinander, ob das angestrebte Ziel erreicht werden konnte. Dieses ist die Vereinigung aller vorgestellten Konzepte in einem Softwareprodukt. Die entscheidende Frage ist, ob der erreichte Stand die Grundlage für weitere Konzepte und Arbeiten bietet.
Grundsätzlich ist zu behaupten, dass die vorgestellten Konzepte insgesamt die Anforderung, ein flexibles, einfach zu benutzendes System zu bilden, erreicht haben. Dies zeigt die durchgeführte Implementierung und die Ausführungen dieser Arbeit, welche sich z. T. auf die Implementierung stützen, aber auch die Grundlage für Teile der Implementierung bilden. Jedes Konzept unterliegt in dieser Arbeit einer grundlegenden Betrachtung, eine Erweiterung des Konzeptes, wie im Kapitel 12 angedeutet, ist bei jedem dieser Konzepte möglich. Die Implementierung orientiert sich an diesen Konzepten und half, diese weiter zu entwickeln und in einigen Fällen überhaupt auf Umsetzbarkeit zu prüfen.
Besondere Anforderungen stellte dabei der Textureditor: Dessen Konzepte sind recht komplex und nicht ohne Weiteres über eine Implementierung umzusetzen. Das ist auch der Grund, warum in dieses Konzept die meiste Zeit investiert wurde.
Etwas anders war es z. B. beim Aktions-Reaktions-Editor bzw. dem gesamten Konzept dahinter. Hier half die Implementierung, das Konzept zu verifizieren. Hilfreich war auch das zuvor entwickelte Modul „2D-Canvas“, schnell zu Ergebnissen zu kommen und den Workflow zu planen.
Alle entwickelten bzw. vorgestellten Konzepte haben das Potenzial, auch in anderen Kontexten Einsatz zu finden. Insbesondere aber der Aktions-Reaktions-Graph bietet, wie bereits im Abschnitt 12.4 gezeigt, in Zusammenspiel mit XL viele neue Möglichkeiten der Definition von Interaktionen zwischen Anwender und Software.
Es ist zu bemerken, dass die Beispielimplementierung an einigen Stellen noch Lücken aufweist. Das hat grundsätzlich die verfügbare Zeit als Ursache. Insgesamt aber konnten alle Funktionalitäten zu-

mindest beispielhaft implementiert werden. Durch den unerwartet hohen Aufwand in der Implementierung stand zu wenig Zeit für das Testen der Software zur Verfügung, mit der Folge, dass noch nicht alle Aspekte und Vorgänge im Detail geprüft wurden.
Abweichend von der Aufgabenstellung musste auf ein Aufbauen auf das 3D-Format X3D verzichtet werden, die Gründe wurden in Kapitel 6.6 erläutert.
Es wurde aber gezeigt, dass alle Konzepte in einem Softwarepaket vereinbar sind. Die Arbeitsvorgänge sind durchgängig durchführbar und liefern nachvollziehbare Ergebnisse.
Etwas schwieriger ist der Vergleich von ArchiViz mit vorhandenen Systemen. Der Grund ist das breit angelegte Funktionalitätsspektrum. Wie bereits im Kapitel 1 erwähnt, gibt es für die verschiedenen Aspekte von ArchiViz durchaus unterschiedliche Lösungen, die einen ähnlichen Funktionsumfang anstreben. Jedoch ist die ganzheitliche Arbeitsweise von ArchiViz in dieser Form nicht anzutreffen. Die Vereinigung der verschiedenen Aspekte in einer Software, die den beschriebenen Arbeitsgang als einen einzigen großen ermöglichen, ist der Anspruch des Neuartigen an ArchiViz. Es sind im Zuge von ArchiViz verschiedene Konzepte entwickelt und angewendet worden, welche in ihrer Gesamtheit komplexe und vielfältige Möglichkeiten offenbaren.

A Mathematische Grundlagen

A.1 Eigenschaften von Relationen

Gegeben sei eine Menge G und eine Relation R über G mit $R \subseteq G \times G$.

- Reflexivität

$$R \text{ ist reflexiv gdw.} \quad \forall g \in G : (g, g) \in R \tag{A.1}$$

- Irreflexivität

$$R \text{ ist irreflexiv gdw.} \quad \forall g \in G : (g, g) \notin R \tag{A.2}$$

- Symmetrie

$$R \text{ ist symmetrisch gdw.} \quad \forall a, b \in G : (a, b) \in R \Rightarrow (b, a) \in R \tag{A.3}$$

- Antisymmetrie

$$R \text{ ist antisymmetrisch gdw.} \quad \forall a, b \in G : (a, b) \in R \wedge (b, a) \in R \Rightarrow a = b \tag{A.4}$$

- Asymmetrie

$$R \text{ ist asymmetrisch gdw.} \quad \forall a, b \in G : (a, b) \in R \Rightarrow (b, a) \notin R \tag{A.5}$$

- Transitivät

$$R \text{ ist transitiv gdw.} \quad \forall a, b, c \in G : (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R \tag{A.6}$$

A.2 Grundlagen zu Abbildungen

A.2.1 Abbildungsfunktion

Eine Abbildung sei definiert als eine Funktion $t : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ mit

$$t(p) = p' \quad \text{mit } p, p' \in \mathbb{R}^2 \tag{A.7}$$

Anders aufgefasst ist dies eine Abbildung eines ebenen kartesischen Koordinatensystems auf ein anderes. Die Funktion t kann eine der in Abschnitt A.3.1 dargestellten Abbildungen realisieren.

A.2.2 Homogene Koordinaten

Eine Darstellung einer affinen Abbildung ist:

$$p^{'} = Ap^T + t \tag{A.8}$$

wobei A eine 2×2-Matrix, $p = (x, y)$ der zu transformierende Punkt und t ein Translationsvektor ist. Um aus dieser Darstellung (eine Multiplikation und eine Addition) eine Multiplikation der Form

$$q^{'} = Mq \tag{A.9}$$

zu machen, erweitert man die Koordinate q um eine *homogene Komponente* w, sodass

$$q = \begin{pmatrix} x \\ y \\ w \end{pmatrix} \tag{A.10}$$

Aus einem eindeutig bestimmten Punkt wird somit ein Vektor, dessen Länge die Position (und somit wieder einen Punkt) im Koordinatensystem bestimmt. Bei der Umwandlung eines Punktes p in homogene Koordinaten wird die homogene Komponente auf 1 gesetzt, bei der umgekehrten Operation müssen die Komponenten x und y zunächst durch w dividiert werden.
Dieses Konzept wird auf die gegebene Transformationsmatrix A angewendet, wobei als zusätzlicher Spaltenvektor die Translationskomponente t in homogenen Koordinaten verwendet wird. Die fehlenden zwei Matrixelemente werden jeweils durch 0 ersetzt.
Das gezeigte Prinzip gilt für $\mathbb{R}^3$ äquivalent.

A.2.3 Transformationsfunktion

Eine Transformationsfunktion $T : \mathbb{R}^n \to \mathbb{R}^n$ mit $n \in \mathbb{N}$ ist definiert als:

$$T(p) = M \cdot p = p' \tag{A.11}$$

wobei M eine $(n+1) \times (n+1)$-Transformationsmatrix in homogenen Koordinaten ist und p, p' Punkte im $\mathbb{R}^n$ sind.

A.2.4 Transformationsverkettung

Eine Verkettung von Transformationen T_c entspricht der Verkettung von Transformationsfunktionen $T_0, T_1, \ldots, T_n$ mit $n \in \mathbb{N}$ (vgl. Kap. A.2.3), also deren Hintereinanderausführung. Somit ist:

$$T_c = T_0 \circ T_1 \circ \ldots \circ T_n \tag{A.12}$$

$$T_c(p) = T_n(T_{\ldots}(T_1(T_0(p)))) \tag{A.13}$$

In Matrixschreibweise nach Definition A.11 bedeutet dies:

$$p' = M_n \cdot \ldots \cdot M_1 \cdot M_0 \cdot p \tag{A.14}$$

wobei $M_0, M_1, \ldots, M_n$ die jeweiligen Transformationsmatrizen der Funktionen $T_0, T_1, \ldots, T_n$ sind.

A.3 Abbildungen im $\mathbb{R}^2$

A.3.1 Abbildungshierarchie

Die verschiedenen Arten der Transformationen im $\mathbb{R}^2$ ergeben eine Abbildungshierarchie. Abbildung A.1 auf Seite 182 zeigt alle Transformationen im Zusammenhang.

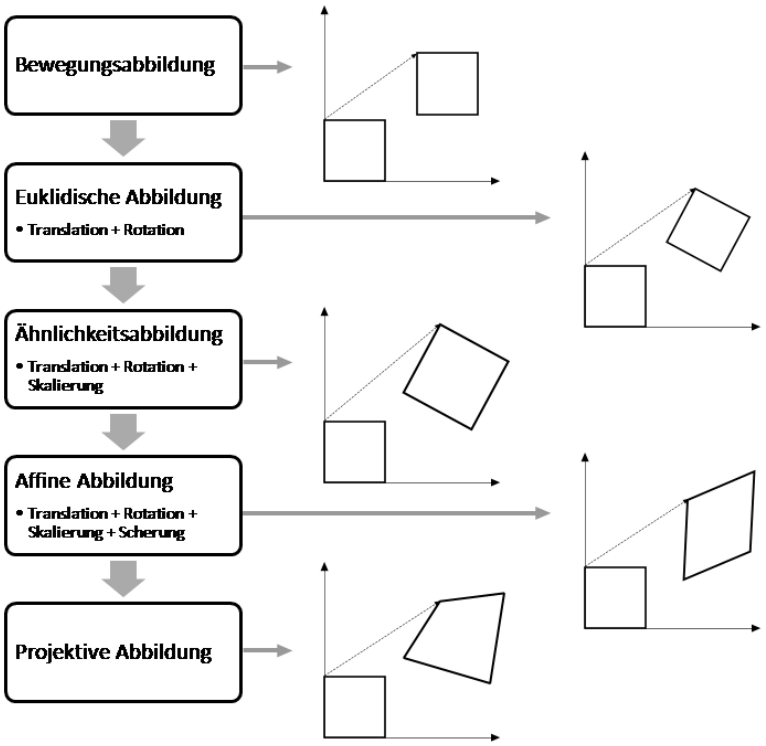


Abbildung A.1: Abbildungshierarchie von Transformationen im $\mathbb{R}^2$

- Translation (Bewegungsabbildung)

$$p' = M_{translation} \cdot p$$

$$= \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \tag{A.15}$$

  wobei t_x und t_y die Translationskomponenten sind.

- Translation und Rotation (euklidische Abbildung[1]; Längen bleiben erhalten, Winkel bleiben erhalten)

[1] Auch bezeichnet als *Rigid-Body-Transformation*, vgl. (Möller und Haines, 1999, S. 33)

$$
\begin{aligned}
p' &= M_{rotation} \cdot M_{translation} \cdot p \\
&= \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot M_{translation} \cdot p \\
&= \begin{pmatrix} \cos\alpha & \sin\alpha & t_x \\ -\sin\alpha & \cos\alpha & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot p
\end{aligned} \tag{A.16}
$$

wobei α der Rotationswinkel ist.

- Translation, Rotation und Skalierung (wenn $s_x = s_y$: Ähnlichkeitsabbildung, dann bleiben Winkel erhalten)

$$
\begin{aligned}
p' &= M_{scale} \cdot M_{rotation} \cdot M_{translation} \cdot p \\
&= \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot M_{rotation} \cdot M_{translation} \cdot p \\
&= \begin{pmatrix} \cos\alpha \cdot s_x & \sin\alpha \cdot s_y & t_x \\ -\sin\alpha \cdot s_x & \cos\alpha \cdot s_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot p
\end{aligned} \tag{A.17}
$$

wobei s_x und s_y die Skalierungsfaktoren in Richtung x und y sind.

- Translation und Rotation und Skalierung und Scherung (affine Abbildung, Dreipunktabbildung; parallele Linien bleiben paral-

lel)

$$p' = M_{shear} \cdot M_{scale} \cdot M_{rotation} \cdot M_{translation} \cdot p$$

$$= \begin{pmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot M_{scale} \cdot M_{rotation} \cdot M_{translation} \cdot p \tag{A.18}$$

wobei sh_x und sh_y die Werte für die Scherung in Richtung x und y sind.

Der letzte Punkt, die „Projektive Abbildung“, sei im Abschnitt A.3.2 genauer erklärt.

A.3.2 Projektive Abbildung

Mittels der projektiven Abbildung ist es möglich, vier beliebige Punktepaare ineinander zu überführen. Sie hat die Form:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{A.19}$$

Die Verhältnisse der Abstände zwischen mehreren Punkten bleiben im Allgemeinen nicht erhalten.
Gegeben sind nun vier Punktpaare im $\mathbb{R}^2$:

$$P = \{(p_0, p_0'), (p_1, p_1'), (p_2, p_2'), (p_3, p_3')\} \tag{A.20}$$

wobei $p_i = \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix}$ und $p_i' = \begin{pmatrix} x_i' \\ y_i' \\ 1 \end{pmatrix}$.
Nach dem Einsetzen in Gleichung A.19 ergibt sich folgendes Glei-

chungssystem (nach (Burger und Burge, 2006, S. 369)):

$$\begin{pmatrix} x'_0 \\ y'_0 \\ x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -x_0\,x'_0 & -y_0\,x'_0 \\ 0 & 0 & 0 & x_0 & y_0 & 0 & -x_0\,y'_0 & -y_0\,y'_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1\,x'_1 & -y_1\,x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 0 & -x_1\,y'_1 & -y_1\,y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2\,x'_2 & -y_2\,x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 0 & -x_2\,y'_2 & -y_2\,y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3\,x'_3 & -y_3\,x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 0 & -x_3\,y'_3 & -y_3\,y'_3 \end{pmatrix} \cdot \begin{pmatrix} a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \end{pmatrix} \tag{A.21}$$

Anschließend kann (a_{ij}) durch Lösung des linearen Gleichungssystems bestimmt werden.
Alternativ zur Lösung des linearen Gleichungssystems ist es möglich, den Weg einer zweistufigen Abbildung über das Einheitsquadrat zu gehen. Die folgenden Ausführungen entsprechen im wesentlichen (Burger und Burge, 2006, S.370 ff.).
Damit ergibt sich folgende Vereinfachung:

$$p_0 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad p_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad p_2 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad p_3 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \tag{A.22}$$

Nach dem Einsetzen ergibt sich folgendes vereinfachtes Gleichungssystem:

$$\begin{aligned} x'_0 &= a_{02} \\ y'_0 &= a_{12} \\ x'_1 &= a_{00} + a_{02} - a_{20} \cdot x'_1 \\ y'_1 &= a_{10} + a_{12} - a_{20} \cdot y'_1 \\ x'_2 &= a_{00} + a_{01} + a_{02} - a_{20} \cdot x'_2 - a_{21} \cdot x'_2 \\ y'_2 &= a_{10} + a_{11} + a_{12} - a_{20} \cdot y'_2 - a_{21} \cdot y'_2 \\ x'_3 &= a_{01} + a_{02} - a_{21} \cdot x'_3 \\ y'_3 &= a_{11} + a_{12} - a_{21} \cdot y'_3 \end{aligned} \tag{A.23}$$

$$\begin{array}{lll} a_{00} = x'_1 - x'_0 + a_{20} \cdot x'_1 & a_{01} = x'_3 - x'_0 + a_{21} \cdot x'_3 & a_{02} = x'_0 \\ a_{10} = y'_1 - y'_0 + a_{20} \cdot y'_1 & a_{11} = y'_3 - y'_0 + a_{21} \cdot y'_3 & a_{12} = y'_0 \end{array}$$

$$a_{20} = \frac{(x'_0 - x'_1 + x'_2 - x'_3) \cdot (y'_3 - y'_2) - (y'_0 - y'_1 + y'_2 - y'_3) \cdot (x'_3 - x'_2)}{(x'_1 - x'_2) \cdot (y'_3 - y'_2) - (x'_3 - x'_2) \cdot (y'_1 - y'_2)} \quad \text{(A.24)}$$

$$a_{21} = \frac{(y'_0 - y'_1 + y'_2 - y'_3) \cdot (x'_1 - x'_2) - (x'_0 - x'_1 + x'_2 - x'_3) \cdot (y'_1 - y'_2)}{(x'_1 - x'_2) \cdot (y'_3 - y'_2) - (x'_3 - x'_2) \cdot (y'_1 - y'_2)}$$

mit

$$A = \begin{pmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & 1 \end{pmatrix} \quad \text{(A.25)}$$

Hierbei ist die Transformation des Einheitsquadrates in p':

$$p' = A \cdot p \quad \text{(A.26)}$$

und die Umkehrung, also die Transformation von p in das Einheitsquadrat:

$$p' = A^{-1} \cdot p \quad \text{(A.27)}$$

Die Gesamttransformation wird nun durch Verkettung erreicht:

$$p' = A_2 \cdot A_1^{-1} \cdot p \quad \text{(A.28)}$$

wobei A_1 die Transformation vom Einheitsquadrat in das Ursprungsviereck und A_2 die Transformation vom Einheitsquadrat in das Zielviereck darstellt. Abbildung A.2 zeigt diese zweistufige Abbildung, wobei T die Transformation der Form $p' = T(p)$ ist und dabei $T(p) = A \cdot p$.
Für die Inversion von A siehe Abschnitt A.3.3.
Insgesamt ist also die projektive Transformation zu verstehen als:

$$p' = M_{proj} \cdot p \quad \text{(A.29)}$$

wobei

$$M_{proj} = A_2 \cdot A_1^{-1} \quad \text{(A.30)}$$

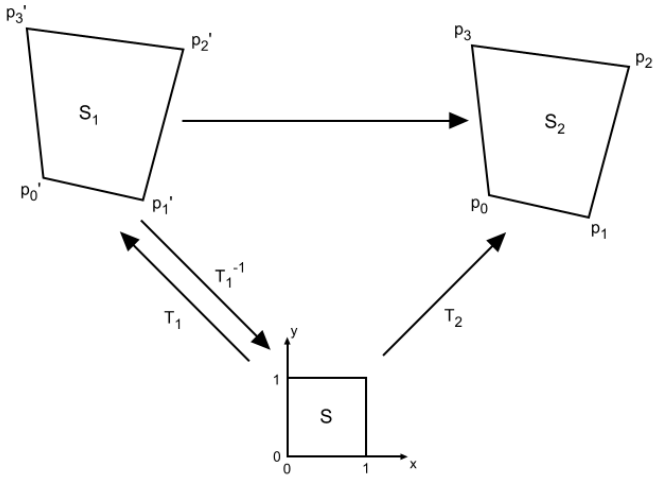


Abbildung A.2: Abbildung über das Einheitsquadrat (nach (Burger und Burge, 2006, S. 372, Abb. 16.5))

A.3.3 Inversion der projektiven Abbildung

Hierbei sei M die projektive Abbildung. Somit ist:

$$M^{-1} = \frac{1}{Det(M)} M_{adj} \tag{A.31}$$

wobei:

$$M = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \tag{A.32}$$

Eine Berechnungsvorschrift für die Determinante lautet:

$$\begin{aligned} Det(M) = \quad & a_{00}\, a_{11}\, a_{22} + a_{01}\, a_{12}\, a_{20} + a_{02}\, a_{10}\, a_{21} \\ & - a_{00}\, a_{12}\, a_{21} - a_{01}\, a_{10}\, a_{22} - a_{02}\, a_{11}\, a_{20} \end{aligned} \tag{A.33}$$

Die adjungierte Matrix M_{adj} ist zu berechnen durch:

$$M_{adj} = \begin{pmatrix} a_{11}\,a_{22} - a_{12}\,a_{21} & a_{02}\,a_{21} - a_{01}\,a_{22} & a_{01}\,a_{12} - a_{02}\,a_{11} \\ a_{12}\,a_{20} - a_{10}\,a_{22} & a_{00}\,a_{22} - a_{02}\,a_{20} & a_{02}\,a_{10} - a_{00}\,a_{12} \\ a_{10}\,a_{21} - a_{11}\,a_{20} & a_{01}\,a_{20} - a_{00}\,a_{21} & a_{00}\,a_{11} - a_{01}\,a_{10} \end{pmatrix} \tag{A.34}$$

A.4 Abbildungen im $\mathbb{R}^3$

A.4.1 Affine Transformation

Eine Transformationsmatrix M im $\mathbb{R}^3$ in homogenen Koordinaten hat folgende Form:

$$M = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{A.35}$$

Im $\mathbb{R}^3$ verhält sich die in Kap. A.3.1 beschriebene Abbildungshierarchie analog. Der einzige Unterschied besteht darin, dass die Rotation aufgeteilt wird in drei verschiedene Matrizen:

$$Rot_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.36}$$

$$Rot_y = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.37}$$

$$Rot_z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.38}$$

sodass

$$Rot_{xyz} = Rot_x \cdot Rot_y \cdot Rot_z \tag{A.39}$$

ist. Analog zu Formel A.17 ist die Ähnlichkeitstransformation eines Punktes p im $\mathbb{R}^3$:

$$p' = M_{scale} \cdot M_{Rot_{xyz}} \cdot M_{translation} \cdot p \tag{A.40}$$

B Listings

In diesem Anhang werden verschiedene Quellcodeausschnitte zur Verfügung gestellt.

```
<project
 name="ArchiViz"
 default="compile">
 <description>
  Ant build file for the ArchiViz plugin of GroIMP
 </description>
 <import file="../Build/buildproject.xml" />
 <target name="-projects">
  <antcall target="-addproject">
   <param
    name="proj"
    value="IMP-3D" />
  </antcall>
    <antcall target="-addproject">
   <param
    name="proj"
    value="Platform-Swing" />
  </antcall>
    <antcall target="-addproject">
   <param
    name="proj"
    value="DXF" />
  </antcall>
 </target>
 <target name="src">
  <antcall target="-src-enhance" />
 </target>
 <target name="-addjavadoc" />
</project>
```

Listing B.1: build.xml

```
<?xml version = "1.0" encoding="UTF-8"?>
<plugin
 id="de.grogra.ext.archiviz"
 version="0.0.1"
 xmlns="http://grogra.de/registry">
 <library
  file="jai_codec.jar"
  prefixes="{com.sun.media.jai}" />
 <library
  file="jai_core.jar"
  prefixes="{com.sun.media.jai,javax.media.jai}" />
 <library
  file="ArchiViz.jar"
  prefixes="{de.grogra.ext.archiviz}" />
 <import plugin="de.grogra.imp3d" />
 <import plugin="de.grogra.rgg" />
 <import plugin="de.grogra.pf.ui.swing" />
 <import plugin="de.grogra.ext.dxf" />
 <registry>
  <ref name="ui">
   <ref name="panels">
    <directory name="archiviz">
     <panel name="main">
      <exists
       name=".available"
       ref="/project" />
      <object expr="de.grogra.ext.archiviz.MainWindow">
       <var name="context" />
       <vars />
      </object>
     </panel>
     <panel name="presentation">
      <exists
       name=".available"
       ref="/project" />
      <object
       expr="de.grogra.ext.archiviz.PresentationPanel">
       <var name="context" />
       <vars />
      </object>
     </panel>
    </directory>
   </ref>
   <ref name="commands">
    <command
     name="newarchiviz"
     class="de.grogra.ext.archiviz.NewArchiViz" />
   </ref>
   <ref name="layouts">
    <layout name="archiviz">
     <exists
      name=".available"
      ref="/project" />
     <main>
      <split orientation="horizontal">
       <split orientation="vertical">
        <link source="/ui/panels/archiviz/toolbar" />
        <link source="/ui/panels/3d/defaultview" />
       </split>
       <link source="/ui/panels/archiviz/main" />
```

```
61        </split>
62       </main>
63      </layout>
64      <layout name="archivizpresentation">
65       <exists
66        name=".available"
67        ref="/project" />
68       <main>
69        <split orientation="horizontal">
70         <link source="/ui/panels/3d/defaultview" />
71         <link source="/ui/panels/archiviz/presentation" />
72        </split>
73       </main>
74      </layout>
75     </ref>
76    </ref>
77    <ref name="hooks">
78     <ref name="projectloaded">
79      <hook name="archiviz">
80       <void method="de.grogra.ext.archiviz.Init.init">
81        <var name="workbench" />
82       </void>
83      </hook>
84     </ref>
85     <ref name="configure">
86      <ref name="main">
87       <insert
88        target="/workbench/menu/src/file/openclose/new"
89        resolveLinks="false">
90        <command
91         name="archiviz"
92         class="de.grogra.ext.archiviz.NewArchiViz" />
93       </insert>
94      </ref>
95     </ref>
96    </ref>
97   </registry>
98 </plugin>
```

Listing B.2: plugin.xml

```
1  package de.grogra.ext.archiviz;
2
3  import java.io.IOException;
4  import java.util.ArrayList;
5
6  import de.grogra.graph.Graph;
7  import de.grogra.graph.impl.GraphManager;
8  import de.grogra.graph.impl.GraphTransaction;
9  import de.grogra.persistence.Transaction;
10 import de.grogra.persistence.XAListener;
11 import de.grogra.persistence.Transaction.Data;
12
13 /**
14  * Listens for transaction events.
15  *
16  */
17 public class AVZXAListener implements XAListener {
```

```
18
19   private Transaction.Reader reader;
20   private ArrayList<Transaction.Consumer> consumer;
21   private boolean registered;
22
23   public AVZXAListener(Graph graph) {
24
25     GraphTransaction transaction =
26       new GraphTransaction( (GraphManager) graph , null );
27     reader = transaction.createReader( );
28     this.consumer = new ArrayList<Transaction.Consumer>( );
29     this.registered = false;
30   }
31
32   /**
33    * Gets called, when a transaction happens.
34    */
35   public void transactionApplied( Data xa , boolean rollback ) {
36
37     reader.getQueue( ).restore( xa );
38     reader.resetCursor( );
39     for (Transaction.Consumer cons : this.consumer) {
40       try {
41         reader.supply( cons );
42       } catch (IOException e) {
43         e.printStackTrace( );
44       }
45     }
46   }
47
48   public void addSuppliable( Transaction.Consumer cons ) {
49
50     this.consumer.add( cons );
51   }
52 }
```

Listing B.3: Beispielhafte Implementierung des Interface `XAListener`

C Aufgabenstellung

C.1 Funktionale Anforderungen

Es ist ein Werkzeug auf Basis der vorhandenen Plattform GroIMP zu entwickeln, das dem Anwender die Erzeugung und Durchführung virtueller Innenraumpräsentationen und einen Vergleich mehrerer Zustände ermöglicht. Hierfür ist GroIMP um die notwendigen Editor- und Betrachtungsfunktionen zu erweitern. Vorlage bilden bekannte 3D-Modellierprogramme und VRML/X3D-Viewer. Ziel ist es also, GroIMP zu einem Editor- und zu einem Präsentationsmodul auszubauen.

C.1.1 Datenformat

Als Basisdatenformat soll X3D dienen. Es muss ein Importer und ein Exporter für eine Auswahl von X3D-Knoten, unabhängig von der späteren Spezialisierung auf die virtuelle Präsentation, geschaffen und an GroIMP angebunden werden. Zu implementierende Knotentypen sind Box, Cone, Cylinder, Sphere, Transform, Group, IndexedFaceSet, Viewport, ImageTexture, Appearance. Beim Export soll keine Eins-zu-Eins Abbildung des GroIMP-Szenegraphen erzeugt werden, sondern nur äquivalente oder vorher importierte Knoten exportiert werden. Exportierte und wieder importierte Graphen sollen möglichst äquivalent sein. Über diese Basisauswahl von X3D-Knoten hinaus können weitere Knoten speziell für die Präsentation entwickelt oder zusätzliche Knoten des X3D-Standards implementiert werden.

C.1.2 Textureditor

Der Textureditor soll ermöglichen, mehrere Fotografien einer Innenwand zu einer Textur zusammen zu fügen. Hierfür müssen

mehrere Transformationen der Fotos umgesetzt werden und in einem einheitlichen und einfachen Workflow zur Verfügung stehen. Zunächst müssen über eine Linsenkorrektur die Verzerrungen jedes Fotos korrigiert werden. Anschließend müssen z.B. über halbautomatische Mechanismen die Fotos zu einem großen Bild zusammen geführt (Panorama) und zuletzt in eine rechtwinklige[1] Form gebracht werden. Dadurch wird eine nicht verzerrte Projektion der Fotografie auf eine Innenwand als Textur möglich. Das zusammengeführte Bild kann als gängiges Bildformat gespeichert werden, optional als Photoshop-Datei.

C.1.3 Editor

GroIMP muss um verschiedene Werkzeuge und GUI-Elemente so erweitert werden, dass die Erzeugung der virtuellen Präsentation ermöglicht wird.

C.1.4 Raumerzeugung

Zunächst soll der Anwender einen Raum anlegen können. Anschließend kann der Anwender einen Grundriss z.B. als JPG-Datei importieren und dem Raum zuordnen. Nun können über das zu implementierende Werkzeug „Wanderzeugung“ durch Markierung der Eckpunkte des Raumes Wände erzeugt werden. Es stehen dem Anwender zusätzliche Optionen, wie z.B. „Wandhöhe“ zur Verfügung. Da Fenster und Türen über Texturen dargestellt werden, sind dafür keine weiteren Werkzeuge notwendig. Ebenso ist von komplexeren Formen, wie z.B. Vorsprüngen oder Nischen abzusehen. Hat der Anwender einen Raum mit einer oder mehreren Wänden angelegt, kann er diesen jeweils eine Textur zuordnen. Diese kann er importieren oder den oben beschriebenen Textureditor nutzen, um eine nicht verzerrte Textur aus mehreren Fotografien zu erzeugen. Da die Präsentation auch den Vergleich verschiedener Zustände eines Raumes enthalten soll, müssen dafür entsprechende Mechanismen implementiert werden, damit der Anwender diese Zustände festlegen kann.

[1] Spezialfälle, wie z.B. Nischen oder Absätze sollten zunächst ausgeschlossen werden. Ebenso sollte die Deckenform vorerst einfach gehalten werden (flache Decke).

C.1.5 Interaktivität

Nun sollen interaktive Möglichkeiten in der Präsentation enthalten sein. Es muss also ein Prinzip implementiert werden, wonach der Anwender Aktionen und deren zugehörigen Reaktionen festlegen kann. Typische Aktionen wären beispielsweise: Ein Klick auf ein bestimmtes Objekt der Szene, eine Reaktion z.B. ein Menü oder eine Textinformation. Es muss darüber hinaus geprüft werden, inwiefern der X3D-Standard bereits solche Mechanismen enthält und ob es sinnvoll ist, diese zu implementieren und zugänglich zu machen. Insgesamt ist GroIMP um eine intuitive GUI zu erweitern, wonach sich die Funktionalität dem Anwender erschließt und einfach anzuwenden ist. Ebenso ist das Datenformat X3D so zu erweitern, dass es die speziellen Anforderungen der Präsentation erfüllen kann. Somit sind neue Knotentypen und Attribute erforderlich und der Importer/Exporter entsprechend zu erweitern.

C.1.6 Platzierung von Objekten

Damit Objekte innerhalb des Raumes leicht platziert werden können, sind zusätzliche Hilfen bei der Platzierung notwendig. Dazu gehört beispielsweise eine Rasterfunktion, die eine Bewegung eines Objektes in diskrete Schritte unterteilt. Darüber hinaus wäre eine Snapping- Funktion, die z.B. ein Gemälde direkt an einer Wand positioniert und somit dem Anwender ein mühsames Drehen und Bewegen von Hand erspart, wichtig.

C.1.7 Viewer

GroIMP ist so zu erweitern, dass es die Funktionalität eines klassischen X3D/VRML-Viewers besitzt. Dazu gehört neben der Anpassung des Layouts von GroIMP auch die Implementierung zusätzlicher Arten der Kamerabewegung (z.B. Flug- oder Walk-Modus). Über diese Funktionalität hinaus soll der Viewer mit zusätzlich implementierten Knoten umgehen können, also auf zuvor definierte Aktionen mit einer oder mehreren definierten Reaktionen reagieren können. Die GUI von GroIMP ist so zu vereinfachen, dass der Betrachter nur für ihn wichtige GUI- Elemente erreichen und verwenden kann, optimalerweise über eine separate GUI.

C.1.8 3D-Scanner

Es ist der zur Verfügung stehende 3D-Scanner zu testen und auf die Verwendbarkeit für dieses Projekt als Input für 3D-Objekte zu prüfen. Für den Import kann eventuell der vorhandene OBJ-Importer aus GroIMP genutzt werden.

C.2 Beispielhafte Beschreibung des Workflows

Gegeben ist ein Raum mit 4 Wänden und mehreren Einrichtungsgegenständen. Zusätzlich befindet sich ein Gegenstand nicht mehr in dem Gebäude, gehörte aber vor einiger Zeit noch zur Einrichtung. Das macht es notwendig, die Digitalisierung des Objektes an einem anderen Ort und zu einem anderen Zeitpunkt durchzuführen.
Folgende Schrittfolge soll beispielhaft erläutern, wie von der gegebenen Situation bis zur fertigen Präsentation zu verfahren ist.

- Anfertigung von Fotografien des Raumes,
  wobei jede Wand möglichst von vorn zu fotografieren ist. Da die Wandgröße im Normalfall den Erfassungswinkel der Kamera übersteigt, werden mehrere überlappende Fotografien pro Wand, aber möglichst senkrecht auf die Wand und immer mit der gleichen Brennweite und Entfernung von der Wand durch den Anwender angefertigt.

- Beschaffung des Grundrisses.
  Hierbei soll der Grundriss als digitales Bild zum Import aufbereitet werden. Wenn der Grundriss noch nicht digital zur Verfügung steht, muss er zunächst gescannt oder abfotografiert werden. Ebenso sollte er anschließend digital beschnitten werden, sodass nur der Raum von Interesse abgebildet ist.

- Digitalisierung der Einrichtungsgegenstände.
  Die verschiedenen Gegenstände sind entweder durch den 3D-Scanner in ein dreidimensionales Modell zu übertragen oder - z.B. im Falle eines Gemäldes - einfach abzufotografieren. Alternativ können Gegenstände auch in anderen 3D-Modellern erzeugt und anschließend importiert werden. Der Gegenstand, der sich nicht in dem Raum zu diesem Zeitpunkt befindet, muss - wie bereits erwähnt - an einem anderen Ort bzw. zu

anderer Zeit digitalisiert werden.

- Anlegen der Szene für die Präsentation.
  Es ist nun der Editor zu öffnen und eine leere Szene anzulegen. Anschließend muss durch einen Klick auf „Raum erstellen“ ein neuer Raum angelegt werden. Innerhalb dieses Raumes können nun Wände erzeugt oder Gegenstände platziert werden.

- Erzeugung von Wänden.
  Um die Wände entsprechend des Grundrisses zu erzeugen, muss zunächst der Grundriss importiert werden. Anschließend kann dieser als Ebene angezeigt werden. Das Tool „Wand erzeugen“ ermöglicht es, durch einfaches „Abklicken“ der Wandecken entsprechend des Grundrisses die Wände zu erzeugen. Jede Wand wird ein Objekt der Szene und im Szenegraphen angezeigt. Anschließend kann der Anwender die Wandhöhe bestimmen, z.B. über eine numerische Eingabe.

- Erzeugung der Wandtexturen.
  Zunächst muss der Anwender die zuvor gemachten Fotografien der einzelnen Wände importieren. Diese sind in dieser Form noch nicht für eine Projektion geeignet. Hierfür kann der Anwender die importierten Texturen in den Textureditor laden. Dort sollte er zunächst eine Linsenkorrektur vornehmen. Hierfür kann der Anwender zunächst die optimalen Einstellungen anhand eines Bildes in Erfahrung bringen. Hat er alle anderen Fotografien mit der selben Brennweite durchgeführt, sollten diese Werte auch auf alle anderen Bilder anwendbar sein. Ein entsprechender Button steht zur Verfügung. Nun kann der Anwender die Texturen zusammenfügen. Hierfür muss er zunächst mehrere Markierungen auf jeder Textur einfügen, welche Gemeinsamkeiten festlegen. Nach einem Klick auf den entsprechenden Button wird die Gesamttextur berechnet und eine Art Panorama erzeugt. Nun sind abschließend noch die Wandecken durch den Anwender zu markieren, damit so die Wand auf dem Panorama-Foto in eine rechtwinklige Form gebracht werden kann. Optional steht nun ein Export in ein Bildformat (JPG, TIF, eventuell Photoshop) zur Verfügung, sodass eventuelle Retuschearbeiten in einem Bildbearbeitungsprogramm durchgeführt werden können.

- Zuordnung der Wandtextur.
  Nach der Erzeugung dieser Textur kann nach dem Schließen des Textureditors die neue Textur einer Wand zugeordnet werden. Die Projektion geschieht automatisch. Schritt 6 und 7 werden so lange wiederholt, bis alle Wände texturiert sind[1].

- Platzierung von Objekten.
  Nun platziert der Anwender zuvor importierte Objekte, z.B. ein Gemälde an einer Wand. Hierfür stehen verschiedene Werkzeuge zur Auswahl, die die Positionierung erleichtern.

- Festlegung eines zweiten Zustandes[2].
  Die Erzeugung eines zweiten Zustandes dupliziert die Szene vollständig und es bietet sich die Möglichkeit, den Wechsel zwischen den verschiedenen Zuständen als eine Reaktion festzulegen. Darüber hinaus kann entschieden werden, welche Objekte, Aktionen oder Reaktion in beiden Zuständen oder nur in einem von beiden gültig sind. In diesem Beispiel könnte ein Zustand den aktuellen reflektieren und ein weiterer einen vergangenen, da z.B. eines der Objekte sich nicht mehr innerhalb des Gebäudes befindet.

- Erzeugung einer Aktion und Reaktion.
  Der Anwender selektiert beispielsweise ein Gemälde und klickt auf „Aktion hinzufügen“. Daraufhin öffnet sich ein Menü, in dem zunächst eine Aktion und anschließend eine Reaktion auswählbar ist. Mögliche Aktionen sind z.B. Mausklick, Doppelklick, Rechtsklick; mögliche Reaktionen sind „Menü“, „Text“ oder „Fly to“, ebenso ein Zustandswechsel wäre denkbar. Für ein Menü kann der Anwender mehrere Menüpunkte definieren und jeweils wieder Reaktionen zuordnen. „Text“ zeigt eine Textmeldung an, z.B. wann das Gemälde entstanden ist und von

[1] Theoretisch kann dieses Konzept auch auf die Decken- und Bodentextur angewendet werden, jedoch ist bei den Fotografien mit mehr Verzerrungen zu rechnen, da ein senkrechtes Fotografieren schwieriger durchzuführen ist. Alternativ ist ein Mustergenerator denkbar, dessen Implementierung allerdings den zeitlichen Rahmen übersteigen würde. Darüber hinaus bieten Programme wie Photoshop bereits Mechanismen dieser Art. Ebenso existieren in GroIMP bereits mehrere prozedurale Texturgeneratoren, eventuell sind diese nutzbar.

[2] Alternativ zu dem Begriff „Zustand“ kann auch „Layer“ gewählt werden. Hierbei definiert ein Layer in GroIMP einen Zustand, wobei ein Objekt in mehreren Layern, also auch in mehreren Zuständen auftreten kann. Es muss noch geprüft werden, ob das Layer-Konzept aus GroIMP hierfür verwendet werden kann.

wem. „Fly to“ transportiert die Kamera an einen bestimmten Punkt in der Szene, welchen der Anwender durch Platzierung eines Zielobjektes definieren kann.

- Erzeugung der interaktiven Präsentation.
  Im letzten Schritt speichert der Anwender die erstellte Präsentation. Die daraus resultierende X3D-Datei enthält alle Daten, um die Präsentation zu einem anderen Zeitpunkt zu verändern, also sie wieder im Editor zu laden, und um die Präsentation im Viewer ablaufen zu lassen.

- Durchführung der interaktiven Präsentation.
  Nun lädt der Anwender die X3D-Datei im Präsentationsmodul. In diesem sind die Editorfunktionen vollständig deaktiviert, sodass ausschließlich die zuvor definierten Aktionen durchgeführt werden können. Darüber hinaus hat der Betrachter verschiedene Werkzeuge zur Kamerabewegung, um sich durch die Szene zu bewegen. Klickt er auf das Gemälde, zu dem vorher die Reaktion auf den Mausklick definiert wurde, wird entsprechend ein Menü oder ein Text angezeigt.

Literaturverzeichnis

[Bischof 2006] BISCHOF, Udo: *Anbindung der Programmiersprache XL an die 3D-Modelliersoftware Maya über ein Plug-in*, Brandenburgische Technische Universität, Bachelor-Arbeit, 2006

[Bourke 2002] BOURKE, Paul: *Lens Correction And Distortion*. http://local.wasp.uwa.edu.au/~pbourke/projection/lenscorrection/. April 2002. – Online; Zugriff am 15.05.2008

[Brown 2008] BROWN, Matthew: *Autostitch*. http://www.cs.ubc.ca/~mbrown/autostitch/autostitch.html. 2008. – Online; Zugriff am 18.10.2008

[Brown und Lowe 2007] BROWN, Matthew ; LOWE, David G.: Automatic Panoramic Image Stitching using Invariant Features. In: *International Journal of Computer Vision* 74 (2007), August, Nr. 1, S. 59–73. – ISSN 0920-5691

[Burger und Burge 2006] BURGER, Wilhelm ; BURGE, Mark J.: *Digitale Bildverarbeitung - Eine Einführung mit Java und ImageJ*. 2. Auflage. Berlin, Heidelberg : Springer-Verlag, 2006. – ISBN 3-540-30940-3

[Dawes u. a. 2008] DAWES, Beman ; ABRAHAMS, David ; RIVERA, Rene: *Boost C++ Libraries*. http://www.boost.org. 2008. – Online; Zugriff am 25.11.2008

[Dersch und Dersch 2001] DERSCH, Astrid ; DERSCH, Helmut: *Panorama Tools*. http://home.no.net/dmaurer/~dersch/index_deutsch.html. 2001. – Online; Zugriff am 02.10.2008

[DevMaster.net 2003–2009] DEVMASTER.NET: *DevMaster.net - 3D Game and Graphics Engines Database*. http://www.devmaster.net/engines/. 2003-2009. – Online; Zugriff am 12.08.2008

[Dönitz] DÖNITZ, Stephan: *Geschichte des Fliegens - Die Anfänge bis zum ersten Weltkrieg*. http://www.luftrettung-hamburg.de/html/pioniere.html. – Online; Zugriff am 04.10.2008

[Edlich und Staudemeyer 2006] EDLICH, Stefan ; STAUDEMEYER, Jörg: *Ant - kurz und gut*. 2. Auflage. Köln : O'Reilly Verlag, 2006. – ISBN 3-89721-519-5

[Gamma u. a. 2004] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuster*. München : Addison Wesley Verlag, 2004. – ISBN 3-8273-2199-9

[Goetz u. a. 2006] GOETZ, Brain ; PEIERLS, Tim ; BLOCH, Joshua ; BOWBEER, Joseph ; HOLMES, David ; LEA, Doug: *Java Concurrency In Practice*. Upper Saddle River, NJ : Addison-Wesley, 2006. – ISBN 978-0-321-34960-6

[Graphisoft Deutschland GmbH 2009] GRAPHISOFT DEUTSCHLAND GMBH: *Graphisoft Home Page*. http://www.graphisoft.de. 2009. – Online; Zugriff am 02.11.2008

[Horstmann und Cornell 2008a] HORSTMANN, Cay S. ; CORNELL, Gary: *Core Java Volume 1 - Fundamentals*. 8. Auflage. Upper Saddle River, NJ et al. : Prentice Hall Sun Microsystems Press, 2008. – ISBN 978-0-13-235476-9

[Horstmann und Cornell 2008b] HORSTMANN, Cay S. ; CORNELL, Gary: *Core Java Volume 2 - Advanced Features*. 8. Auflage. Upper Saddle River, NJ et al. : Prentice Hall Sun Microsystems Press, 2008. – ISBN 978-0-13-235479-0

[hugin] *hugin - Panorama photo stitcher*. http://hugin.sourceforge.net/. – Online; Zugriff am 02.10.2008

[Knauer 2002] KNAUER, Roland: *Entwerfen und Darstellen, Die Zeichnung als Mittel des architektonischen Entwurfs*. 2. Auflage. Berlin : Verlag für Architektur und technische Wissenschaften GmbH, 2002. – ISBN 3-433-01267-9

[Kniemeyer a] KNIEMEYER, Ole: *How to Create a Node Class*. http://groimp.wiki.sourceforge.net/howto-create-node-class. – Online; Zugriff am 26. Januar 2009

[Kniemeyer b] KNIEMEYER, Ole: *How to Create a Plugin*. http://groimp.wiki.sourceforge.net/howto-create-plugin. – Online; Zugriff am 26. Januar 2009

[Kniemeyer 2008] KNIEMEYER, Ole: *Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling*, Brandenburgische Technische Universität, Dissertation, 2008

[Krüger 2006] KRÜGER, Guido: *Handbuch der Java-Programmierung*. München : Addison-Wesley, 2006. – ISBN 978-3-8273-2447-4

[Kurth u. a. 2008] KURTH, Winfried ; KNIEMEYER, Ole ; HEMMERLING, Reinhard: *grogra.de - Growth Grammars*. Lehrstuhl Graphische Systeme BTU Cottbus; `http://www.grogra.de`. 2008. – Online; Zugriff am 28.11.2008

[Lengyel 2002] LENGYEL, Dominik: The Presentation is the Project. In: *Proceedings of the 5th European Architectural Endoscopy Association Conference*, *`http://cumincad.scix.net/cgi-bin/works/Home`*, 2002. – Online; Zugriff am 20.01.2009

[Lengyel und Toulouse] LENGYEL, Dominik ; TOULOUSE, Catherine: *Lengyel Toulouse Architekten*. `http://www.lengyeltoulouse.com`. – Online; Zugriff am 25.01.2009

[Luhmann 2003] LUHMANN, Thomas: *Nahbereichsphotogrammetrie, Grundlagen, Methoden und Anwendungen*. 2. überarbeitete Auflage. Heidelberg : Herbert Wichmann Verlag, 2003. – ISBN 3-87907-398-8

[Mannl 2006] MANNL, Uwe: *Anbindung der Programmiersprache XL an die 3D-Modelliersoftware 3D-Studio Max über ein Plug-in*, Brandenburgische Technische Universität, Bachelor-Arbeit, 2006

[Meyer] MEYER, Matthias: *Lightfeather 3D Engine*. `http://lf.mmdevel.de`. – Online; Zugriff am 12.08.2008

[Möller 2005] MÖLLER, Birgit: *Ein Ansatz zur ikonischen Repräsentation von Bilddaten aktiver Kameras*, Martin-Luther-Universität, Dissertation, Mai 2005

[Möller und Haines 1999] MÖLLER, Tomas ; HAINES, Eric: *Real-Time Rendering*. Natick, Massachusetts : A K Peters Ltd., 1999. – ISBN 1-56881-101-2

[Nischwitz u. a. 2007] NISCHWITZ, Alfred ; FISCHER, Max ; HABERÄCKER, Peter: *Computergrafik und Bildverarbeitung*. 2. Auflage. Wiesbaden : Friedr. Vieweg & Sohn Verlag — GWV Fachverlage GmbH, 2007. – ISBN 978-3-8348-0186-9

[Nokia Cooperation 2008–2009] NOKIA COOPERATION: *Qt Software*. http://www.trolltech.com. 2008-2009. – Online; Zugriff am 24.11.2008

[Prusinkiewicz und Lindenmayer 1990] PRUSINKIEWICZ, Przemyslaw ; LINDENMAYER, Aristid: *The Algorithmic Beauty Of Plants*. New York : Springer-Verlag, 1990. – ISBN 0-387-97297-8

[Schumann und Müller 2000] SCHUMANN, Heidrun ; MÜLLER, Wolfgang: *Entwerfen und Darstellen, Die Zeichnung als Mittel des architektonischen Entwurfs*. Berlin, Heidelberg : Springer-Verlag, 2000. – ISBN 3-540-64944-1

[sourceforgeGroIMP] *Sourcecode von GroIMP*. https://groimp.svn.sourceforge.net/svnroot/groimp. – Online; Zugriff am 01.12.2008

[Szeliski 2006] SZELISKI, Richard: Image Alignment and Stitching: A Tutorial / Microsoft Research. Dezember 2006. – Technical Report. ftp://ftp.research.microsoft.com/pub/tr/TR-2004-92.pdf; Online; Zugriff am 26. Januar 2009

[TBK 1997–2007] TBK: *TBK - Digital Panorama Technologie*. http://www.tbk.de/panorama/show2.htm. 1997-2007. – Online; Zugriff am 1.7.2008

[Wippler 2006] WIPPLER, Daniel: *Algorithmen und Graphik mit Java*. Norderstedt : Books on Demand GmbH, 2006. – ISBN 3-8334-4452-5

[Xiang und Plastock 2003] XIANG, Zhigang ; PLASTOCK, Roy A.: *Computergrafik - IT-Studienausgabe*. Bonn : mitp-Verlag, 2003. – ISBN 3-8266-0908-5

[Zeppenfeld 2004] ZEPPENFELD, Klaus: *Lehrbuch der Grafikprogrammierung - Grundlagen Programmierung Anwendung*. Heidelberg Berlin : Spektrum Akademischer Verlag, 2004. – ISBN 3-8274-1028-2

Index

3D-Fenster, 96, 99

Abbildung, 138
  projektive, 110
Aktions-Reaktions-Editor, 84
Anforderungen
  funktional, 30
  funktionale, 29
  nichtfunktional, 30
  nichtfunktionale, 29
Ant, 54, 58
API, 28
ArchiCAD, 36
Architektur, 16, 21, 35, 36, 83
Architekturvisualisierung, 16, 19
ArchiViz, 56, 70, 83–94, 96, 98, 99, 136, 137
  GUI, 56
  Kern, 85
  Knoten
    AVZActionNode, 89
    AVZCeilingNode, 89
    AVZFloorNode, 89
    AVZGlobalStateNode, 89
    AVZGroundPlanNode, 88
    AVZImageContainerNode, 88
    AVZImageNode, 88
    AVZImageSetNode, 88
    AVZIntersectionPointNode, 89
    AVZMeshNode, 87, 88
    AVZNode, 87, 88, 92
    AVZNullNode, 87
    AVZPointNode, 87
    AVZReactionNode, 89
    AVZRoomNode, 87
    AVZRoot, 90
    AVZRootNode, 87
    AVZSketchingLineNode, 89
    AVZSketchingLinePointNode, 89
    AVZSketchingWallNode, 89
    AVZStateEntryNode, 89
    AVZStateServerNode, 89
    AVZWallNode, 88
  Module, 84, 85
  parallele Nutzung, 99
Artifizielle Wachstumsprozesse, 35
Attribut
  Reaktion auf Veränderung, 67
Auseinandersprenung, 102
Auslösen
  automatisches, 138
  manuelles, 138
Auswertung
  bidirektional, 138
  parallele, 138
  photogrammetrische, 109

sequentielle, 138
unidirektional, 138
Autodesk Maya, 38
AVZEdge, 86, 89, 90
AVZJob, 95
AVZMeshNode, 96, 98
AVZRoot, 98
AWT, 29

Baum, 89
Betrachtungswinkel, 107
Bilddatei, 92
Bilddateien, 91
Bildelement
Verzerrung, 107
Bilder, 91
Bildformat, 108
Bildqualität, 18
Bildverarbeitung, 108, 109
Blender, 68
Boost, 28
BSContact, 28

C++, 28
CAD, 13, 17, 19
Canvas, 84, 87, 92–94
Consumer, 67

Datenformat, 99
Designer, 14, 20, 23, 24
DirectX, 22
DXF, 38, 53

Echtzeit, 22
Eclipse, 29, 53
Editormodus, 84
enhance-Skript, 86
Ereigniskette, 95
Event, 93
MouseRelease, 94, 95

Flächeninhalt, 101, 103

Fotografie, 106, 107

Gebäude, 37
Gemälde, 23
Geoinformationssystem, 109
Geometrie, 36, 38, 42, 88, 89, 101, 108
Erzeugung, 35
polygonale, 88
projektive, 110
regelbasierte Generierung, 34
Geometrieknoten, 88, 98
GPECanvas, 93–95
GPEViewport, 93, 94
Graph, 89, 93
GroIMP, 29, 31–38, 40–42, 45, 46, 50, 53–59, 61–64, 67–69, 83–97, 99, 103, 196
Aktionen und Reaktionen, 89
Button, 59
Einsatzbereich, 31
Erweiterung, 52
Event, 64, 66
Hooks, 63, 64
Initialisierung, 55
Kantenkonzept, 86
Kern, 85
Knoten, 98
Koordinatensystem, 97
Layout, 31–33, 62
Panel, 60–62
Name des, 61
Registrierung, 51, 55, 59, 61
Toolbar, 59, 62
Workflow, 68
Zustände, 62, 63

Grundriss, 23, 36
Grundrisseditor, 83, 89
GUI, 14, 23, 27, 61, 84, 196

Höhlenmalerei, 12
Hashmap, 93–95
    Aktualisierung, 93
Hauptspeicher, 91
Highlighting, 20
Homogenität, 20
Hook, 86

ImageJ, 29
Import, 97
Importierung, 98
Informationsgehalt, 13, 21
Innenräume
    Darstellung, 32
Innenräumen
    Modellierung von, 31
Innenraum, 8, 105
Innenwände, 10, 83
Innenwand, 22–24, 88, 91, 101, 105–109
    Anfangs- und Endpunkt, 86
    Geometrie, 109
    Länge, 101
    Richtung, 101
Interaktion, 14, 136
Interaktivität, 18, 21
    Grad der, 18, 19, 136
Interaktivität, hohe, 18
invokeLater-Konzept, 45

Java, 28, 29, 31
    32-Bit-Version, 91
JEdit, 34
Job, 46, 47, 93, 95

Kamerastandort, 110
Kante, 103
Kartographie, 109
Koordinatensystem, 29, 97

Lascaux, 12
Layer, 99
Listener, 93
Luftbildphotogrammetrie, 109

Möbelstück, 23
Maßeingabe, 104
MeshNode, 95, 96
Modellierung
    bauteilorientierte, 36
    von 3D-Objekten, 31

Nahbereichsphotogrammetrie, 109, 110
    Aufnahmeentfernung, 109
Node, 41
NURBS, 34, 42

OBJ, 38, 97, 197
OpenGL, 22, 28, 31–33, 42

Perspektive, 19
    Wahl der, 19
Pfad, 91, 92
    relativ zur Projektdatei, 91
Pfad, relativ zur Projektdatei, 91
Pflanzenbibliothek, 37
Pflanzenobjekte, 37
Phong, 42
Photogrammetrie, 109
    digitale, 109
    terrestrische, 109
Pixel, 29
    Darstellung eines, 42
    Farbe, 105
Planarprojektion, 106
Plattform, 26, 27, 29

Wahl der, 26
Plugin, 27, 34
Polygon, 88, 98
Speicherung, 88
polygonales Objekt, 88
Polygonizable, 98
PovRay, 42
Präsentation, 10, 24, 84
interaktive, 18
Präsentationsmodus, 24, 84
Produktionskette, 36–38
Integration in, 36
Programmiersprache, 26, 29
regelbasiert, 31
Projekt, 54, 56, 91
Auschecken, 54
Speicherung, 91
Projektdatei, 91
Projektion, 108
orthogonale, 107
perspektivische, 107
Projektionsverfahren, 106
Projektpfad, 91
Projektverwaltung, 91

Qt, 27, 28
Quelloffenheit, 28
Queue, 45, 48

Raum, 23, 24, 91, 101–103, 105–107
Boden, 89
Decke, 89
zentraler Punkt, 102
Zentrum, 102
Raumdefinition, 102, 103
Raumverteilung, 101
logische, 101
Raytracer, 38, 42
Raytracing, 31, 96
Realismus, 19
Grad des, 19, 20
Renderer, 38
Maxwell Render, 38
PovRay, 38
Twilight, 38
Rendering, 38
RGG, 98
Knoten, 98, 99
Root, 98
RGG-Plugin, 98
Rotation, 97
Rotationswerkzeug, 97

Schnittlinie, 101–104
Schnittpunkt, 89, 101, 103, 104
Kette, 103
Selektion, 103
Schnittpunkte, 102
Selektion, 96
Serialisierung, 52, 58, 99
Shader, 32, 37, 42, 105
Shadereditor, 83
Sketching, 89
Sicht auf ein, 101
Sketchingline, 87, 89
Stereoprinzip, 109
Stitching, 107, 108, 110
Struktur
räumliche, 110
subclipse, 53
Subversion, 53
successor-Kante, 90
Swing, 27, 29, 45, 46, 60, 61, 89
Synchronisation, 49
Szene
dreidimensionale, 136
Serialisierung, 52, 99
virtuelle, 137

Szenegraph, 31, 35, 39, 41, 45, 64, 83, 84, 86, 87, 89, 90, 92–94, 98, 99, 103
  Änderung des, 94
  Basis des, 87
  gefiltertes Parsen, 86
  Kanten, 35, 39, 89
    Veränderung von, 85
  Kantenmarkierung, 39, 90
  Kantentypen, 39, 40, 44
  Knoten, 35, 39, 40, 87, 88, 90–95, 98, 99, 103
    Attribute, 35
    Hinzufügen und Entfernen, 85
    Sichtbarkeit, 91
  Knotentypen, 44
  Konsistenz, 45
  Parsing-Funktionen, 86
  Pfad, 90
  successor-Kante, 89, 91
  successor-Kanten, 91
  traversieren des, 43

Textur, 22, 83, 105, 106, 108
  glaubewürdige, 105
  Projektion, 106, 108
Texture, 83
Textureditor, 83, 84
Tiefeninformation
  Rückgewinnung, 110
To-Ground-Tool, 97
Transform3D, 41
Transformation, 40–42, 94, 98
  globale, 41
  Kamera, 31
  lokale und globale, 87
  Transformationsfunktion, 41
  Transformationshierachie, 41
  Transformationsmatrix, 41
Transformationsfunktion, 41
Transformationshierarchie, 41
Transformationsknoten, 98

Viewport, 93
Viewports, 94
visiblity-Kante, 91
Visualisierung, 10, 12–21, 31, 35–37, 42, 136
  dreidimensionale, 31
  Extrema, 14
  funktionale, 14
  hohe Interaktivität, 137
  künstlerische, 14
  non-interaktive, 20
  Pflanzen, 35
  Pflanzen in einer, 37
  rechnergestützte, 19
  spezielle Klassifikation, 15
  von Innenräumen, 17
  Wirkung einer, 20
  wissenschaftlich-technisch, 14
  wissenschaftliche, 14
Visualisierung, interaktiv, 18
Visualisierung, non-interaktiv, 18
Visualisierung, semi-interaktiv, 18
Visualisierungstypen, 14

Wand, 88, 102
Wandecke, 102, 108
Wandgeometrie
  Erzeugung, 104
Wandkante, 108
Wings3D, 68
Wurzelknoten, 93

X3D, 38, 53, 99, 178, 194, 196
    Bibliothek, 28
    Browser, 28
Xj3D, 28
XL, 31, 32, 34–38, 53, 69, 98
    Regel, 32, 35
XML, 54, 99
XY-Ebene, 97

Zeichnung, 17
Zustand, 84, 89, 91
Zyklus, 103
    minimaler, 103