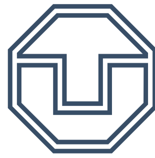


Fakultät Informatik

Institut für Software- und Multimediatechnik

Professur für Mediengestaltung

Konzeption und Realisierung einer Komponentenarchitektur für die Arbeitsumgebung Bildsprache LiveLab (BiLL)

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Medieninformatiker

eingereicht von:

Jan Wojdziak

30.09.2007

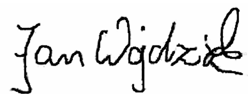
Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Diplomarbeit zum Thema:

„Konzeption und Realisierung einer Komponentenarchitektur für die Arbeitsumgebung Bildsprache LiveLab“

eigenständig verfasst habe. Es wurden keine anderen Quellen und Hilfsmittel als die angegebenen benutzt.

Dresden, 30.09.2007



Jan Wojdziak

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Diplomarbeit beigetragen haben.
Besonderer Dank gebührt Prof. Dr.-Ing. habil. Rainer Groh und Herrn Ingmar Franke, M.Sc. für die Betreuung meiner Diplomarbeit und die zahlreichen wissenschaftlichen Ratschläge, welche stets zur Verbesserung meiner Arbeit beigetragen haben. Des Weiteren möchte ich mich bei Thomas Ebner für seine hilfreichen Anregungen und die vielen Diskussionen und Gespräche bedanken. Ein ganz herzliches Dankeschön geht an Nadine Hohmann, die eigene Interessen oftmals zugunsten meiner Arbeit zurückstellte, mich während der gesamten Diplomarbeitsphase moralisch unterstützte und mir bei Formulierungsproblemen zur Seite stand.
Nicht zuletzt möchte ich mich bei meinen Eltern bedanken, die mir dieses Studium überhaupt erst ermöglicht haben.

Daten zur Diplomarbeit

Technische Universität Dresden

Fakultät Informatik, Institut für Software– und Multimediatechnik

Lehrstuhl für Mediengestaltung

Diplomthema im Studiengang Medieninformatik

Hochschullehrer:	Prof. Dr.–Ing. habil. Rainer Groh
Betreuer:	Dipl.–Ing. Arch. Ingmar S. Franke M.Sc. CV.
Diplomand:	Jan Wojdziak
Matrikelnummer:	2860306
Thema:	Konzeption und Realisierung einer Komponentenarchitektur für die Arbeitsumgebung Bildsprache LiveLab (BiLL)
Bearbeitungszeit:	01.04.2007 – 30.09.2007

Kurzbeschreibung

In dieser Arbeit stehen die Konzeption und die Realisierung einer Komponentenarchitektur für ein bestehendes Softwaresystem im Mittelpunkt. Nach einführenden Worten und der Vorstellung relevanter Begriffe aus den Bereichen Bildsprache, Computergrafik und Softwaretechnologie werden Klassenbibliotheken, die Grundvoraussetzung und Bedingung für die theoretischen Überlegungen sowie praktischen Realisierungen sind, vorgestellt. Daraufhin erfolgt die Darlegung verwandter Arbeiten mit dem Fokus auf Komponenten– und Plug–in Architekturen und die Trennung der beiden Begrifflichkeiten. Außerdem werden zwei Ansätze zur Reduktion von Weitwinkelverzerrungen bei der computergrafischen Zentralprojektion aufgezeigt. Dies ist zum einen die Erzeugung von Multiperspektive in Bildern durch den Einsatz mehrerer Kameras in einer 3D–Szene und zum anderen die Erzeugung von Binnenperspektiven durch die geometrische Veränderung von Objekten mit Hilfe des Verfahrens der erweiterten perspektivischen Korrektur (EPK). Mit beiden Vorgehensweisen ist die Erzeugung von multiperspektivischen Abbildungen dreidimensionaler Szenen umsetzbar.

In der nachfolgenden Synthese werden Rahmenbedingungen für Echtzeitanwendungen im Allgemeinen sowie im Bezug auf die Anwendungssoftware Bildsprache LiveLab (BiLL) untersucht. Des Weiteren werden die kontextabhängigen sowie die softwaretechnologischen Anforderungen an die Echtzeitanwendung BiLL analysiert. Das Ziel dieser Untersuchungen ist die Herbeiführung einer Entwurfsentscheidung für die Entwicklung einer Architektur zur möglichst unkomplizierten Erweiterbarkeit des Programms durch Drittentwickler. Der Entscheidung zugunsten einer Plug–in Architektur folgt eine detaillierte Konzeption zur Neustrukturierung und zum Ausbau der bestehenden Arbeitsumgebung. Darüber hinaus wird ein Entwurf für die Integration des Verfahrens der erweiterten perspektivischen Korrektur als Erweiterung der Basisanwendung, in Form eines Plug–ins, entwickelt. Darauf aufbauend erfolgt die praktische Umsetzung der Architektur in der Arbeitsumgebung. Diese wird in der Arbeit beschrieben und dokumentiert. Abschließend werden Versuche, zur Klärung der bildlichen Eigenschaften dynamischer multiperspektivischer Abbildungen, welche mit der umgesetzten Plug–in Architektur in BiLL erstellt werden durchgeführt. Die Arbeit wird mit einem Fazit über die erreichten Resultate und einem Ausblick auf fortführende Arbeiten abgeschlossen.

Inhaltsverzeichnis

1. EINLEITUNG 1

1.1 MOTIVATION 1

1.2 ZIELSTELLUNG 4

1.3 GLIEDERUNG 5

2. GRUNDLAGEN 6

2.1 BEGRIFFE DER SOFTWARETECHNOLOGIE 6

2.2 BEGRIFFE DER COMPUTERGRAFIK 8

2.3 KLASSENBIBLIOTHEKEN 13

3. VERWANDTE ARBEITEN 16

3.1 DIE KOMPONENTENBASIERTE SOFTWAREENTWICKLUNG 16

3.2 DIE PLUG–IN ARCHITEKTUR 19

3.3 DIE MULTIPERSPEKTIVISCHE ABBILDUNG 21

4. SYNTHESE UND KONZEPTION 28

4.1 ECHTZEITANWENDUNGEN 28

4.2 DIE ARBEITSUMGEBUNG BILDSPRACHE LIVELAB (BILL) 29

4.3 BILL: EIGNUNG ALS KOMPONENTENARCHITEKTUR 32

4.4 BILL: KONZEPTION DER PLUG-IN ARCHITEKTUR 40

4.5 DIE ERWEITERTE PERSPEKTIVISCHE KORREKTUR ALS PLUG–IN 50

5. PRAKTISCHE UMSETZUNG 53

5.1 BILL: DIE ARBEITSUMGEBUNG 53

5.2 BILL: UMSETZUNG DER PLUG–IN ARCHITEKTUR 54

5.3 DIE UMSETZUNG DES EPK–PLUG–INS 65

6. ZUSAMMENFASSUNG 76

6.1 INHALT 76

6.2 FAZIT 77

6.3 AUSBLICK 77

A GLOSSAR 82

B ABKÜRZUNGSVERZEICHNIS 85

C LITERATURVERZEICHNIS 86

D ABBILDUNGSVERZEICHNIS 90

E TABELLENVERZEICHNIS 92

F ZUSATZELEMENTE 92

Zur Verdeutlichung von bestimmten Sachverhalten, befinden sich auf der Begleit–CD interaktive Referenzen und Animationen. Über die dem Text beigefügten Verweise können diese auf der CD-ROM aufgerufen werden.

1. Einleitung

In den nachfolgenden Abschnitten wird auf die Motivation und die daraus abgeleiteten Zielstellungen und speziellen Aufgaben dieser Arbeit eingegangen. Den Schwerpunkt der Betrachtungen stellt die Visualisierung und Manipulation virtueller dreidimensionaler Welten dar. Anhand theoretischer Ausführungen sowie praktischer Anwendungsbeispiele wird die Thematik aufgezeigt und umrissen. Anschließend werden die inhaltlichen Schwerpunkte der einzelnen Kapitel aufgezeigt und somit eine Übersicht über die Arbeit gegeben.

1.1 Motivation

In vielen Bereichen haben 3D-Anwendungen einen progressiven Einfluss auf unser Leben. Bisher fokussierte sich der Wirkungsbereich auf den Unterhaltungssektor wie beispielsweise die Spiele– oder Filmbranche. Bei den Entwicklungen computeranimierter Filme wie „Shrek“ (Dreamworks 2001) oder „Ice Age“ (20th Century Fox 2002) werden 3D–Visualisierungssoftware zur Erzeugung virtueller Welten verwendet. Der Einsatz derartiger Werkzeuge ermöglicht, genau wie bei der Entwicklung von Computerspielen, hauptsächlich der Unterhaltung des Konsumenten. Das Potential virtueller Welten für den Endverbraucher ist damit aber keineswegs ausgeschöpft. Insbesondere die Interaktion mit dreidimensionalen Welten hat in der letzten Zeit stark an Bedeutung gewonnen. Die Wechselbeziehung zwischen Mensch und Computer in Echtzeit, wie sie in Computerspielen schon seit vielen Jahren für einen Dialog benutzt wird, findet allmählich Eingang in die Interfacegestaltung. Dies zeigt sich in den Portierungen von Benutzeroberflächen aus der Zweidimensionalität in eine dreidimensionale Umgebung. Die Interaktion mit einer Anwendung kann dabei ausschließlich in Echtzeit erfolgen. Durch die sich stetig erweiternden Rechenkapazitäten treten 3D–Echtzeitanwendungen in diesem Zusammenhang immer weiter in den Vordergrund. Die Projektion dreidimensionaler Objekte auf eine zweidimensionale Fläche unter Beibehaltung eines räumlichen Eindruckes, wie beispielsweise im Projekt „Looking Glass“ von Sun® Microsystems (Abbildung 1.1), eröffnet neue Möglichkeiten in der Interfacegestaltung. Derartige Vorhaben beginnen die Desktop Metapher als bildnerisches Steuerelement für technische Strukturen zu durchbrechen und die dreidimensionale virtuelle Welt als Interface wie es von GROH definiert wird (vgl. [Groh 05]) zu nutzen. Dazu ist es notwendig ein Datenbild bereitzustellen, welches den Nutzer–Bild–Dialog unterstützt. Hierbei spielt das Projektionsverfahren eine fundamentale Rolle.

Abbildung 1.1: Projekt „Looking Glass“ von Sun® Microsystems

Mit Hilfe von Projektionsverfahren wie beispielsweise die Linear– oder die Parallelprojektion erfolgt die Abbildung einer computergrafisch erzeugten Welt auf dem Ausgabegerät. Dies kann eine Anwendung befähigen Datenstrukturen weitestgehend dialogorientiert darzustellen. Dabei müssen die visualisierten Elemente für den Anwender möglichst wahrnehmungskonform dargestellt werden. Die Parallelprojektion kann diesem Anspruch nicht umfassend genügen, weil eine Tiefenwirkung in der Abbildung der dreidimensionalen Welt durch den Betrachter nicht wahrgenommen wird. Auch die Zentralprojektion ist mangelhaft, da es bei großem Kameraöffnungswinkel zu Verzerrungen der Objekte in den äußeren Bildbereichen kommt. Dieser Effekt ist den mathematischen Abbildungsvorschriften geschuldet, denen sich die Objekte der virtuellen Welt unterwerfen. In Folge dessen ist ein gestörter Dialog zwischen Anwender und Abbildung zu verzeichnen, weil Freiformen wie beispielsweise abgebildete Säulen oder Kugeln nicht der menschlichen Wahrnehmung entsprechen.
Die Prinzipien der Perspektive sind schon seit dem Altertum bekannt und seit der Renaissance fundiert. Künstler haben seit der Renaissance eine Reihe von Techniken entwickelt, Bildkompositionen von dreidimensionalen Welten abweichend von der Zentralprojektion zu erzeugen. Eine der am weitesten verbreiteten Verfahren ist die Erzeugung von Binnenperspektiven und damit die Kombination mehrerer Perspektiven in einem ansonsten systemräumlichen Kontext. PANOFSKY spezifiziert dies als Systemraum (vgl. [Panofsky 85]). In der Malerei der Renaissance werden auf diese Weise perspektivische Verzerrungen unterbunden. Denn eine multiperspektivische Abbildung entspricht nach GROH (vgl. [Groh 05]) mehr den natürlichen Sehgewohnheiten des Menschen, weil die Objekte primär in der Frontalansicht wahrgenommen werden. Der Wahrnehmungskonformität waren sich die Maler der Renaissance bewusst.
Der Einsatz von Binnenperspektiven erfolgt darüber hinaus zur Betonung der narrativen Bedeutsamkeit von Gegenständen durch deren Darstellung in einer eigenen Perspektive. Daraus folgt die Hervorhebung des Objektes aus dem Kontext des Bildes. Diese Abweichung von der Zentralprojektion birgt Möglichkeiten in der ergonomischen

Dialoggestaltung (vgl. [Franke et al. 2005b]). Denn durch den Einsatz von Multiperspektive kann die Verständlichkeit einer Abbildung erhöht und Informationen über die Struktur von Objekten und deren räumliche Verbindungen in einer dreidimensionalen Szene besser vermittelt werden (vgl. [Agrawala et al. 2000]).
Anwendungen, zur Erzeugung eines geeigneten Nutzer–Bild–Dialogs, befinden sich derzeit im Forschungsstadium. Diese müssen neben dem computergrafischen Fortschritt ebenso dialoggestalterischen Anforderungen Rechnung tragen. Die Realisierung derartiger Applikationen ist ein evolutionärer Prozess. Rahmenbedingungen, Arbeitsabläufe oder auch das gesamte Umfeld, in dem die Software eingesetzt wird, befindet sich in einem stetigen Fluss. Um den Anforderungen ihrer Benutzer gerecht zu werden, wird die Anwendung dem Wandel ihrer Umgebung angepasst. Dies bedingt, dass von Zeit zu Zeit installierte Softwareeinheiten durch Andere ersetzt oder erweitert werden. Dies ist vor allem in Bereichen wie beispielsweise den multimedialen Anwendungen, die einem schnellen Entwicklungsprozess unterworfen sind, erfahrbar. Kaum eine Branche ist so dynamisch wie die der Computergrafik. Systeme zur Erzeugung und Präsentation interaktiver Welten unterliegen dadurch einem ununterbrochenen Entwicklungsprozess. Dennoch ist es den Entwicklern nicht möglich alle Bedürfnisse der Nutzer zu befriedigen. Um dessen ungeachtet eine Software individuell für den Anwender anpassbar zu machen, kann eine Applikation als offene Architektur konzipiert werden. Dadurch können Anwender die Programme hinsichtlich ihrer persönlichen Anforderungen modifizieren und anpassen. Dies könnte die Umgestaltung der Benutzeroberfläche ebenso wie die Ergänzung von Charakteristiken involvieren. Eine entsprechende Anpassbarkeit basiert auf einem zugrunde liegenden Framework, bestehend aus der Basisfunktionalität und Komponenten die diese modifizieren und erweitern. Diese Ergänzungen können als Zusatzkomponenten realisiert werden. Damit kann das Erweitern der Funktionalität dynamisch während der Laufzeit erfolgen und eine Anwendung wird demzufolge situativ anpassbar. Adäquate Anwendungen haben sich bereits weit verbreitet und werden oft verwendet. Ein in diesem Zusammenhang allgemein gebräuchliches Programm ist der Internet Browser. Dieser erkennt automatisch zur Laufzeit beim Laden einer Webseite das Erfordern weiterer Funktionalität und stellt diese umgehend bereit. Damit setzt sich der Browser aus einer Basisfunktionalität und erweiternden Softwareeinheiten zusammen, mit denen sich dieser den Gegebenheiten anpasst und damit dem Anwender eine optimale Umgebung bereitstellt.
Die Zerlegung von Software in Einheiten spiegelt den Ansatz der Objektorientierung wieder. Dieses Prinzip erlaubt es komplexe Aufgaben zu strukturieren und dadurch konkrete Probleme zu lösen und den praktischen Erfordernissen gerecht zu werden. Dies erfolgt durch die in der Methodik definierte angewandte Teilung von Aufgaben. Jede Komponente vollzieht dabei den ihr zugewiesenen Teil einer Gesamtaufgabe und stellt ein durch sie ermitteltes Resultat bereit. Wenn im Prozess der Softwareentwicklung durch veränderte Bedingungen ein weiteres Segment hinzukommt, kann

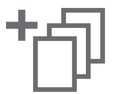
eine eigenständige Komponente, ohne eine Neustrukturierung des Programms zu erzwingen, integriert werden und den zusätzlichen Bereich abdecken.
Einen ähnlichen Entwicklungsprozess durchläuft die Software Bildsprache LiveLab, kurz BiLL, und befindet sich aktuell in einem frühen Stadium der Entwicklung. Das Anwendungsgebiet welches dieses Programm umfassen soll, liegt im Bereich der Computergrafik mit dem Fokus auf die Darstellung dialogorientierter dreidimensionaler Welten. Damit stehen zum einen der technische Erzeugungsprozess, von den Daten bis zur Renderausgabe, zum anderen das Ergebnis und dessen Aufnahme durch den Betrachter im Mittelpunkt. Durch Analyse und Bewertung des Präsentationsbildes, bezüglich dessen Eignung als Interaktionsbild wie es von GROH definiert wird, sowie dem Wissen um die Beschaffenheit des Datenbildes, kann gezielt Einfluss auf die Entwicklung angepasster computergrafischer Verfahren genommen werden.

1.2 Zielstellung

Aus den einleitenden Erörterungen ergeben sich zwei wesentliche Schwerpunkte für diese Arbeit. Zum einen ist, aufbauend auf einem bestehenden Softwaresystem, eine Experimentierplattform zu realisieren, mit der sowohl dialogorientierte als auch wahrnehmungskonforme Abbildungen einer dreidimensionalen Welt erzeugt werden können. Dafür ist eine Architektur zu entwickeln, die es erlaubt, Erweiterungen in das bestehende Softwaresystem BiLL zu integrieren. Dabei soll die Untersuchung auf eine komponentenbasierte Umsetzung gerichtet sein, die einer Integration von Erweiterungen zur Erforschung des Nutzer–Bild–Dialoges dreidimensionaler Welten dienlich ist. Zum anderen ist eine Umsetzung des Algorithmus der erweiterten perspektivischen Korrektur (vgl. [Zavesky 07]) in die Bildsprache LiveLab Arbeitsumgebung, als eine erweiternde Softwareeinheit, zu realisieren. Die Verwirklichung des Verfahrens soll weiterhin in der Echtzeitumgebung analysiert und ausgewertet werden. Schwerpunkt der Explorationen ist dabei die Wahrnehmungskonformität im Kontext der Interfacegestaltung. Nachfolgend wird eine Übersicht über die Teilbereiche sowie die einzelnen Kapitel der Arbeit gegeben.

1.3 Gliederung

Die vorliegende Diplomarbeit behandelt in ihrer Gesamtheit mehrere Themenbereiche. Das sich diesem anschließende Kapitel Zwei beinhaltet die Vorstellung einige für die Arbeit relevante Begriffe sowie Einführungen in Sachverhalte und Strukturen. Insbesondere wird auf Begriffe der Softwaretechnologie (2.1) und der Computergrafik (2.2) näher eingegangen. Ferner werden Klassenbibliotheken, die eine Voraussetzung für die praktische Umsetzung der Arbeit sind, vorgestellt (2.3).
Nachfolgend werden in Kapitel Drei verwandte Arbeiten und Konzepte aufgezeigt. Hierbei werden Komponentenarchitekturen (3.1) (vgl. [Szyperski 02], [Schumacher 03], [Griffel 98]) und im Speziellen die Plug–in Architekturen (3.2) (vgl. [Marquardt und Völter 02]) vorgestellt. Ferner werden Verfahren für die Erstellung multiperspektivischer Abbildungen aufgezeigt (3.3) (vgl. [Agrawala et al. 2000], [König 05], [Zavesky 07]).
Darauf aufbauend wird in Kapitel Vier die Arbeitsumgebung BiLL einführend vorgestellt (4.2). Nach Darstellung des aktuellen Entwicklungsstandes werden Anforderungen an eine Komponentenarchitektur analysiert und evaluiert (4.3). Auf Basis der Ergebnisse dieser Untersuchungen wird ein Entwurf für eine Architektur der Arbeitsumgebung BiLL entwickelt (4.4). Abgeschlossen wird das Kapitel mit theoretischen Betrachtungen bezüglich der Integration einer Erweiterung zur Erzeugung von multiperspektivischen Abbildungen in die Arbeitsumgebung (4.5).
Im Kapitel Fünf erfolgt eine detaillierte Vorstellung der im Rahmen dieser Arbeit erreichten praktischen Ergebnisse im Bereich der Komponentenarchitektur (5.2) und der Erweiterungssoftware für BiLL (5.3).
Die Arbeit schließt in Kapitel Sechs mit einer Zusammenfassung des Inhaltes (6.1) sowie einem Fazit (6.2) und darüber hinaus einem Ausblick (6.3) auf mögliche auf diese Arbeit aufbauende Weiterentwicklungen. Ein Anhang mit Glossar (A), Abkürzungsverzeichnis (B), Literaturverzeichnis (C), Abbildungsverzeichnis (D) und Tabellenverzeichnis (E) sowie einem Verzeichnis der interaktiven Elemente (F) vervollständigen die Arbeit. Im Text sind Verweise auf die Zusatzinformationen eingefügt. Diese befinden sich auf einer CD-ROM, die dieser Arbeit beigefügt ist. Ersichtlich sind die Verknüpfungen durch ein am rechten Rand befindliches und an dieser Stelle exemplarisch eingefügtes Symbol.


Zusatz 0: Verweis

2. Grundlagen

Bei einer umfassenden Betrachtung des Themenbereiches virtueller Welten und deren Kognition durch den Betrachter werden verschiedene Wissenschaftsfelder einbezogen. Hauptbereiche dabei sind die Softwaretechnologie, Computergrafik sowie die Bildsprache. Die beiden letztgenannten Sachverhalte haben im Zusammenhang mit dieser Arbeit eine große Schnittmenge und werden infolgedessen in diesem Kapitel gemeinschaftlich im Abschnitt Begriffe der Computergrafik betrachtet.
Nachfolgend werden zum besseren Verständnis der Arbeit relevante Sachverhalte vorgestellt und erläutert. Im Abschnitt 2.1 werden Zusammenhänge der Softwaretechnologie dargelegt und in 2.2 stehen Begriffe im Kontext der Computergrafik im Mittelpunkt. Ferner werden für die Arbeit essentielle Klassenbibliotheken vorgestellt (2.3).

2.1 Begriffe der Softwaretechnologie

Die Schnittstelle

Software– oder Datenschnittstellen sind logische Berührungspunkte zum Verbinden einzelner Softwarekomponenten (Module) in einem Softwaresystem. Sie definieren den Austausch von Kommandos und Daten zwischen verschiedenen Prozessen und Komponenten. Dabei unterscheidet man Softwareschnittstellen zum Zugriff auf Systemroutinen, zur Kommunikation zwischen Prozessen und zum Verbinden von Softwarekomponenten eines einzelnen Programms beziehungsweise für einen programmübergreifenden Zusammenschluss. Die Softwaremodule setzen sich dabei widerum aus Daten und Operationen zusammen, die eine in sich geschlossene Aufgabe reali-sieren.
Zur Beschreibung einer Schnittstelle wird im Kontrakt die Semantik der einzelnen Funktionen festgelegt, und somit eine Kopplung zwischen dem Anbieter und dem Verwender manifestiert. Dabei wird im Allgemeinen eine Menge von Regeln zu Grunde gelegt, die in der Schnittstellenbeschreibung zusammengefasst sind. In dieser werden die erlaubten wechselseitigen Annahmen definiert auf die sich der jeweilige Kontraktpartner verlassen kann. Über diese Definition ist ein kontrollierter Zugriff auf die hinter der Schnittstelle befindliche Datenstruktur möglich.
(vgl. [Gamma et al. 04], [Parnas 72])

Das Entwurfsmuster

Vorgefertigte Lösungsvorschriften zu wiederkehrenden Problemen in einem eindeutigen Kontext werden als Entwurfsmuster bezeichnet. Sie stellen eine allgemeine parametrierbare Lösung für typische Entwurfsprobleme bereit. Derartige Muster sind

sowohl allgemeiner Art als auch in spezifischen Anwendungsbereiche einsetzbar. Für das Gebiet der Softwaretechnologie sind Entwurfsmuster Beschreibungen zusammenarbeitender Objekte und Klassen, die angepasst sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.
(vgl. [Gamma et al. 04])

Der Proxy

Der Proxy oder auch Stellvertreter genannt, ist ein Entwurfsmuster aus dem Bereich der Softwaretechnologie und gehört zur Kategorie der Strukturmuster. Dieses wird verwendet, um die Kontrolle über ein Objekt auf ein vorgelagertes Stellvertreterobjekt zu übertragen. Das Entwurfsmuster kann eingesetzt werden, wenn ein reales Objekt zu Beginn einer Ausführung noch nicht zur Verfügung steht oder während der Laufzeit ausgetauscht werden muss.
(vgl. [Gamma et al. 04])

Die Vererbung

In der objektorientierten Programmierung ist die Vererbung eine Vorgehensweise, neue Klassen unter Verwendung bereits Bestehender zu erzeugen. Dabei übernimmt die neu erzeugte Klasse die Merkmale der Vorhandenen und ergänzt diese gegebenenfalls um neue Bestandteile. Die Übernahme der Eigenschaften und Funktionen von der existenten Klasse wird als Vererbung bezeichnet. Dieser strukturelle Aufbau bietet sich vor allem bei konzeptionell aufeinander aufbauenden Klassen an.
(vgl. [Middendorf et al. 02]).

Der Polymorphismus

Der Begriff bedeutet generell Vielgestaltigkeit. Im Bereich der objektorientierten Programmierung skizziert er ein Konzept der Vererbung, das es ermöglicht Methoden oder Variablen mit gleichem Namen für unterschiedliche Unterklassen zu implementieren. Dadurch können gleichnamige Operationen in verschiedenen Unterklassen inhomogenes Verhalten aufweisen. Ein Variablenwert oder die Umsetzung einer Methode steht aufgrund dessen erst zur Laufzeit fest und wird vom ausführenden Objekt bestimmt.
(vgl. [Middendorf et al. 02])

Die Modularisierung

Modularisierung ist eine wichtige Technik im Entwicklungsprozess von Software und definiert die Zerlegung eines Systems in einzelne Module. Bei der Realisierung von modularen Systemen ist das Entwurfsprinzip des Information Hiding wesentlich. Dabei werden Informationen eines Softwaresegments verdeckt beziehungsweise gesperrt und stattdessen kontrollierte Vorgehensweisen zur Verfügung gestellt. Über diese ist der Zugriff auf die wesentlichen Informationen der geschützten Softwareeinheit möglich.

Dadurch kann sichergestellt werden, dass die Verbindung und Kommunikation zwischen Modulen übersichtlich und regelkonform abläuft. Die Datenkapselung ist die am weitesten verbreitete und zugleich wichtigste Form des Information Hiding. Dabei wird die konkrete Implementierung einer Datenstruktur von deren sichtbaren Eigenschaften getrennt. Das Resultat dieses Vorgangs nennt man abstrakte Datenstruktur. Die Datenstruktur selbst wird verborgen und der Zugriff erfolgt mittels einer Schnittstelle.
(vgl. [Parnas 72])

2.2 Begriffe der Computergrafik

Das Koordinatensystem

Das Koordinatensystem ist ein mathematisch–geometrisches Ordnungssystem mit Hilfe dessen eine Positionsbestimmung im Raum erfolgt. Diese wird im gewählten Koordinatensystem durch Angabe von den Koordinaten, eindeutig bestimmt. Das System ist durch eine Basis des Vektorraums und einen Koordinatenursprung, in dem alle Koordinaten den Wert Null haben, festgelegt. Im Raum wird ferner zwischen rechts- und linkshändigen Koordinatensystemen unterschieden, wobei gewöhnlich Rechtshändige den positiven Drehsinn bezeichnen. Ein äquivalentes System liegt der Grafikschnittstelle OpenGL™ zugrunde und wird in dieser Arbeit referenziert. Die Ausgangsposition in einem OpenGL Koordinatensystem ist der Koordinatenursprung mit einer Blickrichtung entlang der negativen Z–Achse (Abbildung 2.1).
(vgl. [Foley et al. 94], [Shreiner 05])

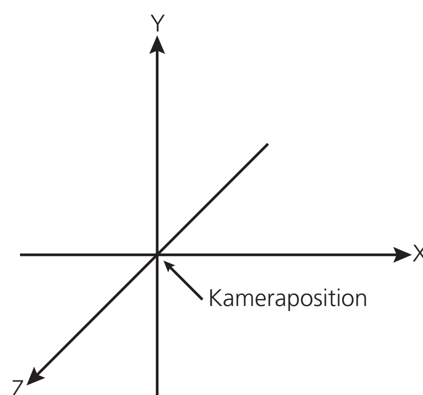


Abbildung 2.1: dreidimensionales Koordinatensystem

Die Projektion

Die geometrische Projektion beschreibt das Abbildungsverfahren, welches dreidimensionale Punkte auf eine festgelegte Ebene abbildet. Die für die Computergrafik bedeutendsten Projektionen sind die Parallelprojektion und die Zentralprojektion. Bei der Parallelprojektion wird das Abbild eines Punktes des Raumes an der Stelle dargestellt,

an dem der zur Projektionsrichtung parallel verlaufende und durch den Ausgangspunkt gehende Projektionsstrahl die Projektionsebene schneidet. Im Gegensatz dazu treffen sich bei der Zentralprojektion alle Projektionsstrahlen in einem Punkt. Die Bildpunkte ergeben sich aus den Schnittpunkten der Projektionsstrahlen mit der Bildebene.
(vgl. [Pareigis 90])

Der Sichtkörper

Der Sichtkörper (Abbildung 2.2) beschreibt in einem computergrafischen Kameramodell den Teil des Raumes, der mittels Projektion auf eine zweidimensionale Fläche abgebildet wird. Die Sichtstrahlen, ausgehend von der Betrachterposition, spannen den Sichtkörper auf. Dieser wird durch die vordere sowie die hintere Bildraumbegrenzung eingefasst. Zusammen bilden die Begrenzungen die Form eines Pyramidenstumpfes. Ausschließlich die Anteile der dreidimensionalen Szene die sich innerhalb des Sichtkörpers befinden werden gerendert und visualisiert.
(vgl. [Shreiner 05], [Franke et al. 06])

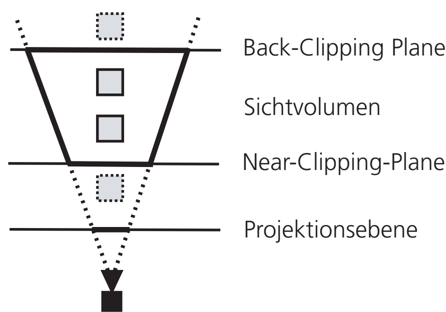


Abbildung 2.2: schematische Darstellung eines Sichtkörpers (nach [Angel 1990])

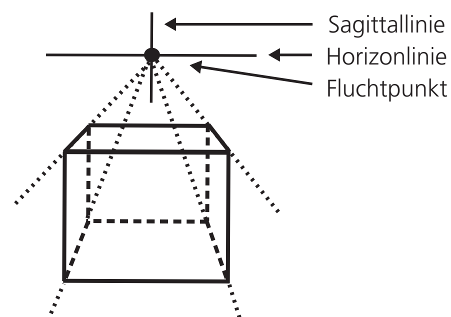


Abbildung 2.3: Die geometrische Mitte (nach [Franke et al. 06])

Die Geometrische Mitte

Nach [Franke et al. 06] beschreibt die geometrische Mitte den Punkt eines zentralperspektivischen Abbildes einer dreidimensionalen Szene, in dem alle orthogonal zur Bildebene verlaufenden Objektgeraden sich in ihrer Verlängerung in einem Punkt schneiden. Dieser Punkt ist der Schnittpunkt von der optischen Achse und der Abbildungsebene und markiert den Zusammenlauf der Horizont– und Sagittallinie des Bildes. Die geometrische Mitte kann dabei ein Fluchtpunkt des Bildes sein (Abbildung 2.3).

Der Szenengraph

Bei der Beschreibung virtueller Welten ist eine ordnende Struktur zur Organisation der Bestandteile der Welt unumgänglich. Der Szenengraph stellt einen Ansatz für diese Aufgabe dar. Die folgenden Abschnitte werden das Konzept und die Funktionsweise

des Szenegraphens einführend erklären und darüber hinaus den Einsatz im Bereich der Computergrafik aufzeigen (vgl. [Wernecke 94], [Seewald 04]).
Ein Szenengraph ist eine objektorientierte Datenstruktur, die häufig Anwendung im Bereich der Computergrafik findet. Dabei dient dieser, mit Hilfe seiner hierarchischen Ordnung von Objekten in 3D–Welten, als abstrakte Darstellungsform der Gesamtszene. Der Szenengraph ist ein gerichteter azyklischer Graph und formiert sich aus Knoten, Knotenverbindungen und Knotenkomponenten. Letztgenannte sind die dem Knoten angegliederten Daten. Die Struktur eines Szenengraphens setzt sich aus einem Ursprung und einer endlichen Menge an Kindknoten zusammen. Diese können ihrerseits Kindknoten besitzen und bilden damit zugleich den Ausgangspunkt eines Subgraphens der Szene. Die Objekte einer Szene sowie deren Attribute werden durch Knoten im Graphen repräsentiert. Dabei wird die Gesamtszene durch die Anordnung und die Relationen der Knoten zueinander beschrieben. Darüber hinaus beinhaltet der Graph weitere Knoten die der Organisation sowie der Positionierung und der Darstellung der Objekte dienen. Der hierarchische Aufbau einer Szene mittels eines Graphens zielt nicht ausschließlich auf die Beibehaltung einer Ordnung der 3D–Szene. Die Verbindungen von Knoten in Mutter–Kind–Beziehungen weisen verschiedene Bedeutungen auf. Eine Wesentliche ist die Vererbung von Transformationen an die Kindknoten, sodass sich Änderungen an den Attributen eines Objektes zugleich auf all seine untergeordneten Objekte auswirken.

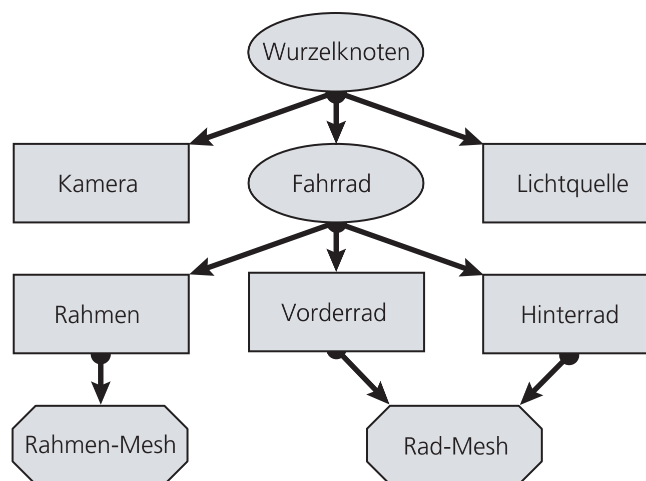


Abbildung 2.4: Szenengraph zur Visualisierung eines Fahrrads [Seewald 04]

Die Abbildung 2.4 zeigt einen einfachen Szenengraphen zur Beschreibung eines Fahrrades. Der Gruppierungsknoten „Wurzelknoten" bildet den Quellknoten der Szene. Dessen Kindknoten sind zum einen die Blattknoten Lichtquelle und Kamera und zum anderen ein Gruppierungsknoten, der zugleich Wurzel des Teilgraphens des Fahrrades ist. Eine Transformation dieses Gruppenknotens würde zu einer Veränderung bei allen unterordneten Knoten führen. Dem Fahrradknoten sind drei Knoten als dessen Kindknoten angegliedert. Jeder dieser verweist widerum auf Geometriedaten

(Mesh), die zur Darstellung der Objekte in der 3D-Welt benötigt werden. Unter der Annahme, dass dem Vorder– und Hinterrad die gleiche Geometrie zu Grunde liegt, können beide Knoten auf ein und denselben Geometrieknoten zugreifen. Dies ist möglich, obwohl die Positionen von Vorder- und Hinterrad different sind. Dies ist ressourcenschonend und zugleich performanter, weil das Rad–Mesh nur einmal im Speicher gehalten werden muss.
Jeder Szenengraph kann Teil eines anderen Graphens werden. Somit ist die Kombination von Graphen zu beliebig komplexen Strukturen denkbar. Jeder Subgraph stellt die enthaltenen Knoten in einem Node Kit bereit. Damit ist die Komposition einer Szene aus einzelnen Subgraphen umsetzbar. Zur Verdeutlichung ist in der folgenden Veranschaulichung, der in Abbildung 2.4 dargestellte Szenengraph erweitert worden. Die Lichtquelle bewirkt die Beleuchtung von allen untergeordneten Graphen. Somit erfolgt eine Manipulation aller Subgraphen.

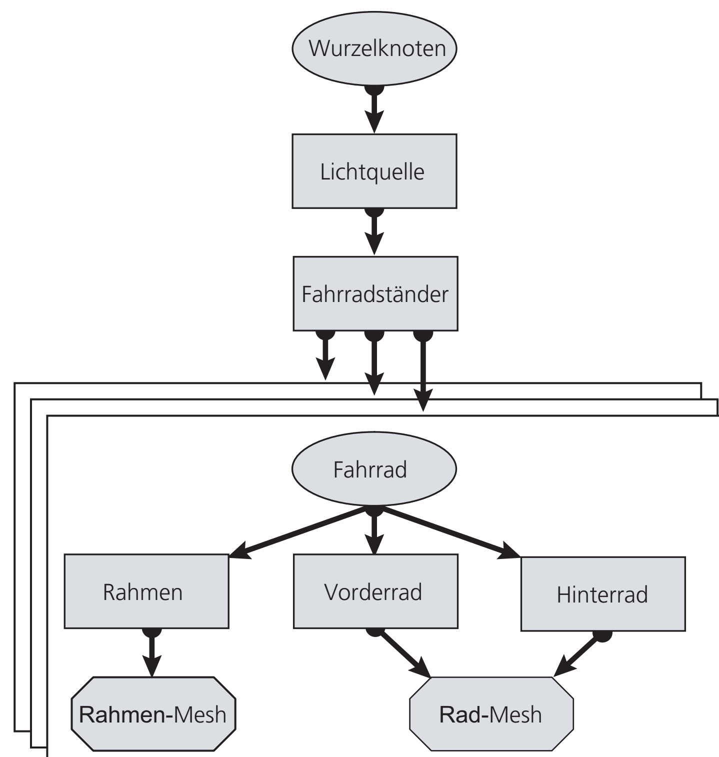


Abbildung 2.5 Subgraphen innerhalb eines Szenengraphs [Seewald 04]

Die allgemeine Grundlage für das Arbeiten auf Szenengraphen bildet die Traversierung und damit das Durchlaufen des Graphens in einer bestimmten Reihenfolge. In diesem Kontext bilden „breadth–first–traversal“ und „depth–first–traversal“ die bedeutendsten Traversierungsarten. Im Zuge des Durchlaufens der Knoten im Graphen, können auf diesen Operationen ausgeführt werden. Die Kombination aus der Traversierung und einer pro Knoten ausgeführten Operation bildet eine semantische Einheit, den Besucher (Visitor). Der Besucher wird von der Wurzel aus, abwärts durch den Baum gereicht und führt auf jedem Knoten seine spezifische Aktion aus. Das Prinzip des

Visitor ermöglicht damit einhergehend die Trennung von struktureller Datenorganisation und Funktionalität. Klassische Probleme der Computergrafik, wie Animation oder Kollisionsdetektion, lassen sich durch die Umsetzung spezieller Besucher realisieren.
Der Gebrauch eines Szenengraphens im Kontext einer 3D-Anwendung stellt zusammen mit einer 3D–Engine ein eigenes Subsystem dar. Dabei setzt es auf eine bereits verfügbare Low–Level–3D–API auf und abstrahiert diese weiter. Der Szenengraph kann dabei die zugrunde liegende API wegkapseln oder gegenteilig den direkten Zugriff auf diese erlauben.

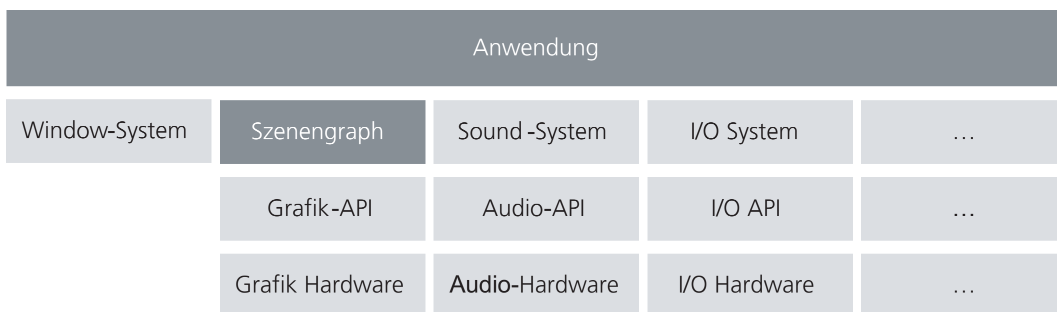


Abbildung 2.6: Szenengraph im Kontext einer Anwendung [Seewald 04]

Neben der Darstellung und Beschreibung von visuellen Szenen erlaubt das Konzept des Szenengraphens die Integration anderer Aspekte wie Geräusche oder Physik in die 3D-Welt (Abbildung 2.6). Ferner besteht die Möglichkeit der Eingliederung von Programmlogik. Diese wird über spezielle Knoten in den Szenengraphen eingefügt und somit bereitgestellt. Dadurch können beispielsweise mehrere Verhaltensoptionen in verschiedenen Zweigen des Graphen festgelegt werden. Bei der Traversierung wird daraufhin lediglich das Segment des Graphen mit dem zutreffenden Verhalten ausgeführt.
Mit diesen Einblicken wird gezeigt, dass die Einsatzmöglichkeiten über die rein visuelle Darstellung hinausgehen. Ein Szenengraph verkörpert daher mehr eine flexible Beschreibung aller Aspekte der virtuellen Welt.

Die Rendering Pipeline

Das abstrakte Modell der Rendering Pipeline wie es in den Arbeiten von FRANKE und ANGEL beschrieben wird, zeigt den Verfahrensablauf vom 3D–Modell zur Grafik auf dem Ausgabegerät. In der nachfolgenden Abbildung werden die Hauptsegmente einer Rendering Pipeline aufgezeigt. Die Schritte können chronologisch abweichen, weil es dem Programmierer möglich ist in den Prozess einzugreifen.

Abbildung 2.7: Rendering Pipeline [Franke et al. 05a]

Im ersten Schritt, der Modelltransformation, werden die Objekte der Szene in das Weltkoordinatensystem transformiert. Dies erfolgt über verschiedene Matrizenoperationen. Die Projektionstransformation schließt sich an den diesen Schritt an und konvertiert die ermittelten Weltkoordinaten in die Kamerakoordinaten. Dadurch wird die Szenerie auf den Betrachter und die Projektionsebene ausgerichtet. Die Darstellung auf der virtuellen Abbildungsfläche und damit die zweidimensionale Ansicht der Szene ist vom gewählten Projektionsverfahren abhängig. Der dritte Schritt, das Clipping, entfernt Objekte oder Teilbereiche dieser, wenn sie sich nicht innerhalb, des von der Kamera festgelegten Sichtkörpers, befinden. Damit wird eine Performancesteigerung erreicht, um im letzten Schritt die Rasterisierung durchzuführen. Die Darstellung wird dabei in Pixel gerastert und über den Bildspeicher auf dem Ausgabegerät visualisiert.

Das Postprocessing

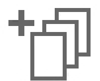
Postprocessing ist eine Technik, die erst mit der Einführung von Pixelshader möglich wurde. Hierbei wird die Abbildung einer Szene nach dem Rendern nochmals auf Pixelebene bearbeitet und verändert. Zunächst erfolgt das Rendern der 3D–Szene in eine Textur und nach Beendigung des Vorgangs werden beim Postprocessing zusätzliche Informationen, wie zum Beispiel die Tiefeninformationen, in weiteren Texturen der Abbildung hinzugefügt und über die ursprüngliche Textur gelegt. Dazu werden diese vom Shader auf ein bildschirmfüllendes Rechteck projiziert und dargestellt. Hierdurch ergeben sich nahezu unbegrenzte und effiziente Möglichkeiten der Nachbearbeitung von Bildern.

2.3 Klassenbibliotheken

OpenSceneGraph™ (OSG)

OpenSceneGraph™ ist ein plattformübergreifendes Toolkit für die Entwicklung von Grafikanwendungen wie beispielsweise Spiele und Flugsimulatoren (vgl. [OSG]). Die Technologie basiert auf einem objektorientierten Framework und setzt das Konzept eines wie in 2.2 beschriebenen Szenengraphens, zur Visualisierung und Manipulation von virtuellen Welten um. OSG ist in der Programmiersprache C++ entwickelt und baut auf die Grafikschnittstelle OpenGL™ auf.
Die Abstraktion eines Szenengraphens auf Basis von OpenGL Befehlen wird durch ein Node Kit umgesetzt. Dieses stellt alle in OSG vorliegenden Knoten zur Verfügung und realisiert damit eine Abstraktionsebene oberhalb der Grafik–API. Die im Node Kit enthaltenen Knotenkomponenten reflektieren spezifische OpenGL Funktionalitäten. Dadurch stehen die OSG Knotenelemente in direkter Verbindung zu OpenGL, bei gleichzeitiger Verdeckung der Schnittstelle. Auf diese Weise bildet zum Beispiel ein Geode-Knoten als Teil des Node Kits einen Geometrieknoten einer Szene ab. Dieser Knoten zeigt sich als Container für *osg::Drawables*, eine Gruppe von Komponenten, welche OpenGL Zeichenoperationen abbilden.

Der Renderprozess von OpenSceneGraph™ setzt sich aus den drei Phasen Update, Cull und Draw zusammen. Diese Abschnitte werden mindestens einmal pro Frame ausgeführt. Ausgangspunkt ist die Update–Phase, in der eine Aktualisierung des Graphens erfolgt. Dabei werden unter anderem die verfügbaren Animationsabläufe erneuert und benutzerdefinierte Manipulationen in den Szenengraph integriert. Im nächsten Schritt, der Cull–Phase, werden Sichtbarkeitsdefinitionen und Vorbereitungen für den eigentlichen Zeichenvorgang der Szene durchgeführt. Diese Festlegungen sind spezifisch für die aktuelle Sicht auf die Szene. Daraufhin wird der Graph mittels eines Cull–Visitors durchlaufen. Dieser übermittelt jedem Knoten individuelle Informationen bezüglich des Renderkontextes. Beispielsweise könnten LoD–Knoten Auskunft über den Abstand zum Betrachter oder die Pixelgröße auf dem Ausgabegerät erhalten, um auf Basis dieser Daten einen Detailgrad festzulegen. Bei der Traversierung wird die Sichtbarkeit des jeweiligen Knotens geprüft und gegebenenfalls in die Renderwarteschlange eingereiht. In der letzten Phase des Vorgangs schließt sich die Sortierung sowie das Rendern der Komponenten, die sich in der Warteschlange befinden, an. Cull– und Draw–Vorgang besitzen während ihrer Tätigkeit nur Lesezugriff auf den Graphen wodurch mehrere dieser Vorgänge gleichzeitig ablaufen und damit mehrere Ansichten aus dem Graphen erzeugt werden können.



Zusatz 1: OSG Referenz

Open Producer

Die Bibliothek Open Producer ist ein auf C++ und OpenGL™ basierendes Toolkit mit der Fokussierung auf die Kamerasteuerung in dreidimensionalen Welten. Die Umsetzung abstrahiert das Prinzip einer realen Kamera (Abbildung 2.8). Die virtuelle Szene ist in einem dreidimensionalen kartesischen Koordinatensystem definiert. Die in der 3D-Welt befindlichen Objekte werden mit Hilfe der Projektionsmatrix, die seitens der abstrahierten Kameralinse erzeugt wird, aus dem 3D–Raum als zweidimensionale Abbildung angezeigt. Es wird der Szenenausschnitt visualisiert, der sich innerhalb des Sichtkörpers der Kamera befindet.

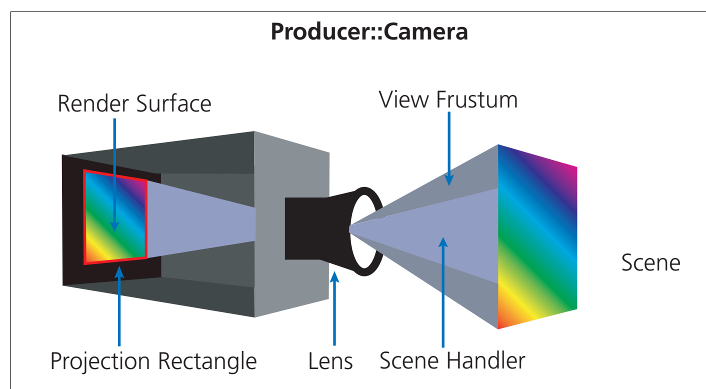
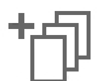


Abbildung 2.8: Funktionsprinzip der Open Producer Kamera (nach [Open Producer])

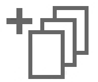


Zusatz 2: Open Producer Referenz

FLTK©

Fast Light Toolkit oder auch FLTK© ist ein Framework für die Erzeugung von Benutzeroberflächen. Im Bereich der elektronischen Datenverarbeitung ist dies eine Sammlung von Bibliotheken, Klassen und Schnittstellen, die eine Erzeugung von graphischen Bedienelementen innerhalb einer Anwendung vereinfacht. Dazu werden Standardfunktionen zur Verfügung gestellt, die zur Erstellung von Benutzerinteraktionselementen verwendet werden. Die enthaltenen Klassen wie beispielsweise Fenster, Menüs, Controls und die Darstellung von Bildern decken den GUI-Bereich ab.

FLTK© ist ein plattformunabhängiges Window Toolkit für Microsoft® Windows, Unix™/ Linux™ und Mac OS X™ (vgl. [FLTK]). Das Framework ist als Nachfolger von XForms implementiert und unterliegt der LGPL Lizenz und kann damit kostenfrei zur kommerziellen als auch nicht kommerziellen Softwareentwicklung verwendet werden (vgl. [LGPL]). Mithilfe von FLTK© werden statisch oder auch dynamisch gelinkte Einzelanwendungen erstellt. Dabei stellt das Framework ausschließlich die Funktionalität für Bedienoberflächen bereit, unterstützt jedoch auch Grafikprogrammierung mit OpenGL Anbindung.



Zusatz 3: FLTK Referenz

3. Verwandte Arbeiten

Der in Kapitel Eins beschriebene Forschungsbereich vereint Segmente aus verschiedenen Fachgebieten. Die im folgenden Kapitel vorgestellten Arbeiten sind auf einzelne Themengebiete fokussiert und bilden Ansatzpunkte für die einführend beschriebene Thematik. Dabei wird auf Grundlagen der grafischen Datenverarbeitung sowie auf Aspekte der Softwareentwicklung eingegangen. Die Arbeiten von GRIFFEL (vgl. [Griffel 98]) und SZYPERSKI (vgl. [Szyperski 02]) umfassen das Gebiet der Componentware im Kontext der Softwaretechnologie. Ferner werden in diesen Arbeiten Komponentenarchitekturen betrachtet (3.1). Folgend werden aufbauend auf den Texten von SCHUMACHER (vgl. [Schumacher 03]) und MARQUARDT/VÖLTER (vgl. [Marquardt und Völter 02]) Plug–in Architekturen sowie Schnittstellendefinitionen zur Erweiterbarkeit von Software vorgestellt (3.2). In Abschnitt 3.3 werden Bereiche der Bildsprache in den Fokus gerückt. Diese Betrachtungen stützen sich auf die Arbeiten von AGRAWALA (vgl. [Agrawala et al. 2000]), ZAVESKY (vgl. [Zavesky 07]) und KÖNIG (vgl. [König 05]).

3.1 Die komponentenbasierte Softwareentwicklung

Wie einleitend dargelegt ist die Softwareentwicklung ein evolutionärer Entwicklungsgang. GRIFFEL zeigt, dass dieser Prozess kontinuierlich an Komplexität zugenommen hat. Folglich ist eine gesteigerte Fehleranfälligkeit und daraus resultierend ein höherer Wartungsaufwand während des gesamten Prozesses zu verzeichnen. Wiederkehrende Problemstellungen werden bei der Softwareentwicklung häufig abermals gelöst, anstatt die bereits bestehenden Lösungen wiederholt aufzugreifen. Eine mögliche Lösung des Konfliktes ist die komponentenbasierte Softwareentwicklung. Dieser Ansatz folgt dem Ziel der Objektorientierung insoweit, bereits bewältigte Probleme nicht erneut zu erschließen. Einen Lösungsansatz bieten objektorientierte Techniken wie Kapselung und Polymorphismus Diese ermöglichen eine flexible Wiederverwendung von Komponente. Derzeit existiert keine allgemeingültige Definition des Begriffes Softwarekomponente. Vielmehr bestehen diesbezüglich differenzierte Ansichten. Im Folgenden wird daher auf die Abgrenzung des Begriffes Komponente von Objekt, Klasse, Typ und Modul nach SZYPERSKI eingegangen.
Ein Hauptmerkmal für Softwarekomponenten ist danach dessen Kompositionsfähigkeit. Folglich ist eine Komponente vereinfacht betrachtet ein in sich geschlossener, komponierbarer, wiederverwendbarer Softwarebaustein, der über eine definierte syntaktische Schnittstelle und eine Semantik verfügt. GRIFFEL beschreibt ebenfalls den Terminus Komponente und erweitert die vorangegangene Ausführung dahingehend, dass eine Softwarekomponente keine Klasse im Quelltext darstellt, sondern eine ausführbare Softwareeinheit. Sowohl SZYPERSKI als auch GRIFFEL knüpfen an den Status

einer Komponente Anforderungen die diese erfüllen muss. Für ein Softwarebaustein, der Teil einer Componentware ist, sind nach SZYPERSKI drei Punkte charakteristisch:

A component is a unit of independent deployment.

Aus dieser Eigenschaft folgt, dass eine Komponente von ihrer Umgebung und anderen Komponenten abgeschirmt sein muss und ihre eigenen Eigenschaften kapselt. Außerdem wird eine Komponente niemals teilweise installiert, weil es sich bei ihr um eine Einheit handelt.

A component is a unit of third-party composition.

Der „third-party" Aspekt verweist darauf, dass in der Regel die Entwickler der Anwendung und der Komponenten sich nicht in einer Person vereinigen. Damit Softwarebausteine durch Dritte entwickelt werden können, sei es Voraussetzung, dass diese in sich geschlossen sind. Zusätzlich sind für eine Umsetzung klare Spezifikationen bezüglich der Anforderungen und der Leistungen von Komponenten nötig.

A component has no persistent state.

Eine Komponente kann nach SZYPERSKI nicht von einer eigenen Kopie unterschieden werden, weil sie keinen Status hat. Aufgrund dessen sind mehrere Kopien eines Softwarebausteins wenig zweckmäßig. Dabei kann jedoch nicht gleichzeitig ausgeschlossen werden, dass eine Komponente zustandsabhängige Ergebnisse liefert. Daher muss gleichwohl zwischen der Komponente und zum Beispiel den Daten auf denen die Komponente arbeitet, unterschieden werden.
SZYPERSKI greift abschließend eine klare Definition auf, die auf der „European Conference on Object-Oriented Programming" im Jahr 1996 unter seiner Mitwirkung entstanden ist und die zuvor genannten Eigenschaften weiter verdeutlicht:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

In dieser Definition verbirgt sich hinter dem Begriff der Kompositionseinheit die Möglichkeit, eine Komponente mit einer anderen zu verknüpfen. Die Zusammenführung und die daraus resultierende Konstellation bildet die Konfiguration eines Systems. Dadurch ist eine flexible Anpassung einer Software an ein signifikantes Problem möglich. Eine spezifizierte Schnittstelle ist notwendig um einem Entwickler die Funktionalität einer Komponente zur Verfügung zu stellen. Weiterhin sind die in der

Definition genannten Kontextabhängigkeiten, Bindungen einer Komponente an ihre Einsatzumgebung. Diese treten auf, wenn das Modul nur auf bestimmten Plattformen funktionsfähig oder an andere Softwarebausteine oder –produkte gebunden ist. Ein unabhängiger Einsatz von Softwarekomponenten erhöht, dem Aspekt entgegenwirkend, die Möglichkeit einer Wiederverwendung. Diese ermöglicht, dass die Komponentenentwicklung nicht direkt mit der Entwicklung einer speziellen Anwendung in Verbindung gebracht wird, sondern aus einer umfassenderen Sichtweise heraus konzipiert werden kann, um die Einsatzmöglichkeit in verschiedenen Softwareprojekten zu ermöglichen. Die Bereitstellung einer Komponente erfolgt über die Integration in eine Anwendung, die Kommunikation über Schnittstellen und die Verwaltung durch eine entsprechende Managereinheit.

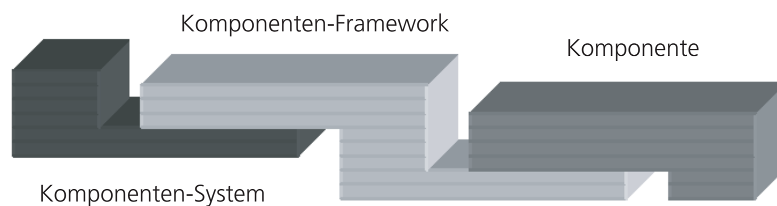


Abbildung 3.1: Skizze einer allgemeinen Komponentenarchitektur
(nach [Szyperski 02], S.277)

Die Abbildung 3.1 verdeutlicht, dass die Ausarbeitung eines verwendbaren Komponentenparadigmas aus mindestens zwei Ebenen bestehen muss. Zum einen aus der Komponente selbst, welche sich wie von SZYPERSKI beschrieben darstellt und zum anderen deren kooperativen und koordinierten Verbindungsmethoden und –mechanismen. Ferner kann eine Systemkomponente bereitgestellt werden, durch die eine Anwendungskonfiguration vorgenommen wird. Die Zusammenführung von Komponenten in einer Infrastruktur wird als Komponentenarchitektur bezeichnet (vgl. [Heinemann/Councill]). Diese bildet den Rahmen für die Entwicklung, Integration und Anwendung von Komponenten. Ferner wird die Struktur jeder Komponente die zu einem Komponentenmodell konform sein soll, festgelegt. Die Vorgaben können beispielsweise Implementierungssprachen, obligatorische Schnittstellen oder andere Einschränkungen sein. Weiterhin legt das Modell die Struktur der Verknüpfung von Komponenten fest. Dies umfasst die Deklaration von Abhängigkeiten zwischen Softwarebausteinen oder Schnittstellen, das Auffinden von Diensten oder Modulen sowie das Zusammenstellen einer Anwendung aus den einzelnen Softwarebausteinen.

JavaBeans verwirklicht beispielsweise die Verwendung eines derartigen Komponentenmodells (vgl. [JavaBeans]). Von der Firma Sun® Microsystems entwickelt, ist es eine auf Java basierende Architektur für die Definition und Wiederverwendung von Komponenten zur Softwareentwicklung. Beans sind insbesondere für den Einsatz in computergraphischen Anwendungsentwicklungen konzipiert. Dabei bilden einzelne Beans die Basisbausteine aus denen zusammengesetzte größere Softwareteile erstellt

werden können. Für die Zusammenführung einzelner Komponenten, müssen diese manifeste Schnittstellenanforderungen erfüllen sowie Namenskonventionen einhalten. Dadurch ist ein störungsfreies Zusammenwirken der einzelnen Softwareeinheiten gegeben.

3.2 Die Plug–in Architektur

In diesem Abschnitt fokussieren sich die Betrachtungen auf eine spezielle Form von Komponenten, die Plug–ins. Hierfür wird zunächst der Begriff selbst umrissen (3.2.1). Anschließend wird auf das Zusammenwirken von der Hauptanwendung und den Plug–ins in einer Architektur eingegangen (3.2.2). Dabei wird insbesondere die Kommunikationsverbindungen, die in einem Vertrag zwischen den einzelnen Softwarebausteinen festgelegt sind, aufgezeigt.

3.2.1 Der Begriff Plug–in

Der Begriff Plug–in wird häufig in aktuellen Softwareprodukten postuliert. Das eingängige Bild des „Einsteckens" von Software in ein bestehendes System ersetzt jedoch keine genaue Begriffsbestimmung. Darüber hinaus zeigt dieser Umstand, dass eine Definition des Begriffes Plug–in nicht feststeht. Eine Präzisierung des Terminus bieten MARQUARDT und VÖLTER.

> *Plug–ins sind Ergänzungen zu einer Applikation, die von dieser vorgesehen sind, die sie aber nicht selbst mitbringen kann oder will. Ein Plug–in deckt dabei eine fachliche Funktion so vollständig ab, dass keine andere Komponente des Systems etwas Spezifisches darüber wissen müsste. Für die dazu nötigen Schnittstellen ist die Applikation verantwortlich.*
> *([Marquardt und Völter 02], S.4)*

In dieser Begriffsbestimmung werden Plug–ins als Ergänzung angesehen, die ein bestehendes System um Funktionalitäten erweitern. Die Hostapplikation stellt zu diesem Zweck eine Schnittstelle bereit. Die Erweiterungen werden dabei als Softwarekomponenten betrachtet. Diese besitzen jedoch Eigenschaften die sie zu einer speziellen Form eines solchen Softwarebausteins werden lassen. MARQUARDT und VÖLTER beschreiben Plug–ins als unabhängige Bausteine, die für eine spezielle Anwendung entwickelt werden und auch nur in deren Kontext funktionsfähig sind. Der Grund dafür ist eine spezifische Nutzung der Daten die den Plug–ins von der Anwendung zur Verfügung gestellt werden. Sie fügen sich in das vorhandene Konzept und die Benutzeroberfläche des Programms ein. Abstufungen zwischen Plug–ins verschiedener Applikationen zeigen sich vor allem in der Implementierung der Schnittstelle aber auch im Freiheitsgrad, der den Plug–ins bezüglich der Kommunikationsmöglichkeiten zugestanden wird.

Die populärsten Anwendungen die sich Plug–ins zunutze machen sind Internet Browser wie Microsoft® Internet Explorer und Mozilla Firefox™. Beim Laden einer Seite erkennt der Browser automatisch wenn ein bestimmtes Plug–in benötigt wird und installiert es, falls dies notwendig ist, automatisch. Daraufhin übergibt der Browser die Kontrolle an das Plug–in. Dieses wird folgend ausgeführt und die Kontrolle nach Beendigung an die Anwendung zurückgegeben. Das Vorgehen erfolgt unsichtbar für den Benutzer und ohne dass die Anwendung dafür beendet werden muss. Die folgende Abbildung verdeutlicht die Plug–in Architektur eines Browsers mit der Verwaltung der Erweiterungen über eine Bibliothek. Diese wird bei gegebenem Erfordernis um das entsprechende Plug–in erweitert.

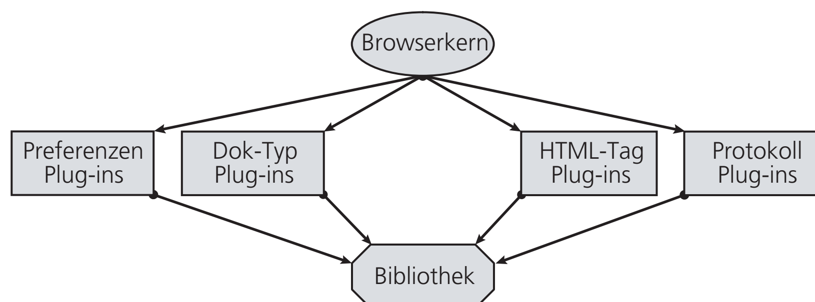


Abbildung 3.2: Plug–in Architektur eines Browsers (nach [Ruess 03])

3.2.2 Der Vertrag

Plug–ins oder genauer betrachtet, die verknüpften Bestandteile die ein Erweiterungsmodul bilden, sind mit der für diese vorgesehenen Anwendung über einen Vertrag gekoppelt. Dieser Kontrakt wird von Seiten der Hostapplikation festgesetzt und forciert eine Erweiterung zur Erfüllung definierter Anforderungen. Demgegenüber konkretisiert die Hauptanwendung Annahmen auf die sich ein Plug–in bei dessen Ausführung stützen kann. Dadurch besitzt die Erweiterung eine Eigenständigkeit, sodass es der Basisanwendung ermöglicht wird, lediglich die Definition der Integrationspunkte festzulegen, ohne dabei eine Kenntnis über die Aufgabe des Plug–ins zu benötigen. Das Hauptprogramm gibt zu Beginn der Ausführung einer Plug–in Funktionalität die Kontrolle des Arbeitsablaufes an die Erweiterung ab und übernimmt erst nach Beendigung der Aufgabe die Zuständigkeit wieder. Diese Kooperation bildet das Kernstück einer Plug–in Architektur. In den dafür festgelegten Vertragsbedingungen ist die Struktur der gesamten Anwendung enthalten. Somit können Voraussetzungen und Konventionen festgelegt werden, die für jedes Erweiterungsmodul gleichermaßen vorliegen. Dies ermöglicht einen geregelten Freiheitsgrad bei der Entwicklung erweiternder Softwarebausteine. Beispielsweise können somit verschiedene Techniken für die Erzeugung einer GUI oder der dahinter stehenden Funktionalität bereitgestellt oder festgelegt werden.

Eine Integration von Erweiterungen ist nur bei der Einhaltung der Vertragsbedingungen gewährleistet. Diese reglementieren eine Reihe von technischen Obliegen-

heiten. Zu diesen gehören Installation, Registrierung, Lebenszyklen der Plug–ins, Dienste und Prozessmodel, und nicht zuletzt die Gestaltung der grafischen Benutzerschnittstelle (GUI) – sofern dieses vorhanden ist. MARQUARDT und VÖLTER legen dazu vier Aspekte einer Vertragsbindung fest:

i. **Installation und Registrierung** legen fest wie die Hauptanwendung von der Existenz einer Erweiterung informiert und wie dieses geladen wird.
ii. **Lebenszyklen** können ein dynamisches Laden und Entladen sowie die Kardinalität von Plug–in Instanzen organisieren.
iii. **Dienste,** die dem Plug–in zur Verfügung stehen sind essentiell und umfassen die Bereiche Fehlerbehandlungen, Kommunikation zwischen Anwendung und Erweiterung sowie die Festlegung von Zugriffsrechten auf Bereiche der Applikation oder auch des Betriebssystems.
iv. **GUI–Gestaltung –** Diese bestimmt welche Bibliotheken ein Plug–in benutzen muss und inwieweit die Erweiterung die Benutzeroberfläche der Anwendung verändern darf.

In welchem Umfang die Hostapplikation und die Plug–ins miteinander fusionieren, hängt von der Komplexität der Schnittstelle ab. Die Verknüpfungspunkte zwischen beiden Softwareeinheiten sind im Vertrag geregelt und bestimmen inwieweit eine Applikation erweiterbar ist und wie eng die Angliederung der Erweiterungen an die Kernanwendung erfolgt.
Eine Plug–in Architektur trennt ein Gesamtsystem in einzelne Fachbereiche. Dabei ist die Grundanwendung auf Kernaufgaben und auf die Funktionalität für den Plug–in Zugriff beschränkt. Die zentralen Punkte sind im, von der Hauptanwendung festgelegten, Plug–in Vertrag fixiert. Dadurch kann mit Hilfe einer Plug–in Architektur eine Teilung von Verantwortlichkeiten vorgenommen und umgesetzt werden.

3.3 Die multiperspektivische Abbildung

In der Motivation der Arbeit wird auf die Darstellung dreidimensionaler Szenen und die perspektivischen Verzerrungen, die in der Zentralprojektion bei großem Kameraöffnungswinkel auftreten können, eingegangen. Der folgende Abschnitt stellt Ansätze und Möglichkeiten vor aus dem perfekten Systemraum auszubrechen und die damit verbundenen perspektivischen Verzerrungen zu eliminieren. Durch eine Beseitigung dieser können wahrnehmungsrealistische Szenendarstellungen von virtuellen Welten wie sie eingangs dieser Diplomarbeit beschrieben worden sind, erzeugt werden. Dazu wird auf die Schaffung von Binnenperspektiven und damit auf die Erzeugung von multiperspekti-vischen Abbildungen eingegangen. Dazu werden zwei Methoden vorgestellt. AGRAWALA, ZORIN und MUNZNER (vgl. [Agrawala et. al. 2000]) beschreiben ein Verfahren zur Erzeugung eines Systemraums, wie von PANOFSKY (vgl. [Panofsky 85])

dargelegt, unter Verwendung mehrerer Kameras (3.3.1). Nachfolgend wird die Vorgehensweise und Methodik eines Prinzips aufgezeigt, welches auf der Manipulation der Objektgeometrie, mittels Transformationsverfahren zur verzerrungsfreien Darstellung von Objekten, beruht (3.3.2).

3.3.1 Multiperspektivprojektion

Im Rahmen eines Projektes mit dem Ziel des multiperspektivischen Rendering untersuchen AGRAWALA, ZORIN und MUNZNER die Möglichkeit der Reduktion von Verzerrungen bei Weitwinkelprojektionen. Jedem Objekt, welches einer Binnenperspektive unterworfen werden soll, wird dazu eine individuelle Kamera zugeordnet. Dadurch wird ein Szenario mit einer Weitwinkelkamera und den Einzelkameras für die jeweilige Binnenperspektive der einzelnen Objekte erzeugt. Mit Hilfe eines speziellen Rendering Algorithmus werden die verschiedenen Perspektiven in einer Abbildung zusammengefügt. Dieser Ansatz bildet das Prinzip der Multiperspektive durch den Einsatz mehrerer Kameras.
Schwierigkeiten zeigen sich bei diesem Verfahren vor allem in der Kombination der durch die Kameras erzeugten Abbildungen in ein multiperspektivisches Bild und der damit verbundenen Bestimmung der Verdeckung der einzelnen Objekte der Szene. Dafür wurde ebenfalls ein Verfahren entwickelt, dass auf Basis von festgelegten Parametern und den verwendeten Kameras das Vereinen der einzelnen Objekte in einem Bild ermöglicht. Die nachfolgende Abbildung zeigt eine computergrafische Abbildung von Säulen in einer Monoperspektive. In dieser sind die perspektivischen Verzerrungen vor allem in den Randbereichen des Bildes erkennbar. Rechts neben dieser Visualisierung ist der Sichtkörper der Weitwinkelkamera schematisch dargestellt. In der darunter befindlichen Darstellung ist dieselbe Szene visualisiert. Jedoch sind den Säulen jeweils eigene Kameras zugeordnet und die Einzelbilder als Komposition zusammengeführt. Es ist eine deutliche Reduktion der perspektivischen Verzerrungen erkennbar. Dadurch entsteht eine wahrnehmungskonforme Abbildung der virtuellen Szene.

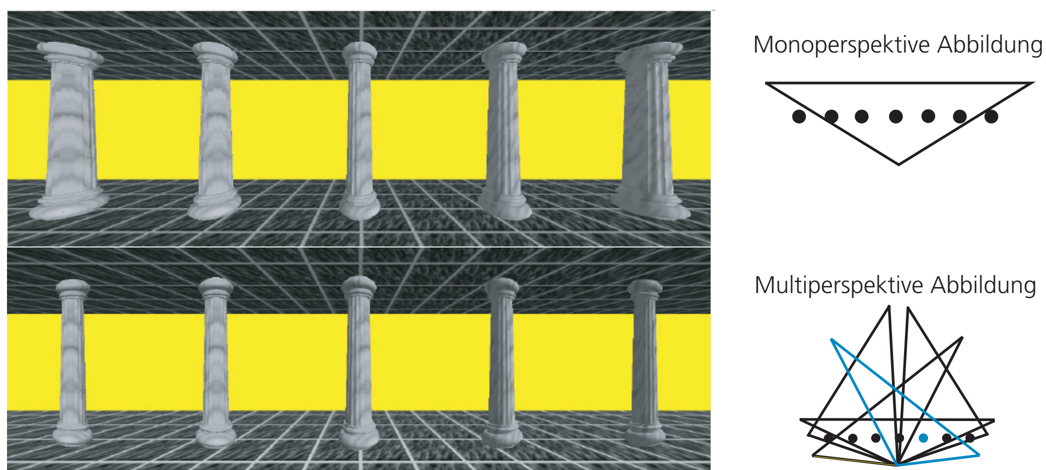


Abbildung 3.3: Reduktion der Weitwinkelverzerrungen bei gekrümmten Flächen (nach [Agrawala et al. 2000])

3.3.2 Geometrische Transformation

KÖNIG und ZAVESKY betrachten in ihren Arbeiten eine Methodik mit einem gegenüber AGRAWALA differierenden Ansatz. Darin wird ein Verfahren, basierend auf Objekttransformationen, vorgestellt, welches ebenfalls den perspektivischen Verzerrungen die durch die Zentralperspektive auftreten, entgegenwirkt.

Die perspektivische Korrektur

Ein Ansatz für die Erzeugung multiperspektivischer Abbildungen zur Minderung der unerwünschten Verzerrungen bei computergrafischen Zentralprojektionen, wird in der Arbeit von KÖNIG vorgestellt (vgl. [König 05]). Zur perspektivischen Korrektur eines Objektes wird dazu in dessen Geometrie eingegriffen. Das Verfahren beruht auf zwei nacheinander auszuführenden Modelltransformationen. Neben der Rotation als ersten Schritt wird in diesem Verfahren die Scherung als zweite Modifikation eingesetzt. Durch diese Objektverformung wird eine Sonderbildebene, in der die involvierten Objekte zentral projiziert und damit entzerrt werden, simuliert. Infolgedessen wird eine Binnenperspektive erzeugt, die aus der monoperspektivischen Darstellung der Szene eine Multiperspektivische generiert.
Die Berechnungen, die für eine wahrnehmungskonforme Darstellung von Körpern notwendig sind, basieren auf den Daten, die sich durch den Abstand zwischen dem selektierten Objekt und der Kamera ergeben. Die Eingabeparameter sowohl für die Scherung als auch für die Rotation, werden aus den Beträgen des X–, Y– und Z–Abstandes des zu korrigierenden Objektes und der Kamera ermittelt. Diese werden aus der Differenz der Komponenten der jeweiligen Ortsvektoren der Kamera und der des Objektes errechnet.
Die Resultate dieser Berechnungen dienen der Ermittlung der Scherfaktoren. Für eine bildebenenparallele Scherung eines Körpers im Raum werden zwei Scherfaktoren benötigt, die eine Transformation sowohl in X– als auch in Y–Richtung realisieren. Unter Anwendung der beiden Faktoren erfolgt die Transvektion des Objekts entlang der YZ– und der XZ–Ebene und ermöglicht die Scherung sowohl in X– als auch in Y–Richtung. Das Ergebnis dieses Transformationsschrittes ist die entzerrte Darstellung des Objektes in der Frontalansicht. Da dies jedoch nicht der ursprünglichen Orientierung des Objektes im Raum entspricht, wird eine Rotation auf dem Objekt ausgeführt. Durch diesen Schritt werden die bisher sichtbaren Objektseiten dem Betrachter zugewandt. Die Rotation des Objektes setzt sich aus einer Drehbewegung um die X– sowie die Y–Achse zusammen. Die Abbildungen 3.4 bis 3.6 veranschaulichen die Teilschritte der perspektivischen Korrektur nach KÖNIG.

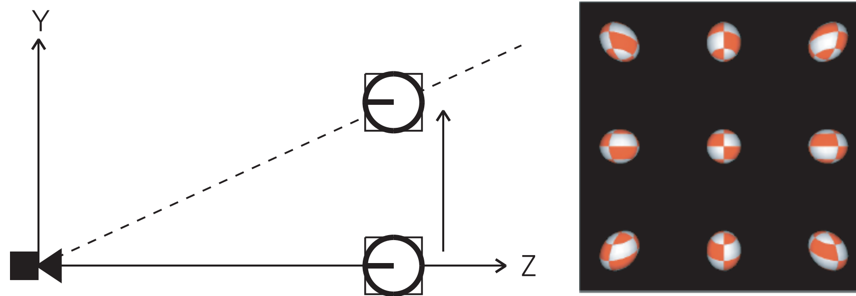


Abbildung 3.4: Ausgangssituation [Franke et al. 07]

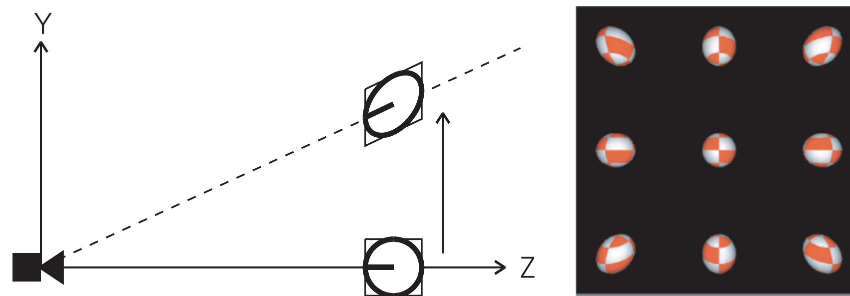


Abbildung 3.5: Positionsabhängige Scherung von Objekten [Franke et al. 07]

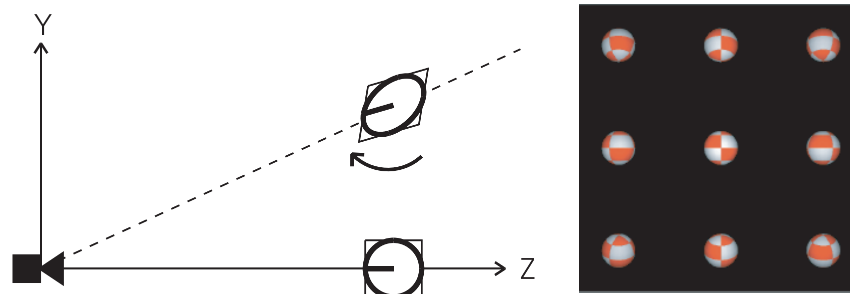


Abbildung 3.6: Rotation zu Beibehaltung der Objektansicht [Franke et al. 07]

Die Arbeit von ZAVESKY stützt sich auf diesen Algorithmus (vgl. [Zavesky 07]). Die Ergebnisse seiner Untersuchungen zeigen, dass die Resultate im Bezug auf Kamerabewegungen und Rotation von Objekten bis dato ungenügend sind. Das Modell der Kamera, welches dem Algorithmus zugrunde liegt, basiert auf der Funktionsweise des Systems, wie es in OpenGL™ verwendet wird. Darin befindet sich die Kamera im Koordinatenursprung. Damit erfolgt bei diesem Verfahren die Navigation der Kamera durch eine virtuelle Szene nur augenscheinlich. Tatsächlich erfolgt eine entgegengesetzte Translation beziehungsweise Rotation der Szene vor der starren Kamera. Demzufolge ist das Kamerakoordinatensystem gleich dem Weltkoordinatensystem und die Kamerakomponente dadurch vernachlässigbar.
Die Intension der perspektivischen Korrektur verlangt eine Scherung entlang der Bildebene. Infolge dessen ermöglicht der Algorithmus die Visualisierung verzerrungsfreier Objekte lediglich während einer Translation. Bei einer Rotation der Kamera in der

Szene werden nach dem beschriebenen Algorithmus falsche Scherfaktoren berechnet und verfälschte Scherrichtungen verwendet. Dies führt zu einem Ergebnis, welches die an das Verfahren gestellten Erwartungen nicht erfüllt.
Damit beschränkt dieser Algorithmus die korrekte Berechnung von verzerrungsfreien Objekten auf statische beziehungsweise dynamische Szenen mit Kameratranslationen, die ausschließlich parallel zu den Koordinatenachsen des Systems erfolgen dürfen.

Die erweiterte perspektivische Korrektur (EPK)

Die Weiterentwicklung der perspektivischen Korrektur nach ZAVESKY auf Grundlage des Algorithmus von KÖNIG, ermöglicht eine Berechnung zur Geometrieveränderung für eine verzerrungsfreie Darstellung von Objekten bei einer freibeweglichen Kamera (vgl. [Franke et al 07]). ZAVESKY schlussfolgert aus den Ergebnissen von KÖNIG, dass eine Scherung der Objekte relativ zur Kameraposition und –ausrichtung erfolgen muss. Dadurch wird eine bildebenenparallele Rotation und Scherung gewährleistet.
Die Grundannahme der erweiterten perspektivischen Korrektur ist die tatsächliche Bewe-gung einer Kamera durch die virtuelle Szene und damit eine Eigenbewegung der Kamera innerhalb des Weltkoordinatensystems. Der Algorithmus zur Umsetzung einer Perspektivkorrektur besteht ähnlich dem Ansatz von KÖNIG aus Teilschritten die durch eine Nacheinanderausführung zu einem verzerrungsfreien Objekt in einer Binnen-perspektive führen. Zur Umsetzung dessen erfolgt die Umrechnung der Koordinaten des Objektes aus dem Weltkoordinatensystem in das Koordinatensystem der Kamera. Damit wird die Objektposition relativ zur Kamera ermittelt. Mit Hilfe des relativen Abstandes von Kamera und Objekt wird im nächsten Schritt eine Rotation des Objektes relativ zur Ausrichtung des Kamerakoordinatensystems ausgeführt. Im Folgenden erfolgt eine Rotation sowohl um die X– als auch um die Y–Achse, um die ursprüngliche Ausrichtung des Objektes in der Szene herzustellen. Der vierte Schritt ist die Scherung des Objektes unter Zuhilfenahme der Scherungsfaktoren, die eine Rücknahme der Verzerrung erzielen. Die Ausrichtung des Scherkörpers als auch die Rotation orientiert sich am Kamerakoordinatensystem, um eine bildebenenparallele Transformation zu ermöglichen. Die Ausführung des Algorithmus erfolgt mittels Matrizenmultiplikation. Dabei werden die beschriebenen Einzelschritte als einzelne Matrizen miteinander multipliziert. Die Einzelschritte des Verfahrens zeigen sich folgendermaßen:

i. Berechnung der Objektposition P_O relativ zur Kamera

Kamerakoordinaten	$K = (x_K, y_K, z_K)$
Objektkoordinaten	$O = (x_O, y_O, z_O)$
Differenz aus den Komponenten	$\partial x = x_O - x_K$
	$\partial y = y_O - y_K$
	$\partial z = z_O - z_K$

Objektposition relativ zur Kamera $P_O = \begin{pmatrix} \partial x \\ \partial y \\ \partial z \end{pmatrix}$

i. Berechnung der Scherfaktoren aus P_O

Scherfaktor in X–Richtung $s1 = -\frac{\partial x}{\partial z}$

Scherfaktor in Y–Richtung $s2 = -\frac{\partial y}{\partial z}$

ii. Bestimmung der Schermatrix aus den Scherfaktoren

Schermatrix $\overline{S} = \begin{pmatrix} 1 & 0 & s1 & 0 \\ 0 & 1 & s2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

iii. Berechnung der Rotationsfaktoren aus P_O

Rotation um die X–Achse $r_X = -\text{arctan2}\ (\ \partial x, \partial z\ )$

Rotation um die Y–Achse $r_Y = -\text{arctan2}\ (\ \partial y, \partial z\ )$

iv. Bestimmung der Rotationsmatrizen aus den Rotationsfaktoren

Rotation um die X–Achse $\overline{R}\ (\delta_x) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \delta_x & -\sin \delta_x & 0 \\ 0 & \sin \delta_x & \cos \delta_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Rotation um die Y–Achse $\overline{R}\ (\delta_y) = \begin{pmatrix} \cos \delta_y & \sin \delta_y & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin_y & \cos \delta_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

v. Bestimmung der modifizierten Objekttransformationsmatrix
Dies erfolgt durch die Modifizierung der Transformationsmatrix des Objektes und bewirkt die Scherung und die Rotation des Objektes bezüglich des Objektschwerpunktes. Dafür wird der Rotationsteil der Objektmatrix durch die Einheitsmatrix ersetzt.

$$\overline{O} = \begin{pmatrix} r_1 & rs_1 & rs_2 & x \\ rs_3 & r_2 & rs_4 & y \\ rs_5 & rs_6 & r_3 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mapsto \quad \overline{O'} = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

vi. Bestimmung der modifizierten Kameratransformationsmatrix
Dabei wird die Transformationsmatrix der Kamera verändert, um die Scherung und Rotation entsprechend der Kameraausrichtung, unter Berücksichtigung der Eigenrotation des Objektes, auszuführen.

$$\overline{K} = \begin{pmatrix} r_1 & rs_1 & rs_2 & x \\ rs_3 & r_2 & rs_4 & y \\ rs_5 & rs_6 & r_3 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mapsto \quad \overline{K'} = \begin{pmatrix} r_1 & rs_1 & rs_2 & 0 \\ rs_3 & r_2 & rs_4 & 0 \\ rs_5 & rs_6 & r_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

vii. Die Teilschritte werden in einer Berechnungsformel zur Anwendung der erweiterten perspektivischen Korrektur zusammengeführt und ermöglichen damit eine verzerrungsfreie Darstellung von Objekten in freibeweglichen, dynamischen, dreidimensionalen Szenen.

$$[P'] = [O'] * [K'] * [S] * [R_x] * [R_z] \cdot [K']^{-1} * [O']^{-1} * [P]$$

4. Synthese und Konzeption

Im folgenden Kapitel wird auf die theoretische Umsetzung der in Kapitel Eins beschriebenen Zielstellungen eingegangen. Auf Basis verwandter Arbeiten wird ein Entwurf zur Erweiterbarkeit der Arbeitsumgebung Bildsprache LiveLab (BiLL) sowie die Konzeption eines Softwarebausteins zur Umsetzung als eine Erweiterung in der Anwendung erarbeitet. Dazu wird einführend auf Echtzeitanwendungen (4.1) und die bestehende Arbeitsumgebung BiLL eingegangen (4.2). Ferner werden die Möglichkeiten einer Erweiterbarkeit im gegebenen Kontext analysiert und bewertet (4.3). Daraufhin werden Aufbau und Funktionsweise einer Architektur für die Arbeitsumgebung beschrieben (4.4) und Betrachtungen bezüglich der Integration einer Erweiterung in die Arbeitsumgebung im Allgemeinen sowie im Speziellen (4.5) durchgeführt. Die Ergebnisse der theoretischen Untersuchungen und konzeptionellen Umsetzungen der folgenden Abschnitte dienen als Basis für die in Kapitel Fünf dargelegten praktischen Realisierungen.

4.1 Echtzeitanwendungen

Echtzeitoperationen sind zeitkritische Vorgänge. Ein ausgelöster Prozess darf nur eine begrenzte Verzögerung aufweisen, welche die Operation selbst oder das menschliche Empfinden nicht beeinträchtigen. Eine Echtzeitwiedergabe ist gegeben, wenn das menschliche Auge der optischen Täuschung unterliegt, es sehe eine Bewegung anstatt einzelner Bilder. Das Gehirn suggeriert ab 16 Bildern pro Sekunde („fps“, frames per second) eine Bewegung, wenn innerhalb der einzelnen Abbildungen eine Veränderung über die Zeit stattfindet. Besonders kritisch ist die Bewegungswahrnehmung bei der Darstellung von Animationen wie es eine frei bewegliche virtuelle Welt ermöglicht. Obwohl eine komplexe dreidimensionale Szene eine Reihe von Berechnungen benötigt bevor sie auf dem Ausgabegerät visualisiert werden kann, muss die Echtzeit gewahrt bleiben. Das Verfahren der Datenverarbeitung, welches von der vektoriellen, mathematischen Beschreibung einer Szene zum Bild auf dem Ausgabegerät führt, wird dabei als Rendering Pipeline beschrieben (vgl. [Angel 90], [Franke et al. 05a]). Rendering bezieht sich hier auf das Berechnen von dreidimensionalen Szenen aus der virtuellen Welt, die in zweidimensionale Koordinaten des Bildschirms umgerechnet werden. Zusätzlich ist die Interaktivität zwischen Anwender und System bedeutsam. Benutzereingaben und andere Ereignisse innerhalb einer Applikation beeinflussen eine dreidimensionale Szene und somit jedes auf den Vorgang folgende Bild, das berechnet und visualisiert wird. Demnach ist es notwendig, dass die Anwendung jegliche Einflüsse in Echtzeit registriert und darüber hinaus auf Änderungen in der virtuellen Welt reagiert. Die Verwendbarkeit

eines Resultates einer Echtzeitanwendung hängt damit nicht nur von diesem selbst ab, sondern ebenso vom Zeitpunkt der Bereitstellung des Ergebnisses.
Die Software Bildsprache LiveLab stellt eine derartige Anwendung, im Zusammenhang mit der Visualisierung dreidimensionaler Szenen, dar. Zur Realisierung von BiLL als Arbeitsumgebung ist das Echtzeit–Rendering unabdingbar. Der Gesamtprozess des Rendern, ist gleichzeitig Ausgangspunkt des Wirkungsbereiches der Bildsprache LiveLab Applikation.

4.2 Die Arbeitsumgebung Bildsprache LiveLab (BiLL)

Die Echtzeitanwendung BiLL ist eine Plattform für analytische Untersuchungen im Bereich der Bildsprache sowie der Wahrnehmungspsychologie und befindet sich aktuell in einem frühen Entwicklungsstadium. Mit der Applikation soll es möglich sein, verschiedene Sachverhalte der Interfacegestaltung im dreidimensionalen Raum zu untersuchen und darüber hinaus durch die Kombination verschiedener Untersuchungsansätze in einer 3D–Szene neue Erkenntnisse im Bereich der Bildsprache sowie über die Wirkung auf den Betrachter zu erhalten. Des Weiteren könnten mit dieser Arbeitsumgebung Forschungsgegenstände bezüglich der Kognitionswissenschaften in dynamischen dreidimensionalen Welten erforscht werden. Aktuelle Untersuchungen, die als Grundlage für Analysen mit Hilfe von BiLL in Betracht kommen, sind die Komposition sowie die harmonische Gestaltung von Abbildungen dreidimensionaler Welten. Beispielsweise ist durch die Integration von Binnenperspektiven eine wahrnehmungsrealistische Darstellung von dreidimensionalen virtuellen Welten möglich. Theoretische Ansätze oder Methoden, die auf Basis statischer Modelle entwickelt wurden, sollen mit BiLL dynamisiert werden. Zur Umsetzung der Basissoftware wird auf diverse Technologien und Entwicklungen im Bereich der Computergrafik sowie der Erzeugung von Benutzeroberflächen aufgebaut. Die in der Arbeitsumgebung zur Anwendung kommenden Technologien und deren Wirkungskreise werden im Folgenden dargelegt.

4.2.1 BiLL: Die Basisanwendung

Bildsprache LiveLab ist ein Anwendungssystem, das die Visualisierung von dreidimensionalen Welten und die Interaktion zwischen der Applikation und einem Benutzer in Echtzeit ermöglicht. Die Echtzeitfähigkeit ist gleichzeitig ein Basiskriterium der Arbeitsumgebung. Ferner ist diese plattformübergreifend konzipiert. Primäre Betriebssysteme sind Microsoft® Windows XP™ und Apple®–Mac OS X™. Neben einer Plattformunabhängigkeit ist die Nutzung von Open Source Technologien eine grundlegende Voraussetzung für die Entwicklung dieser Software.
In der praktischen Umsetzung basiert BiLL auf der Grafik–Engine OpenSceneGraph™ (vgl. [OSG]) und der GUI–Bibliothek Fast Light Toolkit (vgl. [FLTK]). Der Einsatz der Technologien erfolgt in der Umsetzung der Anwendung in zwei aus Benutzersicht entkoppelten Fenstern. Das Visualisierungsfenster (Abbildung 4.1) zur Darstellung der

virtuellen Szene einerseits und dem Editor (Abbildung 4.2) zur Manipulation der 3D–Welt andererseits. Beide Komponenten basieren auf jeweils einer eigenen Technologie, die in der Arbeitsumgebung verknüpft sind. BiLL verwendet zur Darstellung des Editorfensters die Bibliothek FLTK©. Dieses, wie in 2.3 beschrieben, plattformübergreifende Hilfsmittel stellt die Funktionalität zur Erzeugung graphischer Benutzeroberflächen bereit.

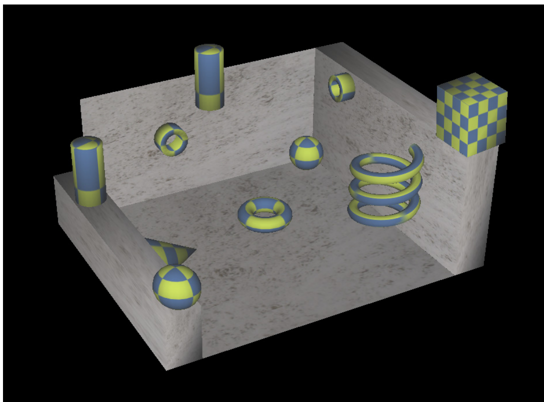


Abbildung 4.1: BiLL – Das Viewerfenster

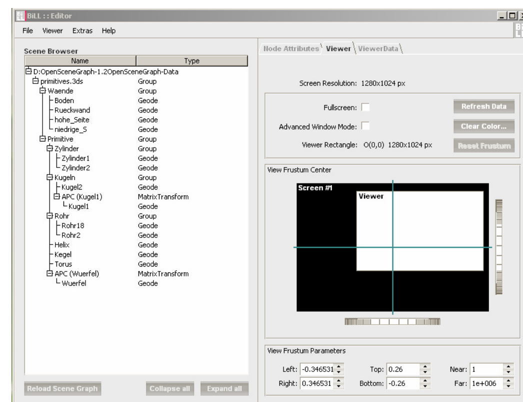


Abbildung 4.2: BiLL – Das Editorfenster

Über das Editorfenster können Manipulationen an der 3D–Szene erfolgen. Dazu werden die Basisfunktionalitäten über die integrierte Menüleiste gesteuert. Ferner gehört das Laden von Szenen ebenso wie das Beenden des Programms zum Funktionsumfang der Anwendung. Die in der Arbeitsumgebung zu visualisierenden Szenen können in einer Modellierungssoftware wie beispielsweise Autodesk® Maya™ oder Autodesk® 3D Studio Max™ erstellt werden. Die Bildsprache LiveLab Software stellt kein äquivalentes Werkzeug zur Erstellung von virtuellen Szenen dar. Aufgrund dessen ist eine Zweitsoftware zur Szenenerzeugung notwendig. Nach dem Export einer Szene aus der Zweitsoftware, kann diese in BiLL geladen werden. Die unterstützten Formate sind unter anderem Object (*.obj), Open Flight (*.flt), 3D Studio (*.3ds) und OpenSceneGraph (*.osg). Weiterhin ist die Festlegung von Beleuchtung und Szenennavigation über die Menüleiste des Editors umsetzbar. Der Szenenbrowser, der die Struktur der virtuellen Welt abstrahiert darstellt, ermöglicht die Auswahl von Knoten aus dem im Viewerfenster dargestellten Szenengraph. Die sich rechts von diesem Szenenbrowser befindlichen Karten stellen weitere Funktionen wie beispielsweise für die Manipulation des Graphens oder des Sichtkörpers der Kamera bereit. Die Ansteuerung der diversen Karten erfolgt über den zugehörigen Kartenreiter.

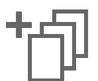
Das Visualisierungsfenster wird mit Hilfe der Grafik–Engine OpenSceneGraph™ (vgl.2.3) erzeugt. Nach dem Laden einer 3D–Szene über das Editorfenster ist eine Navigation innerhalb der dreidimensionalen Welt möglich. Die Steuerung, die über Maus oder Tastatur erfolgt, ermöglicht im Viewerfenster eine vollkommen freie Navigation in der virtuellen Welt in Echtzeit. Der visualisierten Szene als auch OSG liegt das in 2.2

erläuterte Prinzip des Szenengraphens zugrunde (vgl. [Wernecke 94], [Seewald 04]). Durch den Einsatz dieser Technologie in der Arbeitsumgebung, wird die Darstellung von dreidimensionalen Szenen in BiLL realisiert. Die Erstellung und Bearbeitung einer Szene mittels OpenSceneGraph™ ist durch das Prinzip des Szenengraphens komfortabler als eine analoge Umsetzung in OpenGL™. OSG ermöglicht dem Entwickler darüber hinaus eine unkomplizierte Einarbeitung in die Arbeitsumgebung sowie eine entwicklerfreundliche Umsetzung von Erweiterungen. Die Software befindet sich im Entwicklungsstadium und wird im Rahmen dieser Arbeit um verschiedene Funktionalitäten erweitert. Die verwendeten Technologien und der Entwicklungsstand von BiLL zu Beginn dieser Arbeit sind in der nachfolgenden Tabelle zusammengefasst.

	Visualisierungskomponente	**Editorkomponente**
Plattformen	• Microsoft® Windows XP™ • Apple® Mac OS X™	
Programmier-sprache	• C++	
Technologien	• OpenGL™ als Grafik–API • OpenSceneGraph™ als Grafik-Engine und zur Verwaltung der Szenendaten	• FLTK© zur Erzeugung der Benutzeroberfläche und Behandlung von Fenstern, Widgets und Eingabe–Ereignissen
Funktionen	• Navigation durch die Szene in Echtzeit • Selektion von Objekten in der Szene	• Manipulation des Szenengraphens • Modifikation von Basisattributen
Erweiterungen	• Manipulation der geometrischen Mitte • Erweiterter Fenstermodus	

Tabelle 1: Technologiespezifikation der BiLL Arbeitsumgebung

Eine genauere Betrachtung der in diesem Abschnitt thematisierten Technologien sowie der zu diesen konkurrierenden Alternativtechnologien und deren Vergleich im Bezug auf die Arbeitsumgebung BiLL wird von EBNER (vgl. [Ebner 07]) vorgenommen.



Zusatz 4: BiLL starten

4.3 BiLL: Eignung als Komponentenarchitektur

Im folgenden Abschnitt wird geprüft, welches Entwurfsprinzip sich unter den technischen als auch anwendungsspezifischen Voraussetzungen sowie den Anforderungen, welche an BiLL gestellt werden, eignet. Die Untersuchung erfolgt unter dem Gesichtspunkt der Erweiterbarkeit von Software und basiert auf dem Entwicklungsstand der Bildsprache LiveLab Software nach EBNER. In dieser Umsetzung stellt die Applikation eine eigenständige Arbeitsumgebung ohne ein praktikables Konzept für die Verwirklichung von Erweiterungen dar. Es ist nötig dieses zu ergänzen, um die Umsetzung der mit dieser Anwendung verbundenen Ziele zu verwirklichen.
Das Hinzufügen von Funktionalität hat eine Zunahme der Programmkomplexität von BiLL zur Folge. Dadurch wird die Entwicklung von Erweiterungen durch Dritte kontinuierlich erschwert, weil diese die Software in ihrer Gesamtheit kennen müssen, um eine fehlerfreie Erweiterung gewährleisten zu können. Eine Komponentenarchitektur ist eine Möglichkeit, die Weiterentwicklung der Software zu regulieren und zu vereinfachen. Die Arbeitsumgebung soll aufgrund dessen unter dem Gesichtspunkt einer unkomplizierten Erweiterbarkeit konzipiert werden. Den prinzipiellen Aufbau unter Einbeziehung von Erweiterungen und deren technologischen Grundlagen, verdeutlicht die folgende Abbildung.

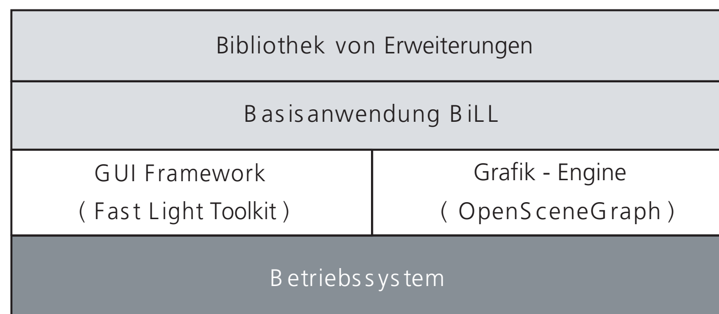


Abbildung 4.3: Architektur von BiLL

Zur Umsetzung eines Erweiterungskonzeptes ist eine Analyse der Anforderungen notwendig, aufgrund derer eine Entwurfsentscheidung bezüglich einer Architektur getroffen werden kann.

4.3.1 Kontextabhängige Anforderungen

Bevor eine Entscheidung über eine Komponentenarchitektur fixiert wird, ist es erforderlich mögliche Anwendungsszenarien zu analysieren und diese in die Entwurfsentscheidung einzubinden. Mögliche Erweiterungen der Anwendung stehen im Kontext der Wahrnehmungskonformität, dienen der Optimierung des Nutzer–Bild–Dialogs oder der Portierung von bildstrukturellen Konzepten der Renaissancekünstler in die dreidimensionale Computergrafik. Die Arbeitsumgebung ist dabei kein Erstellungswerkzeug von 3D–Szenen, sondern baut auf diesen Anwendungen auf und nutzt diese

indirekt durch die Visualisierung von, aus der Modellierungssoftware, exportierten Szenen. Die Anwendung dient hauptsächlich der gezielten Manipulation und Modifikation einer Szene. Aufgrund dessen sind mögliche Erweiterungen in diesen Bereich einzuordnen.

Ein denkbarer Ansatzpunkt ist dabei die Rendering Pipeline. Zur Realisierung einer Komponente kann der Entwickler in die einzelnen Phasen der Pipeline eingreifen und diese modifizieren. Vorstellbar ist dazu eine Abänderung des Szenengraphens. Durch eine Abwandlung der hierarchischen Szenenstruktur können Objekte einer Szene neu geordnet oder gegebenenfalls hinzugefügt oder entfernt werden. Im Bereich der Interfacegestaltung können beispielsweise Wege–Markierungen (Abbildung 4.4) für eine Navigation im dreidimensionalen Raum (vgl. [Groh 05]) oder eine narrative Panelsetzung in Bildräumen (Abbildung 4.5) zur Hervorhebung von Bildbereichen umgesetzt werden.

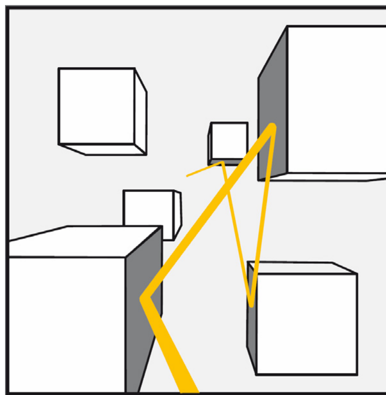


Abbildung 4.4: Wege–Markierung [Groh 05]

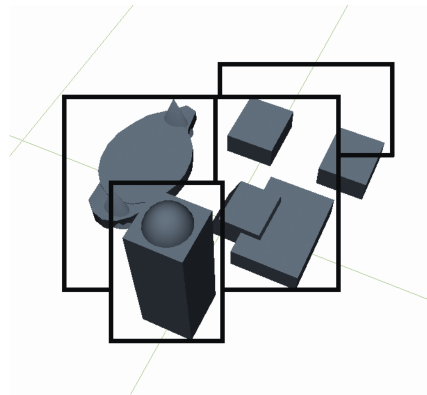


Abbildung 4.5: Narrative Panelsetzung [Kim 06]

Um eine Abbildung der Szene zu ermöglichen wird der Szenengraph durchlaufen. Das Verfahren der Traversierung ist jedoch nicht eindeutig. Durch eine Modifikation dieses kann eine geänderte Abbildung der Szene auf Basis eines gleichbleibenden Szenengraphens erfolgen, denn die Abbildung wird entsprechend dem Durchlaufen des Graphens erzeugt. Weiterhin kann die Darstellung einer virtuellen Welt in der Transformationsphase, in der eine Überführung der 3D–Daten in eine zweidimensionale Abbildung erfolgt, verändert werden. In diesem Teil der Rendering Pipeline spielt die Betrachterposition und Orientierung eine signifikante Rolle, denn durch benutzerdefinierte Clipping Planes oder die Rekonfiguration des Sichtkörpers kann Einfluss auf die Visualisierung einer Szene genommen werden. Beispielsweise wäre die Erzeugung (Abbildung 4.6) und Visualisierung (Abbildung 4.7) von Multipanoramen, denkbar.

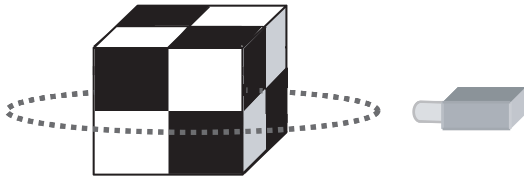


Abbildung 4.6: Kamerafahrt auf rundem Pfad

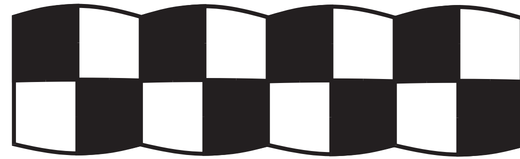


Abbildung 4.7: Multipanorama runder Kamerapfad [Franke et al. 05b]

Ferner besteht die Möglichkeit über die Tiefeninformationen einer Szene die Objekte in Beziehung zueinander zu stellen und damit die Dialogfähigkeit der Abbildung zu erhöhen. Ein möglicher Ansatz für eine Erweiterung in diesem Kontext ist die Farbperspektive (Abbildung 4.8). Die Manipulation einer Szenenabbildung mit Hilfe deren Tiefeninformation ermöglicht dies. Das Resultat ist eine für den Betrachter verbesserte Tiefenwahrnehmung der Szene. Die Erzeugung dieses und verwandter Effekte ist ebenfalls über das Postprocessing, das sich nach dem Rendervorgang anschließt, möglich (Abbildung 4.9). Es erlaubt darüber hinaus die Untersuchung von Abbildungen unter der Einbeziehung von Shader, um Veränderungen des Bildes im Hinblick auf die Wahrnehmungskonformität zu analysieren.

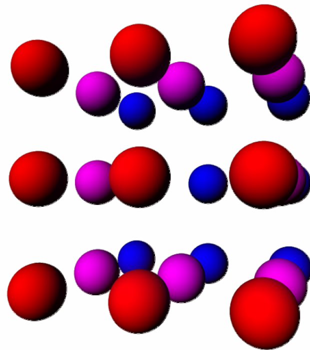


Abbildung 4.8: Farbperspektive der Tiefe [Tzscheutschler und Clemente 06]

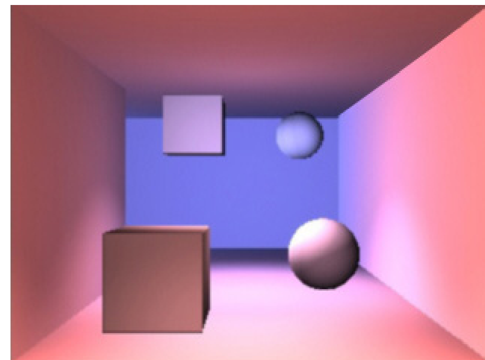


Abbildung 4.9: Farbperspektive im Post-process [Hollmann 06]

Die an dieser Stelle vorgestellten Erweiterungskomponenten zentrieren sich zusammenfassend auf drei Bereiche: Zum einen ist es die Manipulation des Szenengraphens und damit den Aufbau der Szene in seiner Datenstruktur. Zum zweiten die Kameraorientierung und die Perspektive sowie den durch die Kamera erzeugten Sichtkörper. Die Einbeziehung von Shader, die nach dem Rendern über die Abbildung gelegt werden ist ein dritter Bereich in dem sich Erweiterungen thematisch ansiedeln können. Diese Aspekte in ihren Ausprägungen werden in die Entwurfsentscheidung für eine Architektur integriert.

4.3.2 Softwaretechnologische Anforderungen

Die Anforderungen, die an Komponenten gestellt werden, teilen sich zum einen in die Nutzer- und zum anderen in die Entwicklerspezifischen auf. Für die Nutzer von Anwendungen ist es besonders wichtig, dass die jeweiligen Komponenten des Programms über die geforderte Funktionalität verfügen. Diese sollten darüber hinaus möglichst unkompliziert verwendbar sein. Dies kann besonders durch eine ausführliche Dokumentation unterstützt werden. Dabei sollte diese explizit auf die bereitgestellten Funktionalitäten und die Spezifikation der dazugehörigen Schnittstellen eingehen. Eine Beschreibung von konkreten Anwendungsfällen kombiniert mit einer beispielhaften Integrationen der Komponente kann den Einstieg zusätzlich vereinfachen. Letztendlich kann die Dokumentation mit Testfällen, Testergebnissen, technischen Details der Architektur und Leistungsangaben abgerundet werden.
Aus der Entwicklersicht ist die Ausführung der Funktionalität einer Komponente über die definierte Schnittstelle relevant, insbesondere weil deren genaue Implementierung für den Drittentwickler verborgen bleibt. Daher ist eine detaillierte Spezifikation unentbehrlich. Dabei ist es elementar, dass sich diese zu einem späteren Zeitpunkt nicht mehr verändert. Dies garantiert auch dann einen problemlosen Einsatz der Anwendungen, wenn die von der Basisapplikation angesprochenen Komponenten in unterschiedlichen Versionen in das Zielsystem integriert werden. Durch diese Entkopplung wird sowohl die Modifikation als auch ein Austausch der Komponenten ermöglicht. Aufgrund dessen können, solange den Komponenten die Funktionen über die gleichen Schnittstellen zur Verfügung stehen, diese beispielsweise durch aktualisierte Versionen ersetzt werden. Außerdem beschränkt sich die einzelne Komponentenentwicklung auf einen begrenzten Umfang, wodurch die Forderung nach qualitativ hochwertigen Komponenten gestellt werden kann, um unter anderem die Robustheit und die Fehlerfreiheit zu gewährleisten.
Besonders wichtig ist es für Komponenten, dass sie in ihrer Wirkungsweise möglichst unabhängig und daher autonom einsetzbar sind. Zur Gewährleistung dessen sollten Abhängigkeiten von externen Komponenten, Anwendungen oder Daten weitestgehend minimiert werden. Die Funktionsweise einer Erweiterung sollte ferner nicht durch das Fehlen einer anderen Komponente beeinträchtigt werden. Diese Voraussetzung eröffnet eine Flexibilität der Softwareeinheit bezüglich der zugehörigen Komponentenarchitektur sowie gegenüber neuen Anforderungen. Ferner besteht dadurch ein Toleranzbereich im Bezug auf Benutzerwünsche.

4.3.3 Vergleich von Komponentenarchitekturen

Aus den vorangegangenen zwei Abschnitten und unter Einbeziehung des bestehenden Softwaresystems BiLL werden im Folgenden zwei verschiedene Ansätze der komponentenbasierten Softwareentwicklung untersucht und verglichen. Mit Hilfe von Komponenten sowie Plug-ins kann eine Weiterentwicklung der Software erfolgen. Für die Umstellung auf eine komponentenbasierte beziehungsweise Plug–in Architektur bedarf

es eines Ausbaus der Arbeitsumgebung auf softwaretechnologischer Ebene. In beiden Fällen stehen die Erweiterbarkeit und damit die Einführung von Applikationsfunktionalität im Mittelpunkt. Der Ausbau des Anwendungsumfangs des Softwaresystems soll von Drittentwicklern durchgeführt werden. Diese weisen nur eine begrenzte Kenntnis über die internen Abläufe der Basisanwendung auf. Daher würde die bisherige Architektur für sie, vor der Entwicklung der Erweiterung, eine aufwendige Einarbeitung in das Grundprogramm bedeuten. Dies sollte möglichst vermieden werden, denn die Entwickler besitzen bezüglich einer gegebenen Problemstellung ein Spezialwissen und sollten dieses mit möglichst wenig Einarbeitungsaufwand in das System integrieren können. Unter Beachtung dieser Rahmenbedingungen werden im Folgenden Komponenten und Plug–ins bezüglich ihrer Gemeinsamkeiten und Unterschiede beleuchtet. Diese Analyse wird unter dem Aspekt der Anwendbarkeit in der Arbeitsumgebung Bildsprache LiveLab durchgeführt. Aus den Ergebnissen der Untersuchung geht eine Entwurfsentscheidung für die Architektur der Anwendung hervor.

4.3.3.1 Die Analysekriterien

Aus den aufgezeigten Anforderungen sowie Zielstellungen und Rahmenbedingungen leiten sich Analysekriterien ab, die eine Entwurfsentscheidung ermöglichen. Die nachstehenden Merkmale sind aus den bisherigen Betrachtungen abgeleitet.

i. **Hostapplikation**

   Die Grundfunktionalität der BiLL–Software und damit die Hostapplikation stellen den Container dar, in den die Integration der Komponenten erfolgt. Die Untersuchung soll die Bedingungen aufzeigen die eine Basisanwendung für die jeweilige Architektur erfüllen muss.

ii. **Unabhängigkeit**

   Die Untersuchung analysiert die bestehenden softwaretechnologischen Abhängigkeiten zwischen der Hostapplikation und den Erweiterungen sowie dieser untereinander.

iii. **Schnittstelle**

   Bei diesem Kriterium fokussieren sich die Untersuchungen auf den Aufbau der Schnittstellen als auch deren Definition und Umfang bei der Umsetzung einer Komponenten– beziehungsweise Plug–in Architektur.

iv. **Wiederverwendbarkeit**

   Die mögliche Wiederverwendung von Erweiterungen in anderen Kontexten oder Anwendungen steht im Mittelpunkt der Betrachtung des Kriteriums der Wiederverwendbarkeit.

v. **Abgeschlossenheit**

Die Prüfung des thematischen und softwaretechnischen Umfangs einer Erweiterung wird hierbei analysiert. Ferner wird die Möglichkeit untersucht ein gestelltes Problem durch einen einzelnen Softwarebaustein zu lösen.

vi. **Dynamik**

Ein weiterer Aspekt von Erweiterungen ist die Dynamik. Statische Module werden vor der Ausführung in die Datenstruktur der Anwendung integriert und somit zu einem Softwaresystem zusammengefügt. Bei der Ausführung des Programms stehen diese daraufhin zur Verfügung. Als dynamische Module werden diejenigen bezeichnet, die zur Laufzeit in das System eingefügt werden und ihre Funktionalität zur Verfügung stellen, wodurch sie wesentlich flexibler einsetzbar sind. Die Analyse soll die Dynamik von Komponenten und Plug–ins aufzeigen.

vii. **Trennung der Funktionalität**

Die Trennung von Präsentation, Datenmodell und Programmsteuerung wird unter dem Blickwinkel der Realisierbarkeit in Erweiterungen für die Anwendung betrachtet und untersucht.

4.3.3.2 Die Analyse

Folgend werden die Kriterien bezüglich der Anforderungen, welche an die Arbeitsumgebung Bildsprache LiveLab gestellt werden analysiert. Dazu werden diese getrennt voneinander untersucht, die Resultate daraufhin verglichen und diese abschließend tabellarisch zusammengefasst.

Hostapplikation

Die Hostapplikation ist der Softwarebaustein in den die Integration der Erweiterungen erfolgt. BiLL stellt eine solche Hostanwendung dar. Es ist das Grundprogramm der Arbeitsumgebung und ohne Erweiterungen ein funktionsfähiges Werkzeug. Die Software ist damit auch abzüglich der Ergänzungen für den Anwender nutzbar. Dieser Aspekt ist für Plug–in Architekturen charakteristisch, weil diese die Basisanwendung generell um Funktionalität erweitern. Bei Komponenten ist dies abweichend. Der Container, der einer Hostapplikation entspricht, verrichtet ohne installierte und aktivierte Komponenten keinen verwendbaren Dienst und ist für den Nutzer nicht dienlich.

Unabhängigkeit

Sowohl Plug–ins als auch Komponenten stellen Erweiterungen ohne gegenseitige Abhängigkeiten dar. Es sind Funktionsbausteine, die einzeln oder auch gesamtheitlich in eine Software integriert werden können. Dadurch ist es für den Entwickler möglich, eine Erweiterung ohne genaue Kenntnis der weiteren Komponenten oder Plug–ins zu entwickeln und dieses zu integrieren. Gegenseitige Abhängigkeiten bei paralleler Softwareentwicklung sind dadurch ausgeschlossen.

Schnittstellen

Die Schnittstellen sind in Plug–in Architekturen sehr spezifisch konzipiert, weil die zu integrierenden Erweiterungen für ihre Ausführung auf viele Bereiche der Anwendung zugreifen müssen. Um die Hostapplikation zu schützen muss diese daher sehr präzise sein. Komponenten hingegen haben, um eine mögliche Wiederverwendbarkeit zu gewährleisten, weniger Interaktionspunkte und somit einer schmalere Schnittstelle.

Wiederverwendbarkeit

Plug–ins stellen gemeinhin Erweiterungsmöglichkeiten bezüglich eines speziellen Programms zur Verfügung. Sie sind applikationsspezifisch und sehr eng in die Anwendung eingebunden. Komponenten und Container sind dahingegen grundsätzlich generischer, weil sie im Allgemeinen nicht an eine Anwendung gebunden sind, sondern nur die Funktionalität, unabhängig von der Basisanwendung, bereitstellen. Eine Integration einer Komponente ist daher weniger eng an eine Hostapplikation gebunden.

Abgeschlossenheit

Plug–ins sind vollständige Erweiterungen deren Funktionalität sich über mehrere technische Ebenen verteilt. Sie bilden eine abgeschlossene Einheit, die sich auf einen festgelegten Problembereich fokussiert und diesen thematisch abschließt. Diese Art der Abgeschlossenheit erfüllen Komponenten nicht. Sie beschränken sich auf einen Teilaspekt eines Themenbereiches, der meistens nur eine technische Ebene abdeckt.

Dynamik

Die Integration von Komponenten und Plug–ins ist sowohl durch eine statische Bindung als auch über eine dynamische Integration, während der Laufzeit eines Systems, möglich. Das Programm, welches die Module lädt, muss dafür Sorge tragen, dass nur Erweiterungen zum Einsatz kommen, die alle Anforderungen, die an diese gestellt werden, erfüllen. Bei dynamischen Komponenten und Plug–ins ist dies nicht nur beim Start des Systems sondern auch während der Laufzeit der Anwendung relevant.

Trennung der Funktionalität

Komponenteninfrastrukturen trennen die Funktionalität zwischen technischen Bezügen (Container) und der Anwendungslogik (Komponenten). Plug-in Architekturen trennen in erster Dimension verschiedene Bereiche der Anwendungslogik und erst in zweiter Instanz Anwendungslogik und technische Aspekte.

	Softwarekomponente	**Plug-in Komponente**
Hostapplikation	• Ohne Komponenten nicht dienlich	• Ohne Komponenten lauffähig und nutzbar
Unabhängigkeit	• Sind unabhängig von anderen Softwarebausteinen • Können einzeln installiert werden	

Schnittstellen	• Weniger spezifische Verträge	• Sehr spezifische Verträge
Wiederverwendbarkeit	• Komponenten sind generisch • Weniger enge Integration vorhanden	• Plug-ins sind an eine Anwendung gebunden • Sehr enge Integration vorhanden
Abgeschlossenheit	• Thematisch nicht abgeschlossen • Beschränkung auf eine technische Ebene	• Thematisch abgeschlossen • Über mehrere technische Ebenen verteilt
Dynamik	• Die Verwendung von statischen und dynamischen Softwareeinheiten ist möglich	
Trennung der Funktionalität	• Primär Aufteilung in technische Funktionalität und Anwendungslogik	• Primär Trennung der Anwendungslogik

Tabelle 2: Übersicht der Analyse von Komponenten– und Plug–in Architekturen

4.3.3.3 Evaluierung des Vergleiches

Eine komponentenbasierte Architektur bietet eine Reihe von Vorteilen gegenüber einem monolithischen Softwaresystem. Module in einem System, die auf die gleiche Schnittstelle zugreifen, können modifiziert oder gegeneinander ausgetauscht werden, ohne dass dies die Umgestaltung anderer Systemkomponenten zwingend erforderlich macht. Darüber hinaus ist durch eine Modularisierung des Softwaresystems die gleichzeitige Entwicklung verschiedener Erweiterungen für ein Softwaresystem möglich. Unter den gegebenen Voraussetzungen und Anforderungen haben die Untersuchungen gezeigt, dass eine Umsetzung von Modulen in einer Plug–in Architektur einer Komponentenarchitektur vorzuziehen ist. Die Gründe für diese Entwurfsentscheidung sind, wie die nachfolgenden Ausführungen zeigen, vielseitig.
Die Basisanwendung bildet das Fundament der Arbeitsumgebung. Um darauf aufbauend eine möglichst einfache Erweiterbarkeit zu gewährleisten, ist die Modularisierung der Bildsprache LiveLab Software unabdingbar. Durch die Dekomposition des Systems in einzelne Softwareeinheiten ist es möglich die Anwendung auszubauen ohne die Basisanwendung zu modifizieren. Durch die nicht vorliegende Notwendigkeit der Veränderung des Fundamentes der Anwendung, besteht die Option der Parallelentwicklung nebeneinander laufender Plug–ins. Außerdem werden die Rahmenbedingungen für eine effiziente Softwareentwicklung verbessert, weil die Komplexität der

Gesamtanwendung vor dem Plug–in Entwickler verborgen wird, indem diesem lediglich Informationen über die Erweiterungsintegration bereitgestellt werden müssen. Weiterhin besteht durch eine Rekombination von Softwarebausteinen die Möglichkeit neue Systeme mit festgelegter Funktionalität zu erzeugen und diese damit für differente Anwendungsszenarien anpassbar zu machen. Die Grundanwendung der Arbeitsumgebung ist ein funktionsfähiges Werkzeug zur Manipulation virtueller Szenen und kann damit auch ohne Erweiterungen eingesetzt werden. Diese bereits bestehende Software erfüllt dadurch eine fundamentale Voraussetzung für eine Plug–in Architektur. Zur Einbindung von thematisch abgeschlossenen Erweiterungen für die Untersuchung von Aspekten der Bildsprache bieten Plug–ins gegenüber Komponenten Vorteile, weil jeder Themenbereich als eine eigenständige Erweiterung integriert werden kann. Entwickler können dadurch einen gesamten Themenbereich integrieren und durch eine praktische Umsetzung diesen als Plug–in in der Anwendung installieren. Die Erweiterungen werden ausschließlich für die Integration in BiLL entwickelt und es bedarf keiner weiteren Nutzung in anderen Softwaresystemen. Dadurch wird der Nachteil, den eine Plug–in Architektur bereithält trivial. Zur Verwaltung der einzelnen Module dient eine Managerkomponente, die in der Hostapplikation eingebettet ist. Über diese ist es möglich vorhandene Module in die Anwendung zu integrieren, sowie wieder zu entkoppeln und damit die Funktionalität der Bildsprache LiveLab Anwendung variabel und dynamisch während der Laufzeit zu gestalten.
Eine Software die auf einer solchen Struktur beruht, ist beispielsweise Autodesk® Maya™. Diese Anwendung besitzt eine Kernfunktionalität, welche nach außen abgeschirmt ist und somit Modifikationen des Kerns der Anwendung verhindert. Über Schnittstellen ist es möglich, kontrolliert auf Bereiche der Software zuzugreifen und Maya™ mit Hilfe dieser Schnittstellen zu erweitern. Die Plug–ins werden in Bibliotheksdateien abgelegt und können während der Laufzeit mittels eines Browsers geladen und entladen werden. Dadurch kann die Software sowohl individuell als auch situativ adaptiert werden.

4.4 BiLL: Konzeption der Plug-in Architektur

Die Portierung der Plug–in Architektur von der bisher abstrakten Ebene hin zur praktischen Umsetzung in der Anwendung BiLL, soll in diesem Abschnitt analysiert und in den folgenden Ausführungen dargestellt werden. Dabei werden zunächst die Anforderungen, die an eine Umsetzung gestellt werden, untersucht und unter verschiedenen Gesichtspunkten analysiert. Dabei wird auf softwaretechnische Aspekte sowie auf Merkmale der Mensch–Computer–Interaktion eingegangen. Eine Grundvoraussetzung ist dabei vorgegeben und gleichzeitig Ausgangspunkt dieser Untersuchung. Es sollte möglich sein ein Softwaresystem zu erzeugen, welches, ohne den ursprünglichen Quellcode zu verändern und zu rekompilieren, Erweiterungen integriert und die Funktionalitäten in der Arbeitsumgebung BiLL zur Verfügung stellt. Darüber hinaus sollte

eine Architektur umgesetzt werden, die während der Laufzeit das Hinzufügen und das Entfernen von Plug–ins ermöglicht.
Diese Anforderungen setzen sich aus zwei Teilaspekten zusammen. Zum einen aus der Verwaltung der zusätzlich verfügbaren Funktionalität, zum anderen aus deren Nutzung. Für eine Umsetzung muss die Arbeitsumgebung in ihrem Kern erweitert werden. Dies erfolgt durch die Einbindung eines Managers, der den Zugriff der Anwendung auf alle Plug-ins ermöglicht. Außerdem sollten sich die Konzepte des Information Hiding und der Datenkapselung in der Architektur wiederfinden. Daraufhin kann die Kommunikation zwischen Hostapplikation und den Modulen über definierte Schnittstellen erfolgen.

4.4.1 Die Schnittstelle

In der Arbeitsumgebung BiLL fungiert die Schnittstelle als Verbindung zwischen der Haupanwendung und den Erweiterungen. Um ein fehlerfreies Arbeiten eines Plug–ins gewährleisten zu können, muss dieses Zugang zu Teilen der Hauptanwendung besitzen. Die Zugriffe erfolgen über die Schnittstelle und werden durch diese geregelt. Die Festlegungen, wie beispielsweise die Kommunikation reglementiert ist, werden von Seiten der Hauptanwendung forciert. Damit wird sichergestellt, dass ein Modul nur auf festgelegte Bereiche einen Zugriff besitzt und inwieweit diese durch die Erweiterung modifizierbar sind. Eine Komponente kann dadurch, ohne dass eine genaue Kenntnis der Implementierung der Arbeitsumgebung notwendig ist, integriert werden. Die Schnittstelle bestimmt damit einhergehend, durch ihren Aufbau sowie Umfang, den Aufwand für eine Plug–in Realisierung. Sind die Funktionalitäten und damit auch die Schnittstellen sehr spezifisch konzipiert, verursachen diese bei späteren Änderungen an deren Struktur einen großen Aktualisierungsaufwand. Widerum erfordern universelle Schnittstellen einen fundierten Weitblick, auf die späteren Nutzungsszenarien, und verursachen dadurch einen erheblichen anfänglichen Entwicklungsaufwand.
In der Bildsprache LiveLab Anwendung ist es notwendig einer Erweiterungskomponente sowohl Informationen über den aktuellen Zustand des Editors als auch über die visualisierte dreidimensionale Szene bereitzustellen. Die Schnittstelle sollte Informationen über den aktuellen Szenengraph sowie dessen Visualisierung in der Szene bereithalten. Die Darstellung der 3D–Welt erfolgt über die Open Producer Kamera. Die Daten dieser Kamera können sich kontinuierlich ändern und daher ist es unerlässlich eine stetige Übermittlung von Informationen zu offerieren, ohne jedoch einen expliziten Aufruf für eine Aktualisierung zu initialisieren. Beispielsweise sollten Kameradaten, wie die Positionierung der Kamera im Raum, die sich aufgrund der Bewegung in einer frei navigierbaren Szene in einer Sekunde mehrfach ändern kann, bereitgestellt werden. Zum anderen muss es ebenfalls möglich sein, Informationen nur zu einem von Seiten des Plug–ins festgelegten Zeitpunkt explizit anzufordern oder aber auch Funktionen innerhalb der Grundanwendung ausführen zu können. Beispielsweise sollte die Festlegung der Kameraposition innerhalb einer Szene mit Hilfe des Erweiterungsmoduls durchführbar sein. Daraus resultiert, dass die Kommunikation über die Schnitt-

stelle in zwei Richtungen erfolgen muss. In der Abbildung 4.10 wird dies veranschaulicht.

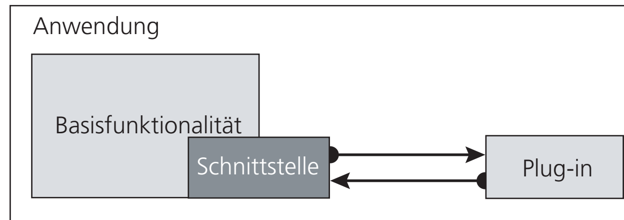


Abbildung 4.10: schematischer Aufbau der BiLL Schnittstelle

Die Schnittstelle besteht in der Arbeitsumgebung aus einer Reihe von Funktionen, sowie einer Beschreibung ihrer Semantik. Dadurch werden die Anforderungen expliziert, die von der Basisanwendung an die Erweiterungen gestellt werden. Eine konstante Schnittstellendefinition ist dabei grundlegend, da eine Änderung des Interfaces auch Modifikationen an den Erweiterungen zur Folge hat. Somit müssen die Bezeichnungen, Parameter und die Semantik stetig bleiben um Abänderungen an allen Softwarebausteinen zu verhindern. Darüber hinaus wird eine Entkopplung der in der Schnittstellenarchitektur involvierten Komponenten vorgenommen um eine Minimierung der Abhängigkeiten zu erreichen und dadurch die Kapselung der einzelnen Softwaremodule zu verstärken. Eine Abhängigkeit zwischen Komponenten sollte lediglich uni-direktional sein. Dadurch werden die Komponenten in ihrem Umfang abgrenzbarer und ferner die Wartungs– und Aktualisierungsarbeiten minimiert. Die Entkopplung von bidirektionalen Beziehungen wie sie die Abbildung 4.10 zeigt, kann durch Invertieren der beteiligten Relationen umgesetzt werden. Die nachfolgende Darstellung verdeutlicht zum einen die Ausgangssituation (Abbildung 4.11) und zum anderen den Aufbau und die Struktur der Komponenten nach der Entkopplung (Abbildung 4.12).

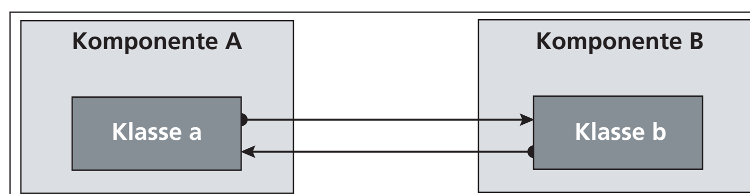


Abbildung 4.11: bidirektionale Bindung

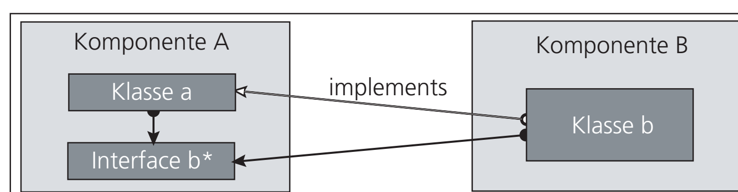


Abbildung 4.12: zwei unidirektionale Bindungen

Bestehen zwischen zwei Klassen a und b der Komponenten A und B zwei entgegengesetzte unidirektionale Verbindungen, so kann in Komponente A ein Stellvertreter für die Klasse b in Komponente B in Form eines Interfaces b* realisiert werden. Mit diesem geht a nun eine unidirektionale Verbindung ein. Die bidirektionale Kopplung findet nun nicht mehr zwischen a und b, sondern zwischen a und b* innerhalb der gleichen Komponente statt. Komponente B behält die unidirektionale Verbindung zu a und implementiert darüber hinaus zusätzlich die Schnittstelle b*. Dadurch besteht zwischen den Komponenten A und B nur noch eine schwache Kopplung.

Durch die Entkopplung der Komponenten und eine genaue Spezifikation der Kommunikationsmöglichkeiten zwischen den einzelnen Softwarebausteinen ist eine Modularisierung und damit eine einfache Erweiterbarkeit gegeben. In der Spezifikation der Anwendung bestehen wie bereits erwähnt, die Funktionen mit denen ein Zugriff von Plug–ins auf Daten der Anwendung umgesetzt werden. Da sich die Anwendung aus zwei Teilbereichen, dem Editorfenster und dem Viewerfenster, zusammensetzt, müssen beide Komponenten in die Schnittstellenspezifikation involviert werden. Den Erweiterungen werden damit aus beiden Teilbereichen der Basisanwendung Informationen zur Verfügung gestellt. Ein weiterer zentraler Punkt der Schnittstelle ist die Bereitstellung von Information des Erweiterungsmoduls für die Hostapplikation. Daten über die aktuelle Version können ebenso wie der Name des Plug–ins in die Hauptanwendung transportiert werden. Neben dem Übermitteln von Informationen ist eine Einbettung einer Plug–in Oberfläche in die Hostapplikation wesentlich. Durch sie ist eine Steuerung der Erweiterungsfunktionalität möglich. Gleichzeitig erfolgt die optische Verschmelzung der Plug–in GUI mit der Anwendungsoberfläche, wodurch das Plug–in nicht als Erweiterungsmodul sondern als integraler Bestandteil der Anwendung vom Nutzer wahrgenommen wird. Außerdem ist es über die Schnittstelle möglich Einstellungen sowie Attribute, die Veränderung an der Basisanwendung herbeiführen, festzulegen. Grundeinstellungen wie die Szenenbeleuchtung können damit über das Plug–in gesteuert werden.

Neben dem Offerieren von Informationen und der Visualisierung einer Benutzeroberfläche ist die Bereitstellung der Funktionalität des Moduls in BiLL fundamental, weil sonst eine Nutzung des Plug–ins unmöglich ist. Die Informationen die an die Basisanwendung übermittelt werden, sind Ergebnisse von Funktionen, die innerhalb des Plug–ins ausgeführt werden. Das Bereitstellen von verwertbaren Informationen wird jedoch nur durch die Übermittlung von Daten aus entgegengesetzter Kommunikationsrichtung ermöglicht. Die Verfügbarkeit von Ausgangsdaten aus der Hostapplikation ist daher gleichbedeutsam. Diese Daten entstammen sowohl dem Editorfenster als auch dem Viewerfenster. Der Szenengraph als Grundlage der virtuellen Welt innerhalb der Arbeitsumgebung steht dem Plug–in in seiner Gesamtheit zur Verfügung. Somit können Plug–ins die Datenstruktur lesen und modifizieren und dadurch diese und die auf ihr ausgeführte Traversierung verändern.

Ein weiteres zentrales Element von BiLL ist die Kamera, die innerhalb der dreidimensionalen Welt positioniert ist. Diese kann über die Schnittstelle gelenkt und modifiziert werden. Die Manipulation des Sichtkörpers oder die Festlegung der Auflösung auf dem Ausgabegerät sind nur einige Bestandteile des Kameramodells, welches über die Schnittstelle vom Plug–in angesprochen werden kann.
Mit der Schnittstelle erhält der Entwickler eines Moduls kontrollierten Zugriff auf die verschiedenen Bereiche der Applikation und kann diese in einem für die Entwicklung von Erweiterungen notwendigen Maße manipulieren. Eine genauere Betrachtung der Spezifikation mit Beschreibung der einzelnen Funktionen der Schnittstelle wird in 5.2.1 dargelegt.

4.4.2 Der Plug–in Manager

Der Plug–in Manager ermöglicht die Verwaltung und Organisation von Erweiterungskomponenten in der Arbeitsumgebung. Dieser ist neben der Schnittstelle ein zweites Bindeglied zwischen der Basisanwendung und den Erweiterungen. Die folgende Abbildung zeigt die prinzipielle Struktur des Gesamtsystems.

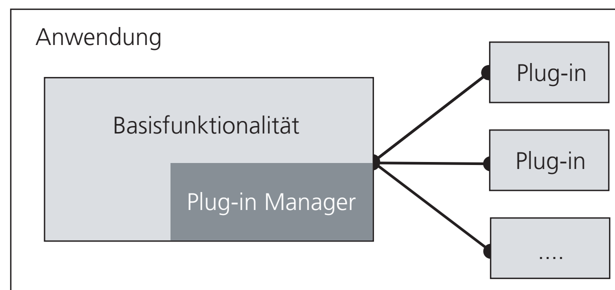


Abbildung 4.13: schematische Darstellung der Plug-in Architektur

Der Plug–in Manager ist in die Basisanwendung integriert und legt die Konventionen fest, denen die Erweiterungen für eine Einbindung unterliegen. Bei Nichtbeachtung dieser Regelungen würden diese von der Anwendung nicht unterstützt werden und folglich zu möglichem Fehlverhalten oder zur Beendigung der Anwendung führen. Die Konsequenz daraus ist die Ausführung eines Schutzmechanismus, der ein Laden falscher oder unvollständiger Bibliotheken in die Arbeitsumgebung verhindert. Neben dem Schutz der Hostapplikation verwaltet und organisiert der Manager das Laden und das Entladen alle zur Verfügung stehenden Erweiterungen. Darüber hinaus fungiert dieser als Kontrollinstanz beim Aufruf von Erweiterungskomponenten um den Zugriff nur auf registrierte und geladene Komponenten zu legitimieren. Neben den technischen Anforderungen an den Manager, werden aus der Nutzersicht ebenfalls Ansprüche erhoben. Aus der Sicht des Anwenders ist eine intuitive und möglichst einfache Bedienung des Plug–in Manager unabdingbar. Diesen Ansprüchen soll ein graphisches Nutzerinterface, welches in die Anwendung integriert ist, genügen. Folgende Funktionen sind im Manager umgesetzt und stellen die Funktionalität des Nutzerinterfaces dar:

i. Mit einer Auflistung aller zur Verfügung stehender Erweiterungen erfolgt die Bereitstellung möglicher Plug–ins in der Anwendung. Damit wird ein Überblick über den optionalen Funktionsumfang der Arbeitsumgebung gewährleistet.
ii. Ein individuelles Laden und Entladen von Plug-ins ermöglicht es dem Nutzer, die Anwendung nach seinen persönlichen Anforderungen anzupassen.
iii. Mit einem quasiparallelen Laden beziehungsweise Entladen von allen bereitstehenden Plug-ins ist es einerseits möglich den gesamten Funktionsumfang der Anwendung dem Nutzer zu offerieren, zum anderen kann durch einen Aufruf die Arbeitsumgebung in den Grundzustand zurückversetzt werden.
iv. Das Nutzerinterface ermöglicht die Visualisierung des aktuellen Ladestatus jeder einzelnen Erweiterung. Dies ermöglicht den Einblick in den momentanen Konfigurationsstand der Anwendung.
v. Durch eine Autostartfunktion werden Plug–ins die ein entsprechendes Attribut besitzen beim Start der Anwendung automatisch geladen. Somit wird dem Anwender bei jeder Programmausführung automatisch eine individuelle Programmfunktionalität bereitgestellt.

Neben den Anforderungen, die von Seiten des Nutzers an eine Anwendung gestellt werden, ergeben sich unter softwaretechnologischen Gesichtspunkten weitere Kriterien, die für einen problemfreien und effizienten Gebrauch des Plug–in Managers Voraussetzung sind.

i. Der Manager stellt verschiedene Basistechnologien, wie zum Beispiel FLTK© für Benutzeroberflächen oder OpenSceneGraph™ Funktionen zur Manipulation von 3D–Szenen bereit. Diese können in der Plug–in Entwicklung genutzt werden.
ii. Die Eindeutigkeit einer Erweiterung ist gegeben. Damit wird eine systemweit eindeutige Kennung realisiert.
iii. Die Management–Komponente listet verfügbare Module auf und kann diese über die eindeutige Kennung bereitstellen.
iv. Dem Manager sind die Verzeichnisse, in denen sich die Erweiterungen befinden, bekannt. Dadurch sind das Laden und damit auch die Nutzung der Funktionalität der Erweiterungen möglich.
v. Ein Schutzmechanismus verhindert das Laden falscher Plug–ins in die Arbeitsumgebung
vi. Neue Plug–ins können auch während der Laufzeit der Anwendung dem Manager bekannt gemacht werden.

Das nachfolgende Schema (Abbildung 4.14) zeigt eine statische Sicht auf den Plug-in Manager. Eingebettet in die Anwendung, verfügt dieser über Zugriffsrechte auf ein Interface und eine Informationseinheit. Über die Schnittstelle kann der Manager die

Verwaltung von Plug–ins vornehmen. Daten bezüglich einer Erweiterung sowie dessen aktuellen Zustand erhält der Manager über die Plug-in Information. Diese wird sowohl vom Manager als auch vom Plug–in selbst, wenn dieses geladen ist, aktualisiert. Durch diese Architektur wird die Verwaltung der Erweiterungsmodule, die sich in drei Hauptaufgaben aufteilt, ermöglicht.

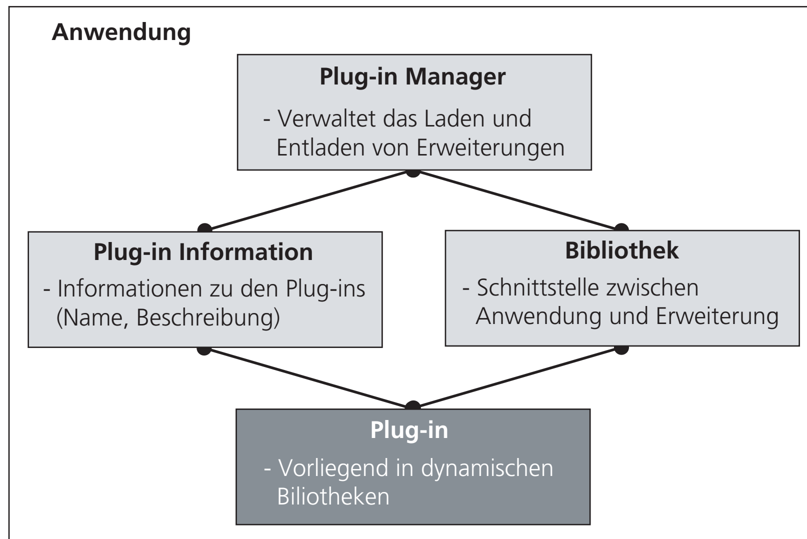


Abbildung 4.14: Architektur des Plug–in Managers

Die erste Aufgabe beinhaltet das Laden der zur Verfügung stehenden Erweiterungen. Der Vorgang des Bereitstellens der, in den Plug–ins, festgelegten Funktionalität wird im folgenden Abschnitt beschrieben und darüber hinaus im Sequenzdiagramm (Abbildung 4.15) dargestellt. Den ersten Schritt dieses Prozesses bildet der Aufruf des Managers aus der Basisanwendung heraus. Dadurch wird dieser veranlasst in festgelegten Verzeichnissen nach Bibliotheksdateien zu suchen. Nach Beendigung dieses Vorgangs wird ein Kontrollmechanismus gestartet, der alle zur Verfügung stehenden Bibliotheken nach deren Struktur und dadurch bezüglich ihrer Verwendbarkeit in der Arbeitsumgebung untersucht. Somit werden dem Anwender zu einem späteren Zeitpunkt ausschließlich der Arbeitsumgebung zugehörige Plug–ins bereitgestellt. Nach der Kontrolle erfolgt der Ladevorgang. Dabei werden die Bibliotheken mittels des Managers geladen und initialisiert. Nach erfolgreicher Beendigung werden der Status über den Ausgang des Vorgangs sowie weitere Informationen über die Komponente an den Manager übermittelt. Als letzten Schritt wird die Aktualisierung der Benutzeroberfläche umgesetzt, um dem Anwender die Funktionalität der Erweiterung anzubieten und abstrahiert zu visualisieren. Mit diesem Schritt ist der Ladevorgang abgeschlossen. Daraufhin stehen die geladenen Module zur Verfügung und können vom Anwender über die GUI der Arbeitsumgebung genutzt werden. Die Kommunikation zwischen der Hostapplikation und dem Plug–in wird über die definierte Schnittstelle realisiert.

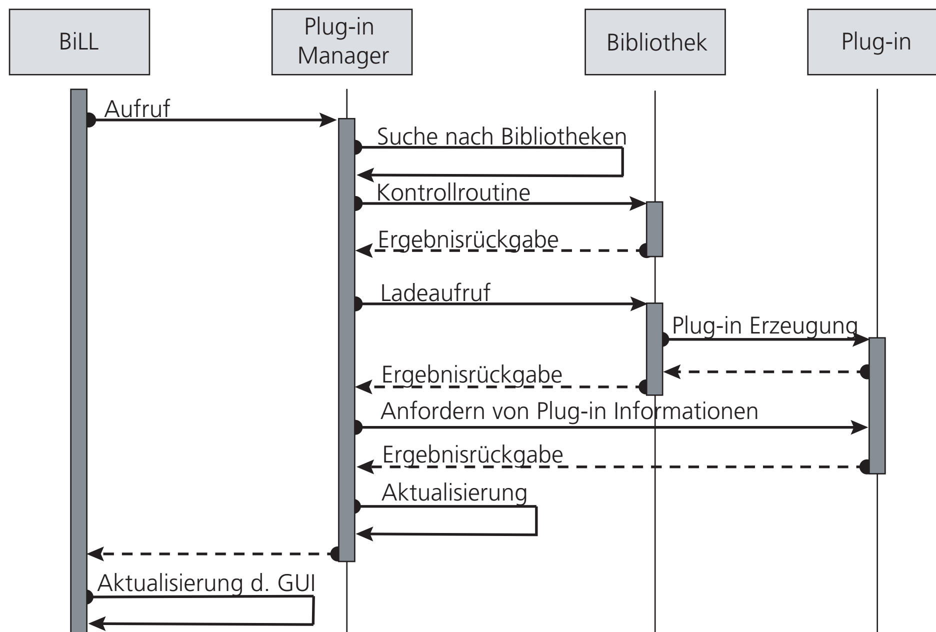


Abbildung 4.15: Sequenzdiagramm des Ladevorgangs eines Plug–ins

Das zweite Szenario zeigt die Nutzung einer Erweiterungskomponente. Diese wird ausgeführt, wenn der Anwender beispielsweise über die Benutzeroberfläche der Anwendung eine Aktion veranlasst. Vor der Ausführung der dahinter liegenden Funktion wird zunächst von Seiten des Managers kontrolliert ob diese innerhalb des zugehörigen Plug–ins ausgeführt werden kann. Die Ausführung ist nur bei einwandfreier Bereitstellung der Erweiterung möglich. Wenn dieser Kontrollaufruf erfolgreich ist, wird die gewünschte Funktion ausgeführt. Der Aufruf der Funktion wird über die Schnittstelle an das Modul übermittelt und in diesem ausgeführt. Ab diesem Zeitpunkt besitzt die Erweiterung die Kontrolle über den Ablauf der Anwendung. Gleichzeitig können Daten an die Erweiterung übermittelt werden, die zur Umsetzung der Funktionsausführung notwendig sind. Damit stehen der Erweiterung Informationen über den Zustand der Arbeitsumgebung zur Verfügung. Darüber hinaus können von Seiten des Plug–ins Anfragen an die Schnittstelle gestellt werden um weitere benötigte Daten zu erhalten. Nach der Beendigung des Funktionsaufrufs werden die Ergebnisse der Operation an die Hostapplikation übermittelt und diese wird entsprechend angepasst. Ab diesem Zeitpunkt ist der Ablauf der Arbeitsumgebung wieder unter der Kontrolle der Basisanwendung. Die folgende Abbildung zeigt den Ablauf der Funktionsausführung eines Plug–ins.

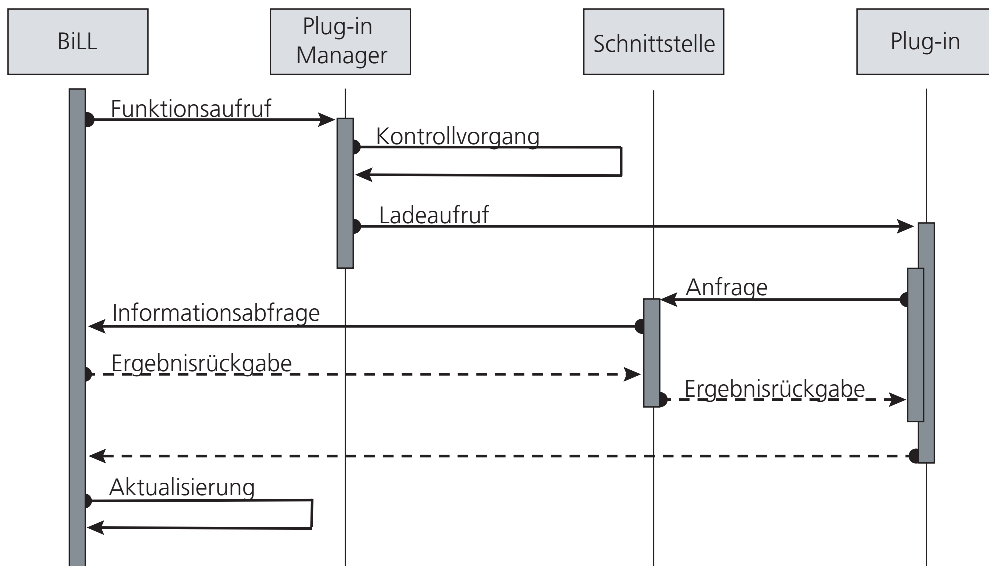


Abbildung 4.16: Ausführung einer Plug–in Funktion in der Arbeitsumgebung

Eine weitere Aufgabe, welche im Folgenden beschrieben wird, ist das Entladen einer Erweiterung (Abbildung 4.17). Dieser Vorgang kann nur nach vorherigem Laden des entsprechenden Plug–ins erfolgen. Der Ablauf ist konträr zum Ladevorgang. Das Entfernen eines Moduls erfolgt nach einem vorangegangenen Aufruf des Managers über die GUI der Arbeitsumgebung. Da dieser über die Informationen zu den aktuellen Ladezuständen aller Erweiterungen verfügt, kann das Entfernen eines nicht geladenen Moduls ausgeschlossen werden. Der Anwender initiiert den Entladevorgang. Daraufhin wird das Plug–in zerstört und die dazugehörige Bibliothek aus der Arbeitsumgebung entfernt. Nach erfolgreichem Abschluss des Vorgangs wird der Manager aktualisiert. Damit ist ein erneutes Laden der Erweiterung möglich. Im letzten Schritt wird die Oberfläche der Arbeitsumgebung dem aktuellen Zustand des Programms angepasst.

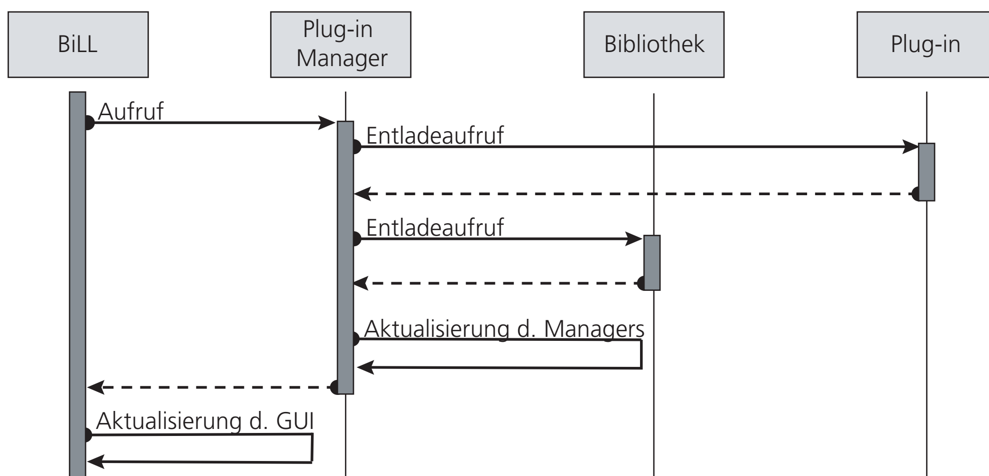


Abbildung 4.17: Vorgang zum Entladen eines Plug–ins

4.4.3 Das Plug–in

Eine Erweiterung der Funktionalität der Bildsprache LiveLab Anwendung erfolgt mit Hilfe von Plug–ins, die während der Laufzeit in die Arbeitsumgebung integriert werden können. Die für den Nutzer sichtbaren Bestandteile der Erweiterung sind dessen Interaktionselemente innerhalb von BiLL. Diese werden für die Wechselbeziehung zwischen dem Programm und dem Anwender benötigt. Die Umsetzung dieser GUI liegt somit im Aufgabenbereich der Erweiterungsentwicklung und wird im Zuge dieser umgesetzt. Dem Entwickler steht dafür ein Platzhalter in Form einer Karte in der Anwendung zur Verfügung. In diese werden die Benutzerinteraktionselemente integriert. Das Design der Bedienoberfläche ist dabei dem Entwickler überlassen, solange sich die Gestaltung auf das zugesicherte Segment beschränkt. Andere Bereiche, wie beispielsweise der einer Karte zugehörige Kartenreiter, können nicht durch Drittentwickler modifiziert werden. Damit wird ein einheitliches Layout der Oberfläche in der Arbeitsumgebung gewährleistet. Die Funktionalität als zweiter Teil einer Erweiterung wird über das Editorfenster steuerbar. Eine Erweiterungskomponente setzt sich aus der Oberfläche und der mit dieser verknüpften Funktionalität zusammen und wird als Einheit in die Anwendung integriert. Die Umsetzung der Bereitstellung erfolgt über die Managerkomponente und die Kommunikation über die Schnittstelle.
Die Entwicklung eines Plug–ins ist gleichbedeutend mit der Konzeption und der Umsetzung einer Software. Daher gelten für dessen Entwicklung die gleichen Vorgaben und Rahmenbedingungen wie bei der Realisierung einer eigenständigen Anwendung. Die nachfolgende Abbildung zeigt ein Schema, welche die Einzelschritte der Plug–in Entwicklung beinhaltet und darüber hinaus die Eingliederung in die bestehende Anwendung verdeutlicht.

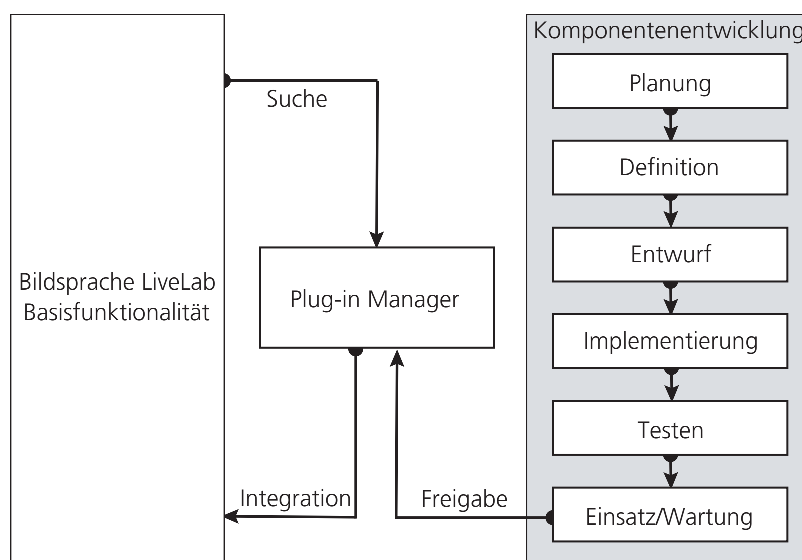


Abbildung 4.18: Entwicklung und Integration eines Plug–ins in BiLL

4.5 Die erweiterte perspektivische Korrektur als Plug–in

Nachdem im letzten Abschnitt die Plug–in Entwicklung im Allgemeinen analysiert wurde, thematisiert dieser die Entwicklung einer konkreten Erweiterung. Die Umsetzung der erweiterten perspektivischen Korrektur in BiLL wird nachstehend betrachtet und eine Konzeption für die Realisierung erarbeitet.

4.5.1 BiLL: Integration der EPK

Die Umsetzung der erweiterten perspektivischen Korrektur zur Verminderung von unerwünschten Verzerrungen bei der computergrafischen Zentralprojektion in der Arbeitsumgebung Bildsprache LiveLab erfolgt mit Hilfe eines Plug–ins. Das zugrunde liegende Verfahren wird in 3.2.2 beschrieben.

Das Verfahren der EPK basiert auf einer geometrischen Transformation des zu korrigierenden Objektes. Eine gezielte Veränderung der Objektgeometrie führt zu einer wahrnehmungskonformen Darstellung eines Körpers in der Abbildung einer dreidimensionalen Szene. Das Resultat ist die Erzeugung einer Binnenperspektive innerhalb der Abbildung und folglich ein multiperspektivisches Bild der Szene. Die Anpassung des Objektes erfolgt im Transformationsschritt der Rendering Pipeline. Die darauf folgenden Phasen innerhalb der Pipeline können unabhängig davon ausgeführt werden. Durch die Anwendung des Verfahrens innerhalb der Transformationsstufe kann die Perspektivkorrektur eines Körpers auf dem Szenegraph erfolgen. Dieser liegt der Visualisierung einer dreidimensionalen Szene in der Arbeitsumgebung zugrunde. Gleichzeitig erlaubt diese Gegebenheit eine Integration des Verfahrens in Systeme, die wie BiLL auf Basis einer Rendering Pipeline arbeiten.

Eine Abwandlung des Szenengraphens erzwingt eine unmittelbar veränderte Abbildung der Szene, weil der Graph durchlaufen wird und die Änderungen innerhalb der Graphenstruktur in Echtzeit umgesetzt werden. Somit wird nach der geometrischen Veränderung eines Objektes, dieses in der korrigierten Form visualisiert.

Transformationen an der Objektgeometrie, die sich über die Zeit verändern, werden in einem Szenengraphen nicht auf dem Objektknoten selbst ausgeführt. Das gewählte Objekt besitzt einen Transformationsknoten als dessen Elternknoten. Die Transformationsschritte, die in diesem Knoten deklariert werden, vererben sich auf die Kindknoten und damit auf das Objekt auf welches die Transformation angewendet werden soll. Für die Umsetzung der EPK in der Bildsprache LiveLab Arbeitsumgebung wird somit ein Matrix–Transformationsknoten als Elternknoten im Szenengraphen platziert. In diesem Knoten werden die Matrizen unter Vorgabe der Daten von Kamera und Objekt erzeugt und durch Multiplikation dieser die Transformationsmatrix ermittelt. Die Matrix des Transformationsknotens wird während der Traversierung des Graphens an den Kindknoten übertragen und die Geometrieveränderung am Körper vorgenommen. Die Abbildungen 4.19 und 4.20 veranschaulichen die Neustrukturierung des Szenengraphens zur Umsetzung der erweiterten perspektivischen Korrektur.

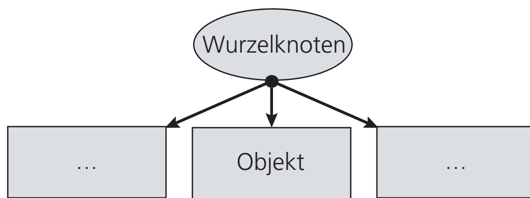


Abbildung 4.19: Szenengraph ohne Transformationsknoten

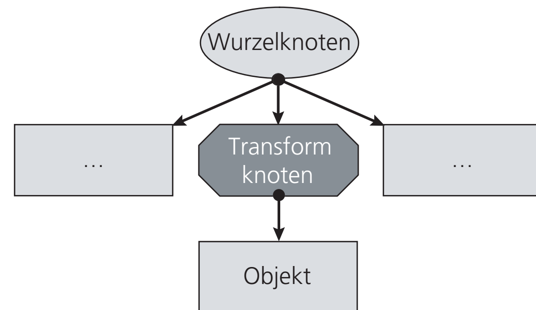


Abbildung 4.20: Szenengraph mit Transformationsknoten

Die Einbindung des Verfahrens in die Arbeitsumgebung erfolgt über den Plug–in Manager der Anwendung. Nach Beendigung des Plug–in Ladevorgangs steht eine Benutzeroberfläche für die Ausführung der erweiterten perspektivischen Korrektur in einer neu erzeugten Karte bereit. Über diese kann auf ein zuvor ausgewähltes Objekt die Perspektivkorrektur angewendet werden. Dabei erfolgt die eben beschriebene Manipulation des Szenengraphens. Durch die Aktualisierung des Graphens, die einmal pro Frame durchgeführt wird, erfolgt zu jedem Zeitpunkt die Berechnung der korrigierten Objektgeometrie und damit die Darstellung des Körpers in einer Binnenperspektive.

4.5.2 Evaluierung der EPK

Bisher wurde das Verfahren der erweiterten perspektivischen Korrektur in vorgerenderten Animationen angewandt und untersucht. Im Gegensatz dazu, wird die EPK mit Hilfe von BiLL in einer Umgebung mit freier Navigation für dreidimensionale Szenen umgesetzt. Das Verfahren wird damit erstmalig in eine Echtzeitumgebung integriert. Einschränkungen und Grenzen des Verfahrens werden durch die vorliegenden Rahmenbedingungen unmittelbar erkennbar.
Die erweiterte perspektivische Korrektur involviert in ihrer Ausführung lediglich die Position und Orientierung der Kamera sowie die des Objektes zu dieser. Andere Objekte der Szene, die der Zentralperspektive unterworfen sind, werden nicht in die Berechnungen einbezogen. Dadurch zeigen sich Durchdringungseffekte zwischen veränderten und nicht modifizierten Objekten der Szene. Die Durchdringungseffekte entstehen durch die geometrischen Manipulationen am Objekt, welche durch die EPK hervorgerufen werden. Demzufolge entstehen die Durchdringungseffekte durch die Rotation des gescherten Körpers während der Ausführung des Verfahrens. Aufgrund dessen ist die Anwendung der EPK nicht in jedem Kontext möglich. Dies zeigt sich im Einklang mit den von GROH (vgl. [Groh 05]) dargelegten Regeln für den Verstoß darzustellender Objekte gegen das allgemeine perspektivische System:

> *Ummantelung durch geschlossene gekrümmte Flächen (hier ist ein Gebiet geometrischer Gestalten gemeint, das von den komplexen Erscheinungen*

menschlicher Leiber und Naturformen bis hin zu zylindrischen Säulen und Kugeln reicht)
Singularität und Isoliertheit (gemeint ist die Losgelöstheit vom systemischen Verbund und ein additives Verhältnis zu anderen Objekten) und dialogische Bedeutsamkeit
([Groh 05], S. 49)

Nach GROH ist die Anwendung einer Perspektivkorrektur nur bei isolierten Objekten und Körpern, die eine Singularität innerhalb eines Bildes darstellen, möglich. Dieser Umstand ist jedoch bei einem Verbundobjekt nicht gegeben. Aufgrund dessen ist eine Anwendung des Verfahrens in derartigen Objektkonstellationen bezüglich der Wahrnehmungskonformität nicht wirksam.
Ein Ansatz für die Minderung der Durchdringungseffekte stellt die Translation des Pivotpunktes dar. Dieser befindet sich generell im Zentrum des Bounding Volumens eines jeden Objektes. Die Transformationsschritte des Verfahrens und damit die Scherung und ebenso die Rotation des Körpers erfolgen um diesen Punkt. Durch die Transformationsschritte kann eine Durchdringung mit anderen Körpern erfolgen. Durch die Neupositionierung erfolgen die Verfahrensschritte um einen veränderten Punkt im Raum und führen zu einer Minderung der Durchdringungseffekte. Ein weiterer Ansatzpunkt zur Minimierung des Effektes ist die differenzierte Anwendung des Verfahrens. Durch eine Anpassung der Rotationsfaktoren an den gegebenen Szenenkontext, können Durchdringungen vermieden werden. Untersuchungen diesbezüglich werden im Abschnitt 5.3 beschrieben. In diesem wird dargelegt, inwieweit die vorgestellten Ansätze zu einer Anwendung der EPK bei Verbundobjekten führen.

5. Praktische Umsetzung

In diesem Kapitel wird die praktische Umsetzung der Plug–in Architektur, wie sie im vorherigen Kapitel dargelegt wurde, beschrieben. Dabei steht die Implementierung der einzelnen Elemente der Architektur im Mittelpunkt. Im Abschnitt 5.1 wird dazu die Arbeitsumgebung Bildsprache LiveLab aufgezeigt und deren Aufbau beschrieben. Die Umsetzung der Plug–in Architektur erfolgt in 5.2. Ferner wird in 5.3 die Realisierung des Plug–ins zur Erzeugung von Binnenperspektiven in Abbildungen dreidimensionaler Szenen erläutert. Alle Implementierungen werden in der Programmiersprache C++ (vgl. [Wolf 06]) sowie unter Einbindung der OpenSceneGraph™– und Fast Light Toolkit© Bibliotheken (vgl. [OSG], [FLTK]) vorgenommen.

5.1 BiLL: Die Arbeitsumgebung

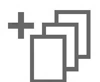
Die Bildsprache LiveLab Arbeitsumgebung ist ein Programm zur Manipulation virtueller Szenen. Es ist eine Zwei–Fenster–Anwendung bestehend aus einem Editor– und einem Viewerfenster. Beide laufen in jeweils eigenständigen Threads. Bei der Ausführung der Anwendung werden diese nacheinander erzeugt. Dadurch können beide Fenster quasiparallel arbeiten und vom Nutzer verwendet werden. Der Viewerthread bildet die Basis der Anwendung. Demzufolge hat, im Gegensatz zum Editor, ein Schließen des Viewers die Beendigung der Anwendung zur Folge. Neben der Initialisierung der zwei Threads werden der Plug–in Manager sowie eine Instanz der Schnittstelle erzeugt. Der Manager sowie das Interface werden mit dem Editorthread verknüpft. Damit ist gewährleistet, dass der Manager in den Editor eingebunden ist und dessen Funktionalität im Programm bereitsteht. Durch die Einbindung der Schnittstelle wird das Interface zwischen den Erweiterungen und der Hostapplikation positioniert.
Ferner wird beim Ausführen der Main-Methode die FLTK–Oberfläche der Arbeitsumgebung erzeugt und der Editorthread mit dem Viewerthread verbunden sowie folgend initialisiert. Der Startvorgang der Anwendung wird in der Datei BiLL.cxx umgesetzt und zeigt sich in der Main–Methode der Anwendung wie folgt:

```
void main()
{
	// Initialize Threading, so that the program can use threads
	OpenThreads::Thread::Init();

	// Instantiate threads
	EditorThread		editorThread;
	ViewerThread		viewerThread;
	LoadManager		loadManager;
	BillInterface		billInterface(&viewerThread);
```

```
        // Set up components of the application
        editorThread.setViewerThread(&viewerThread);
        editorThread.setBillInterface(&billInterface);
        editorThread.setLoadManager(&loadManager);
        editorThread.createEditorWindow();
        viewerThread.setEditorThread(&editorThread);
        billInterface.setLoadManager(&loadManager);
        billInterface.setEditorWin();

        // Configure thread, set their priorities, and spawn them
        editorThread.start();
        viewerThread.start();
        // Join the program's process with the threads
        //and call thread-specific run() methods
        viewerThread.join();
        // End threads after run() returned
        viewerThread.cancel();
        return;
}
```

Die Oberfläche des Plug–in Managers wird während der Erzeugung des Editors als eigenständiges Fenster in die GUI der Anwendung eingebunden. Dadurch wird die Bedienoberfläche des Managers zusammen mit den FLTK–Komponenten des Editorfensters erzeugt. Während des Instanziierungsprozesses wird in der Menüleiste des Editorfenster ein neuer Eintrag unter „Extras" erzeugt. Über diesen Menüeintrag wird das Plug–in Managerfenster angezeigt. Eine genauere Betrachtung der internen Abläufe des Managers wird in 5.2.2 dargelegt. Ferner erfolgt das Erstellen der Karten und Kartenreiter der Plug–ins, die über das Managerfenster mit einer Autostartoption versehen wurden.

Durch die Verbindung von Viewer– und Editorthread besteht eine Kommunikation zwischen beiden Anwendungsfenstern, woraufhin sich durch den Anwender ausgelöste Ereignisse direkt auf beide Fenster auswirken können. Somit ist gewährleistet, dass ein Ergebnis einer Operation im Editorfenster ohne Zeitverzögerung äquivalent im Viewerfenster dargestellt wird. Der Umfang der verfügbaren Funktionalitäten beschränkt sich in der Basisanwendung auf Grundfunktionen einer Echtzeitanwendung sowie die Manipulation des Sichtkörpers der verwendeten Open Producer Kamera. Das Ziel möglicher Erweiterungen ist eine veränderte Abbildung der Szene im Viewerfenster. Die Steuerung der Modifikationen erfolgt jedoch über das Editorfenster. Daher müssen die Erweiterungen Einfluss sowohl auf die Viewer– als auch auf die Editor–Funktionalität besitzen.



Zusatz 5: BiLL starten

5.2 BiLL: Umsetzung der Plug–in Architektur

Die Erweiterbarkeit der Anwendung ist durch eine Plug–in Architektur realisiert. Die Umsetzung und Implementierung dieser wird in den nächsten Abschnitten aufgezeigt. Zunächst wird auf die Schnittstelle eingegangen, die eine Verbindung zwischen Erweiterung und Grundanwendung herstellt (5.2.1). Danach werden die Verwaltung der

Plug–ins mittels eines Managers (5.2.2) und die Grundstruktur einer Erweiterungskomponente (5.2.3) beschrieben.

5.2.1 Die Realisierung der Schnittstelle

Die Schnittstelle setzt sie aus zwei Klassen zusammen. Zum einen werden dem Plug–in während der Laufzeit kontinuierlich Daten aus der Arbeitsumgebung zur Verfügung gestellt. Dies erfolgt über die Klasse Plug_Object. Die darin umgesetzten Funktionen können abgeleitet und in der erbenden Klasse überschrieben und damit spezialisiert und erweitert werden. Die Klasse Plug_Object bildet das Verbindungsglied zwischen der Grundanwendung und den Plug–ins. Die Tabelle 3 zeigt detailliert die in der Schnittstelle verfügbaren Funktionen, deren Parameter und Rückgabewerte. Darüber hinaus sind die Funktionen und deren Anwendungsbereiche beschrieben.

Die Klasse Plug–Object

void setGUI(void)
Die Interaktionselemente, die von einem Plug–in bereitgestellt und in die Karte der Anwendung integriert werden, können in dieser Funktion definiert und initialisiert werden. Die Attribute jeder Komponente können zu einem späteren Zeitpunkt in der Erweiterung geändert werden.
void autorunPlugin(void)
Diese Funktion wird ausgeführt sobald eine Szene in die Arbeitsumgebung geladen wird. Dadurch können beispielsweise Manipulationen am Szenengraph oder an der Kamera ohne ein Einwirken des Anwenders vorgenommen werden.
void stopPlugin(void)
Wenn ein Plug–in über den Manager entfernt werden soll, dieses jedoch noch Funktionen ausführt, können diese mit Hilfe der Funktion beendet und das Plug–in auf das Entladen vorbereitet werden.
void getSGItemName(const char *nameOfItem)
Diese Funktion stellt kontinuierlich einen Zeiger auf den Namen des selektierten Knotens des Szenengraphens bereit.
void getBrowserNode(osg::Node *selectedNode)
Mithilfe der Funktion wird der gewählte Knoten des Szenengraphens als Zeiger auf den entsprechenden *osg::Node* bereitgestellt. Die Eigenschaften des Knotens können mit Hilfe dieser Funktion abgefragt oder verändert werden.

void getCamera(Producer::Camera *camera)
Die Anwendung verwendet als Standardkamera die Open Producer Kamera, welche als Zeiger über diese Funktion bereitgestellt wird. Damit stehen dem Plug–in zu jedem Zeitpunkt alle Daten der Kamera zur Verfügung. Dadurch sind beispielsweise Manipulationen am Sichtkörper möglich.
const char* setName(void)
Durch Überschreiben der Funktion kann jede Erweiterung einen eigenen Namen erhalten, der im Kartenreiter innerhalb der Anwendung angezeigt wird.
const char* setVersion(void)
Mit dieser Funktion kann eine Versionsnummer in das Plug–in integriert werden, um den momentanen Entwicklungsstand aufzuzeigen.
const char* setAuthor(void)
Die Funktion bietet die Möglichkeit den Namen des Plug–in Entwicklers festzulegen.
const char* setDescription(void)
Eine Beschreibung der Funktionalität kann mit Hilfe dieser Funktion definiert werden. Eine Kurzbeschreibung der Arbeitsweise der Erweiterung kann das Verständnis für die Plug–in Funktionalität verbessern und das Arbeiten mit der Erweiterung vereinfachen.

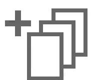
Tabelle 3: Referenz der Schnittstellenklasse Plug_Object

Die zweite Klasse mit der ein Modul auf die Funktionalität der Arbeitsumgebung zugreifen kann, ist die BiLLInterface Klasse. Über diese ist es dem Plug–in möglich Informationen des aktuellen Zustandes der Anwendung abzufragen sowie die erhaltenen Daten zu manipulieren und dadurch Einfluss auf die Basisanwendung zu nehmen. Eine Instanz der Klasse wird im Konstruktor der Plug–in Instanz übergeben und somit ist die Schnittstelle jederzeit verfügbar. Ferner ist eine Bereitstellung von Informationen aus der Hostapplikation zu einem von der Erweiterung definierten Zeitpunkt gewährleistet.

Die Klasse BiLLInterface

fltk::Group * getTab (void) Mit dieser Funktion wird ein Zeiger auf eine Karte in der Arbeitsumgebung übergeben. Diese Karte ist das dem Plug–in zugeteilte Segment innerhalb des Editorfensters, in welche die Interaktionselemente der Erweiterung integriert werden. Es ist somit der Container für alle zu visualisierenden FLTK–Elemente einer Erweiterung.
const char * getSelectedNodeName (void) Die Funktion gibt einen Zeiger auf den Namen des aktuell selektierten Knotens zurück.
osg::Node * getSelectedNode (void) Der Zeiger auf den gewählten Knoten des Szenengraphens wird mit Hilfe dieser Funktion übergeben. Dadurch können Manipulationen am Knoten und dessen Attributen vorgenommen werden. Darüber hinaus ist es möglich von dem übergebenen Knoten ausgehend, die Struktur des Graphens zu durchlaufen und zu manipulieren. Es kann beispielsweise ein Gruppierungsknoten als Elternknoten eingefügt werden.
void getNodePath (osg::NodePath &nodePath, osg::Node *selectedNode) Mit dieser Funktion werden, vom übergebenen Knoten ausgehend, alle übergeordneten Knoten bis zum Wurzelknoten der Szene übergeben. Dadurch kann die Vererbung, welcher der Knoten unterliegt, nachvollzogen werden.
osg::Node* getRootNode (void) In dieser Methode wird ein Zeiger auf den Wurzelknoten des aktuell geladenen Szenengraphens an das Plug-in übergeben.
void setBrowserItem (osg::Node*) Die Funktion ermöglicht das Selektieren eines Knotens im Szenengraph. Dazu wird der Schnittstelle ein Zeiger auf diesen übergeben.
void reloadSceneBrowser (void) Das Aktualisieren des Szenengraphens im Szenenbrowser ermöglicht nach einer Manipulation des Graphen eine korrigierte Darstellung der Szenenstruktur.

void expandSceneGraph (void) Das Einblenden aller Teilgraphen und damit die Darstellung der Datenstruktur der Szene im Browser des Editorfensters ist mit dieser Funktion gegeben.
void collapseSceneGraph (void) Mit Hilfe der Funktion erfolgt das Ausblenden aller Teilgraphen und folglich die Anzeige des Wurzelknotens im Szenenexplorer.
Producer::Camera * getCamera (void) Auf die Open Producer Kamera, die in BiLL als Standardkamera verwendet wird, kann mithilfe der Funktion zugegriffen und diese manipuliert werden.
Producer::CameraConfig * getCameraConfig (void) Einstellungen die das aktuelle Kameramodell verändern, wie beispielsweise die Änderung der Kameraanzahl, sind durch diese Funktion umsetzbar.
Producer::RenderSurface * getRenderSurface (void) Mit dieser Funktion sind Manipulationen an der Projektionsebene möglich. Damit ist beispielsweise die Auflösung oder das Seitenverhältnis des Kamerabildes modifizierbar.
double * getViewFrustum (void) Die Daten des Sichtkörpers können mit Hilfe dieser Funktion abgefragt werden. Durch eine Manipulation dieser Daten kann das View Frustum verändert werden. Dabei werden die folgende Information über dieses zurückgegeben: Links, Rechts, Oben, Unten, Near–Clipping–Plane, Far–Clipping–Plane.
void selectCameraManipulator (const std::string modeName) In der Arbeitsumgebung besteht für den Nutzer die Möglichkeit unter den drei Navigationsvarianten „Drive“, „Ufo“ und „Trackball“ zur Steuerung durch die 3D–Szenen zu wählen. Mit dieser Funktion wird die Steuerungsart festgelegt.
void advancedWindowMode (bool select) Das Aktivieren beziehungsweise Deaktivieren des erweiterten Fenstermodus ist mithilfe dieser Funktion, aus der Erweiterung heraus, realisierbar.



Zusatz 6: Interface Referenz

Tabelle 4: Referenz der Schnittstellenklasse BiLLInterface

Mit Hilfe der Funktionen, die in beiden Klassen implementiert sind, werden die für eine Plug–in Entwicklung nötigen Strukturen bereitgestellt. Dadurch sind die Kommunikationsmöglichkeiten zwischen der Basisanwendung und den Erweiterungen gegeben. Diese Schnittstelle ist ein Teilaspekt bei der Umsetzung der Plug–in Architektur. BiLL ist als dynamisch erweiterbares System konzipiert und stellt Erweiterungsmöglichkeiten erst zur Laufzeit des Systems bereit. Die Umsetzung der Plug–in Verwaltung wird im Folgenden beschrieben.

5.2.2 BiLL: Die Realisierung des Plug–in Managers

Die Arbeitsumgebung BiLL stellt bisher einen festen Umfang an Funktionalität bereit. Dieser soll jedoch dynamisch erweiterbar sein. Aufgrund dessen wird ein Plug–in Manager in die Anwendung eingebunden. Die Verwaltungskomponente wie sie konzeptionell in Abschnitt 4.4.2 beschrieben ist, wird in die Arbeitsumgebung integriert. Bei der praktischen Umsetzung wird das GUI–Framework FLTK© zur Visualisierung der Benutzeroberfläche, wie es ebenfalls die Hostapplikation umsetzt, verwendet. Im Folgenden wird die Realisierung der Plug–in Management–Komponente in der Arbeitsumgebung BiLL dargelegt.

Bei der Realisierung sind vier wesentliche Aufgaben durch den Manager zu erfüllen. Eine erste Aufgabe stellt das Durchsuchen von Verzeichnissen nach potentiellen Plug–ins und einer Tauglichkeitsprüfung derer dar. Daraufhin schließt sich das Laden der Erweiterungen und das damit verbundene allokieren von Speicher an, um somit die Verfügbarkeit des Moduls in der Arbeitsumgebung zu gewährleisten. Eine dritte Aufgabe ist die Übergabe der Kontrolle an die Erweiterung zur Ausführung deren Funktionalität. Die vierte Aufgabe umfasst das Entfernen eines Plug–ins aus der Anwendung und die Freigabe des Speichers.

Der Aufruf des Plug–in Managers erfolgt innerhalb der Bildsprache LiveLab Arbeitsumgebung. In der Menüleiste unter „Extras“ wird der Manager geladen und zeigt sich wie in der folgenden Abbildung.

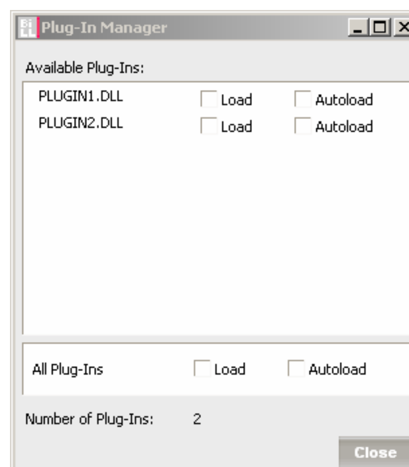


Abbildung 5.1: Plug–in Manager in BiLL

Über die Bedienoberfläche stehen dem Anwender eine Reihe von Optionen zur Verfügung. Innerhalb des Fensters ist es möglich, die darin aufgelisteten Erweiterungen einzeln oder gesamtheitlich zu laden. Darüber hinaus können Plug–ins über den Status „autoload" automatisch beim Programmstart in die Anwendung integriert werden. Die Speicherung der Konfiguration des Managers erfolgt in einer Initialisierungsdatei. Diese wird beim Ausführen der Anwendung ausgelesen und es werden die Module geladen, deren Autostart–Attribut gesetzt ist. Damit ist eine beständige, individuell anpassbare Anwendung konfigurierbar.
Bevor jedoch die Erweiterungen im Plug–in Browser aufgelistet werden, sucht der Manager rekursiv durch die Unterordner der Anwendung nach potentiellen Modulen. Diese sind als dynamisch verknüpfte Bibliotheksdateien (*.dll) in einem Unterordner der Anwendung abgelegt. Die Erweiterungen können direkt im Ordner der Startdatei der Arbeitsumgebung oder in Unterordnern innerhalb des Verzeichnisses abgelegt werden. Nach dem Durchlaufen der Verzeichnisstruktur werden alle Bibliotheksdateien hinsichtlich ihrer Verwendbarkeit in der Anwendung untersucht. Dadurch ist das Laden unvollständiger oder falscher DLL–Dateien ausgeschlossen. Diese Kontrolle erfolgt mit Hilfe einer dem Plug–in zugehörigen Definitionsdatei (*.def). In dieser ist festgelegt, welche Funktionen aus der Bibliotheksdatei exportiert und in die Anwendung integriert werden können. Sind die vom Manager benötigten Funktionen in der Definitionsdatei nicht fixiert, können die Plug–in Objekte nicht instanziiert werden und die Integration wäre nicht möglich. Diese Dateien werden daraufhin vom Manager nicht weiter betrachtet. Die verfügbaren Bibliotheken können nun geladen werden. Um Erweiterungen bereitstellen zu können, steht in der Basisanwendung ein vordefiniertes abstraktes Plug–in ohne erweiternde Funktionalität zur Verfügung. Dieses bildet den Musterrahmen für die verschiedenen Module. Über dieses ist es möglich neue Softwarebausteine zu erzeugen und damit auf Grundlage dieser Plug–in Schablone ein Modul mit erweiterter Funktionalität zu erzeugen. Dies erfolgt nach dem Prinzip des Polymorphismus. Von der Basisklasse können dadurch abgeleitete Klassen gebildet werden, die somit von dieser erben. Aufgrund dessen können Objekte erzeugt werden, die alle Plug–in Objekte sind, jedoch unterschiedlich abgeleitet und spezialisiert sind. Dadurch verkörpern diese, voneinander unabhängige Erweiterungsmodule. Diese können parallel in der Anwendung existieren, wodurch eine Nacheinanderausführung und damit eine Kombination von Funktionen verschiedener Plug–ins gewährleistet ist. Die Abbildung 5.2 zeigt schematisch den Prozess der Erzeugung beziehungsweise der Zerstörung von Plug–in Instanzen.

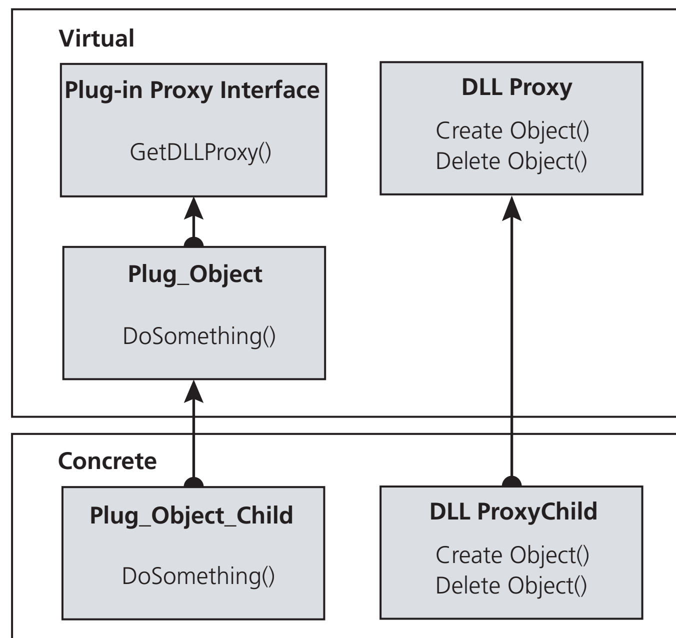


Abbildung 5.2: Skizze der Struktur des Plug-in Konzepts

Eine Schwierigkeit, die sich beim Arbeiten mit dynamischen Bibliotheken ergibt, ist deren Speicherverwaltung. Es ist wichtig, dass die Speicherallokation von der gleichen Stelle ausgeführt wird wie die Deallokation. Wenn dies nicht geschieht, kann das zu fehlerhaften Speicherzugriffen führen, die Systemabstürze zur Folge haben. Wenn beispielsweise eine Erweiterung A von einer Instanz in die Anwendung geladen wird und daraufhin wieder entladen werden soll, muss der Speicher, der mit Erweiterung A verknüpft ist, freigegeben werden. Jedoch hat die ausführende Einheit keinerlei Informationen über die Belegung von Erweiterung A im Speicher und dies würde zwangsläufig zu einer Zugriffsverletzung in diesem führen. Der Grund dafür ist, dass die Zuteilung und die Freigabe des Speichers von Objekt A nicht an gleicher Stelle durchgeführt werden. Aufgrund dessen wird bei der Plug–in Initialisierung eine Instanz erzeugt, welche die Aufgabe der Speicherverwaltung übernimmt. Die Umsetzung dieser Instanz erfolgt nach dem softwaretechnologischen Entwurfsmuster des Proxys. Für jedes Erweiterungsmodul existiert ein individuelles Proxy–Objekt. Dieses ist für die Speicherverwaltung des ihm zugewiesenen Plug–in Objektes zuständig. Dadurch wird die gesamte Speicherverwaltung von einer Klasse ausgeführt, wodurch Speicherverletzungen nicht mehr auftreten. Im Plug–in Manager übernimmt diese Aufgabe die DLL Proxy Klasse. Wenn der Anwender die Erweiterungsbibliotheken lädt, werden Funktionszeiger auf die in der Bibliothek bestehenden Funktionen gespeichert. Über diese werden die Funktionen der Plug–in Bibliothek der DLL Proxy Klasse bekannt gegeben. Eine Instanz dieser Klasse ist für die Instanziierung des ihm zugeordneten Plug-in Objektes verantwortlich. Dazu wird für jede neue Plug–in Instanz (Plug_Object_Child) ein zugehöriges DLL ProxyChild Objekt erzeugt. Dieses Objekt erbt von der Klasse DLL Proxy und ist als Instanz für die Speicherverwaltung der ihm zugeordneten Erweiterung zuständig. Dadurch ist gewährleistet, dass die Speicherverwaltung von nur einer Instanz

durchgeführt wird. Die Klasse Plug–in Proxy Interface ist das Verbindungsglied zwischen der Plug_Object und der DLL Proxy Klasse. Diese verknüpft die Instanz für die Speicherverwaltung mit dem Plug_Object Objekt und durch die Vererbung ebenfalls das Plug_Object_Child Objekt mit dem dazugehörigen DLL ProxyChild.

Da während der Entwicklung der Plug–in Architektur die Dimensionen der möglichen Erweiterungen nicht umfassend absehbar sind, wird die bereitgestellte Funktionalität des Basis Plug–ins über virtuelle Funktionen realisiert. Diese sind in der Plug–in Basisklasse deklariert und zeichnen sich dadurch aus, dass sie in abgeleiteten Klassen überschrieben werden können. Dies bedeutet, dass eine virtuelle Funktion in einer abgeleiteten Klasse nicht nur anders implementiert werden kann, sondern, dass darüber hinaus über einen Zeiger, der auf das abgeleitete Plug–in der erbenden Klasse zeigt, die Methode der abgeleiteten Klasse und nicht die der Basisklasse, aufgerufen wird. In der Umsetzung des Plug–in Managers, wie es die Abbildung 5.2 veranschaulicht, wird die Funktion *doSomething()* in der Klasse Plug_Object aufgerufen, jedoch die Funktionalität, die sich in der Funktion befindet durch die gleichnamige Funktion in der Plug_Object_Child Klasse überschrieben und ausgeführt. Auf diese Weise können verschiedene Funktionen, zur Untersuchung sowohl bildsprachlicher als auch computergrafischer Aspekte, ausgeführt und die Ergebnisse der Methoden in der Hostapplikation umgesetzt werden.

Ein weiterer Vorgang, der innerhalb des Managers ausgeführt wird, ist das Entladen von Plug–ins. Dazu wird das Plug_Object_Child Objekt durch das DLLProxy Child Objekt zerstört. Wenn dies abgeschlossen ist wird die Instanz, der für die Speicherverwaltung verantwortlichen Proxy-Klasse, ebenfalls entfernt, da diese nicht mehr benötigt wird. Ist kein DLL ProxyChild mehr vorhanden, wird die zugehörige Bibliotheksdatei aus dem Speicher gelöscht. Das Plug–in ist damit vollständig aus der Arbeitsumgebung beseitigt.

5.2.3 Die Realisierung eines Plug–ins

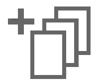
Bei der Entwicklung eines Plug–ins werden primär zwei Technologien verwendet. OpeneSceneGraph™ wird zur Visualisierung der Resultate der Plug–in Funktionalität und FLTK$_{©}$ zur Interaktion des Anwenders mit dem Editor verwendet. Um eine Eingliederung einer Erweiterung in die Oberfläche der Arbeitsumgebung zu ermöglichen, wird dem Modul eine von der Anwendung erzeugte freie Karte zugewiesen. Diese steht dem Plug–in zur Verfügung und grenzt dieses von anderen Bereichen der Benutzeroberfläche ab. Das Codegerüst der Klasse Plug_Object_Child eines Plug–ins wird nachfolgend aufgezeigt. Die Klasse erbt von der Basisklasse Plug_Object und stellt die zentrale Einheit der Plug–in Entwicklung dar.

```
#include "Plug_Object_Child.h"

//Constructor
Plug_Object_Child::Plug_Object_Child(BillInterface* billInterface) :
Plug_Object(billInterface)
{
	_billInterface = billInterface;
}

//Destructor
Plug_Object_Child::~Plug_Object_Child()
{

}

//to set the name of the plug-in
const char* Plug_Object_Child::setName(){
	#ifdef _DEBUG
		name = "Plug-in Name(D)";
	#else
		name = "Plug-in Name(R)";
	#endif DEBUG
	return name;
}

//to set the number of the present version
const char* Plug_Object_Child::setVersion(){
	version = "0.0";
	return version;
}

//to set the name of the author of the present plug-in
const char* Plug_Object_Child::setAuthor(){
	author = "Max Mustermann";
	return author;
}

//to give a short description of the functionality of the plug-in
const char* Plug_Object_Child::setDescription(){
	description = "nothing to say...";
	return description;
}

//in this mehtod it is possible to set up the GUI of the plug-in
void Plug_Object_Child::setGUI(){

	fltk::Group* tab = _billInterface->getTab();
	tab->begin();
		//FLTK-Components
	tab->end();
}
```

Im Konstruktor der Klasse Plug_Object_Child wird ein Zeiger auf die Schnittstelle übergeben, der einen Zugriff des Plug–ins auf die Klasse BillInterface gestattet. Die verfügbaren Funktionen sind von der Klasse Plug_Object abgeleitet und können im Plug-in überschrieben werden. Dadurch können die Attribute des Plug-ins wie Name oder Versionsnummer festgelegt werden. In der Funktion *setGUI(void)* wird mit Hilfe der Schnittstelle ein Zeiger auf eine FLTK–Gruppe übergeben. Diese Gruppe repräsen-

tiert die zugewiesene Karte im Editorfenster der Arbeitsumgebung und kann vom Entwickler durch FLTK–Komponenten ergänzt werden. Auf Basis dieses Codegerüstes und den dazugehörigen Komponenten, Schnittstelle und Manager, ist eine Plug–in Entwicklung durch Drittentwickler möglich. Diese benötigen für die Umsetzung einer Erweiterung den Quellcode der Basisanwendung nicht. Eine Bibliotheksdatei, welche die Basisanwendung beinhaltet, wird für die Kompilierung und Verlinkung von Erweiterungen verlangt und ist gegeben. Diese ist kompiliert und kann nicht verändert werden. Somit ist die Voraussetzung erfüllt, den ursprünglichen Quellcode für die Entwicklung von Erweiterungen nicht modifizieren zu müssen und ermöglicht gleichzeitig die Parallelentwicklungen von Modulen. Diese Funktionalitätserweiterungen können nach der Fertigstellung in die Installationsroutine der BiLL Arbeitsumgebung integriert werden. Diese ermöglicht eine Bereitstellung der Anwendung auf einem Windows® Betriebssystem ohne die Installation von Zusatzsoftware. Alle nötigen Komponenten werden während der Einrichtung auf dem Zielsystem angelegt und können bei Bedarf über eine Deinstallationsroutine wieder entfernt werden. Ein Assistent führt den Anwender durch die Installation der Software. Nach dieser kann die Anwendung über das Desktop–Icon oder den Eintrag im Startmenü ausgeführt werden.

Abbildung 5.3: Installationsroutine der Arbeitsumgebung



Zusatz 7: BiLL Setup

Der Installationsassistent wird mit Hilfe des Open Source Programms Nullsoft Scriptable Install System erzeugt. Der Installationsdatei der Arbeitsumgebung basiert auf einem Skript, welches durch den Kommandozeilen–Compiler des Programms zu einer ausführbaren Datei kompiliert wird. Diese beinhaltet sämtliche Dateien der Bildsprache LiveLab Software sowie das Installationsprogramm und ermöglicht damit eine einfache und komfortable Verbreitung der Anwendung.

5.3 Die Umsetzung des EPK–Plug–ins

Die Implementierung des Plug–ins zur Umsetzung der erweiterten perspektivischen Korrektur setzt sich aus FLTK–Komponenten und den damit verknüpften Funktions-aufrufen, den so genannten Callbacks zusammen. Diese führen Funktionen aus, wenn ein Nutzer eine Aktion über ein Element der Benutzeroberfläche auslöst. Die Implementierung der Funktionen, die durch die Callbacks aufgerufen werden sowie die Realisierung des Verfahrens der EPK sind für eine Umsetzung des Plug–ins notwendig. Darüber hinaus ist der Speichervorgang von Szenen, welche durch die erweiterte perspektivische Korrektur modifiziert wurden, in der Umsetzung der Erweiterung berücksichtigt.

Wie in 5.2.2 erläutert, erbt die Klasse Plug_Object_Child von der Klasse Plug_Object und somit auch die Funktion *setGUI(void)* (vgl. 5.2.1). Durch diese Funktion werden alle im Plug–in integrierten FLTK–Komponenten kontinuierlich in der Arbeitsumgebung visualisiert. Über die Funktion *BiLLInterface::getTab(void)* wird dem Plug–in eine Karte in Form einer *FLTK::Group** übergeben und kann daraufhin mit FLTK–Komponenten besetzt werden. Die übergebene Gruppe ist ein Container und kann mit Hilfe von Untergruppen und Bedienelementen die GUI einer Erweiterung bilden. Die Oberfläche des Plug–ins zur erweiterten perspektivischen Korrektur setzt sich, wie in der nachfolgenden Abbildung erkennbar, aus drei Gruppen innerhalb der Karte zusammen.

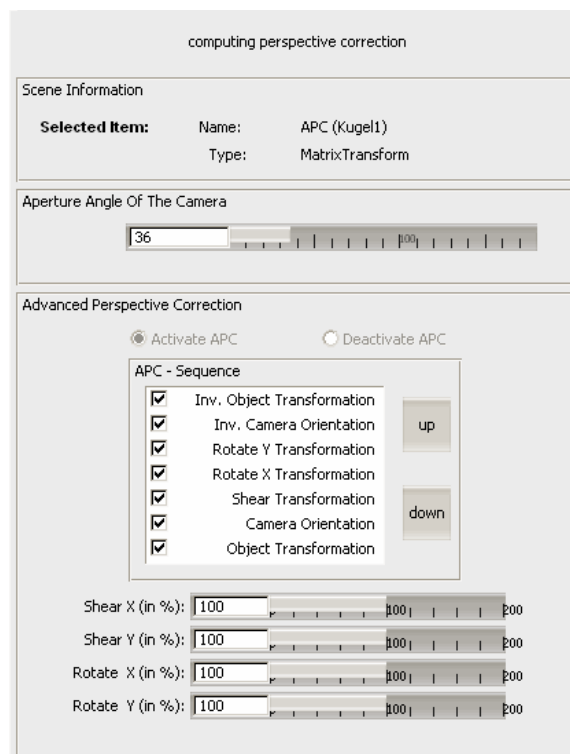


Abbildung 5.4: Benutzeroberfläche des Plug–ins in BiLL

In der ersten Gruppe werden der Name sowie der Knotentyp des aktuell selektierten Objektes visualisiert. Die Gruppe unter dem Namen „Aperture Angel Of The Camera" ermöglicht die Veränderung des Kameraöffnungswinkels der Open Producer Kamera. Die dritte Gruppe dient der Erzeugung sowie der Steuerung der Perspektivkorrektur des gewählten Objektes. Die Aktivierung, wie auch die Deaktivierung über Radio–Buttons, sowie die Festlegung der prozentualen Anteile der einzelnen Faktoren der Transformationsschritte des Verfahrens sind möglich. Dafür stehen dem Anwender vier Schieberegler zur Verfügung. Mit diesen können die Scherungsfaktoren in X– und Y–Richtung, sowie die Rotation um die X– als auch um die Y–Achse eines jeden korrigierten Objektes individuell vom Anwender konfiguriert werden. Der Wertebereich der Schieberegler erstreckt sich von 0 bis 200 Prozent des im Verfahrens errechneten Wertes. Dadurch können die entzerrten Objekte vom Anwender kontextabhängig in die Gesamtszene eingepasst werden. Darüber hinaus sind die Einzelschritte des Verfahrens der EPK in der Oberfläche des Plug–in aufgelistet. Die dargestellten Schritte können gegeneinander getauscht und individuell aktiviert oder deaktiviert werden. Zur Änderung des sequenziellen Ablaufes des Verfahrens sind rechts neben der Auflistung zwei Schaltflächen zur Navigation in die Oberfläche integriert. Ferner erfolgt die Aktivierung beziehungsweise Deaktivierung von Schritten über die jeweils zugehörigen Checkboxen innerhalb der Erweiterung. Dadurch werden die entsprechenden Matrizen nicht in die Berechnung der erweiterten perspektivischen Korrektur einbezogen.
Jede Komponente, die in der *setGUI(void)* Funktion erzeugt wird, kann mit einem Callback–Befehl versehen werden. Dadurch erfolgt bei der Aktivierung einer Oberflächenkomponente durch den Anwender eine definierte Reaktion von Seiten des Programms. Über eine derartige Bindung einer Funktion an eine FLTK-Komponente wird der Kameröffnungswinkel verändert. Dies ist nötig um die Verzerrungen, die bei der Zentralperspektive auftreten und ab einem Öffnungswinkel von ungefähr 60 Grad für den Betrachter sichtbar werden, darzustellen. Dazu werden der horizontale und der vertikale Kameraöffnungswinkel durch eine Manipulation des View Frustum an die Vorgaben, die der Nutzer über den Schieberegler der Anwendung vornimmt, angepasst. Dies erfolgt in dem die Werte für linke, rechte, obere und untere Begrenzung des Sichtkörpers, unter Berücksichtigung des Verhältnisses der Projektionsebene für den jeweiligen Kameraöffnungswinkel neu gesetzt werden. Die für die Aktualisierung des Sichtkörpers benötigten Daten werden über die BiLLInterface Klasse vom Plug–in abgerufen. Die Modifikation des View Frustum erfolgt durch die Neuberechnung der Near–Clipping–Plane. Für die Berechnung des Sichtkörpers werden die Daten ausschließlich der Near–Clipping–Plane fixiert. Entsprechend dem vom Anwender festgelegten Öffnungswinkel wird daraufhin der Sichtkörper neu berechnet. Durch die Ermittlung des Wertes der Near–Clipping–Plane kann der Öffnungswinkel auch bei einem asymmetrischen View Frustum neu festgelegt werden. Die Berechnung der Near–Clipping–Plane erfolgt mit Hilfe der folgenden Funktion:

Komponenten der Berechnung:

Gegeben:

Kameraöffnungswinkel	=	W
Rechtsseitige Begrenzung des Sichtkörpers	=	R
Linksseitige Begrenzung des Sichtkörpers	=	L

Gesucht:

Near–Clipping–Plane	=	N

$$N = \frac{(R - L) * 0{,}5}{\tan \frac{W}{2}}$$

Durch eine Neuberechnung des View Frustum können in der Anwendung Kameraöffnungswinkel zwischen 1 und 179 Grad festgelegt werden. Bei einem Winkel von 120 Grad sind perspektivische Verzerrungen bei sphärischen Objekten wie in Abbildung 5.6 sichtbar.

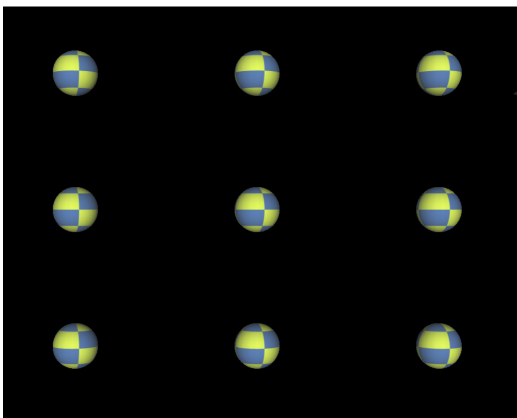


Abbildung 5.5: Szenendarstellung mit 36 Grad Kameraöffnungswinkel

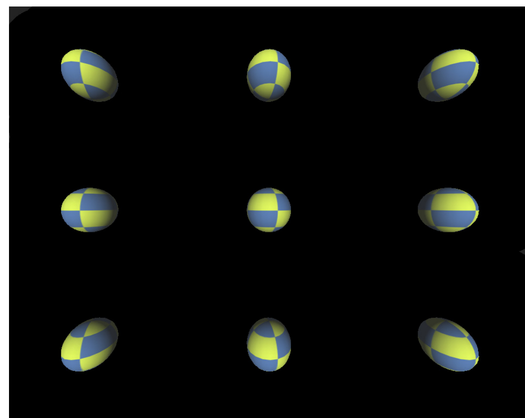


Abbildung 5.6: Szenendarstellung mit 120 Grad Kameraöffnungswinkel

Die Anwendung des Verfahrens der erweiterten perspektivischen Korrektur in BiLL erfolgt auf der Datenstruktur des Szenengraphens. Die geometrischen Veränderungen, die während der Korrektur am Objekt vorgenommen werden, setzen sich aus diversen Transformationsschritten zusammen. Diese werden in einem Transformationsknoten subsummiert und als Elternknoten des selektierten Objektes in den Szenengraphen integriert. Der Transformationsknoten wird bei der Traversierung durchlaufen und ausgewertet. Über die Vererbung werden die Kindknoten aktualisiert und entsprechend der Vorgaben des Transformationsknotens in ihrer Struktur verändert. Damit ist es möglich, Gruppen von Objekten einer Geometrieveränderung zu unterziehen. Weil die perspektivische Korrektur eines Objektes von der Position und Ausrichtung der Kamera abhängt, ist es notwendig, dass der Transformationsknoten

bei jedem Frame aktualisiert wird. Dies erfolgt mit Hilfe eines OpenSceneGraph–Callbacks. Dieser kann als benutzerdefinierte Funktion verstanden werden, die bei jeder Traversierung des Graphens ausgeführt wird. Callbacks können mit individuellen Knoten oder auch Knotentypen verknüpft werden. Bei der Umsetzung der EPK wird ein Callback mit jedem für die Korrektur ausgewählten Geometrieknoten assoziiert. Für die Umsetzung der durch den Callback aufgerufenen Routine werden Daten benötigt, die nicht unmittelbar Bestandteil des selektierten Knotens sind. Die Berechnung der Matrix, die für das Verfahren essentiell ist und dem Transformationsknoten übergeben wird, erfolgt in einem Datenspeicher der Node–Klasse von OSG. Dieser steht in Verbindung zum selektierten Objekt und beinhaltet alle Informationen, die mit dem gewählten Objekt verknüpft werden sollen. Diese Klasse ist abgeleitet von der *osg::Referenced* Klasse des OpenSceneGraph Node Kit. Die Verknüpfung eines Objektes mit einer Data–Klasse erfolgt über einen, den Szenengraphen durchlaufenden, Node Visistor.
Wenn der Visitor während der Traversierung den zu korrigierenden Knoten erreicht wird in der Data–Klasse das Verfahren der erweiterten perspektivischen Korrektur ausgeführt und das Ergebnis, die errechnete Transformationsmatrix, an den Transformationsknoten oberhalb des Geometrieknotens übergeben und damit die Geometrieveränderung am Körper durchgeführt. Die Umsetzung des Verfahrens der erweiterten perspektivischen Korrektur wie sie in 3.3.2 beschrieben und in Abbildung 5.7 dargestellt wird, erfolgt in der BiLL Arbeitsumgebung äquivalent.

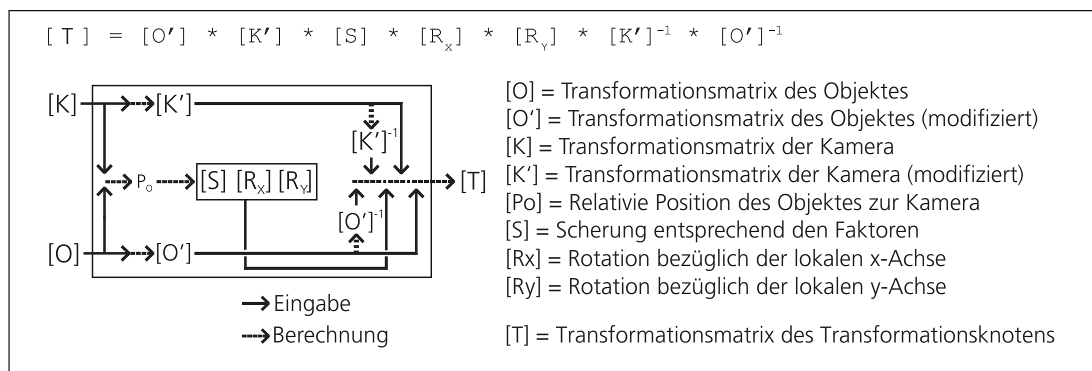


Abbildung 5.7: schematischer Aufbau des Verfahrens der EPK [Zavesky 07]

Zusatz 8: Quellcode der EPK

```
void EPK_Data::updateEPK(){

    //get camera in camera coordinates
    Producer::Matrix::value_type* matrixProd(cam->getViewMatrix());
    [tmpView].set(matrixProd);

    //take the eye coordinates of the camera into world coordinates
    [K] = osg::Matrix::inverse(osg::Matrix(matrixProd));
    //transform the camera matrix for preparing apc
    [K]Trans.set(
      [tmpView](0,0),[tmpView](1,0),[tmpView](2,0),[tmpView](3,0),
      [tmpView](0,2),[tmpView](1,2),[tmpView](2,2),[tmpView](3,2),
      [tmpView](0,1),[tmpView](1,1),[tmpView](2,1),[tmpView](3,1),
      [tmpView](0,3),[tmpView](1,3),[tmpView](2,3),[tmpView](3,3));
```

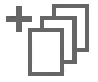
```
    //set world coordinates of the selected object
    worldObj.set(worldObjTmp[0],worldObjTmp[1],worldObjTmp[2],1);

    //get coordinates of the object in camera coordinates
    camCoordObj = [K]Trans * worldObj;
    camCoordObj[1] = camCoordObj[1]*(-1);

    //modify camera matrix
    [K](3,0) = 0.0;
    [K](3,1) = 0.0;
    [K](3,2) = 0.0;
    [K](3,3) = 1.0;

    //invert camera matrix
    [K]Invert.invert_4x4([K]);

    //set object matrix
    [O].set(1,0,0,0,
            0,1,0,0,
            0,0,1,0,
            worldObj[0],worldObj[1],worldObj[2],1);

    //invert object matrix
    [O]Invert.invert_4x4([O]);

    //get coefficient of shear
    shearXFull = (camCoordObj[0]/camCoordObj[1]);
    shearXPercent = -(shearXFull * shearXSlider)/100;
    shearYFull = (camCoordObj[2]/camCoordObj[1]);
    shearYPercent = -(shearYFull * shearYSlider)/100;

    //set shear matrix
    [S].set(1,0,0,0,
            0,1,0,0,
            shearXPercent,shearYPercent,1,0,
            0,0,0,1);

    //get coefficients of rotation
    rotateXFull = (atan(camCoordObj[2]/camCoordObj[1]));
    rotateXPercent = -(rotateXFull * rotateXSlider)/100;
    rotateYFull = (atan(camCoordObj[0]/camCoordObj[1]));
    rotateYPercent = -(rotateYFull * rotateYSlider)/100;

    //set matrix of Rotation around X-axis
    [Rx].set(1,0,0,0,
             0,cos(rotateXPercent),sin(rotateXPercent),0,
             0,-sin(rotateXPercent),cos(rotateXPercent),0,
             0,0,0,1);

    //set matrix of Rotation around Y-axis
    [Ry].set(cos(rotateYPercent),0,sin(rotateYPercent),0,
             0,1,0,0,
             sin(rotateYPercent),0,cos(rotateYPercent),0,
             0,0,0,1);

    matrixList.clear();

    //involve only selected parts of apc calculation
    for (seqIterator = seqList.begin();
         seqIterator != seqList.end();
         seqIterator++){
        fltk::Group* seqGroup = (*seqIterator);
```

```
        fltk::CheckButton* seqButton =
            (fltk::CheckButton*) seqGroup->child(0);

            if(!strcmp((*seqIterator)->label(), "Inv. Object")
                  && invObjButton) matrixList.push_back([O]Invert);
            else if(!strcmp((*seqIterator)->label(), "Inv. Camera")
                  && invCamButton) matrixList.push_back([K]Invert);
            else if(!strcmp((*seqIterator)->label(), "Rotate Y")
                  && rotYButton) matrixList.push_back([Ry]);
            else if(!strcmp((*seqIterator)->label(), "Rotate X")
                  && rotXButton) matrixList.push_back([Rx]);
            else if(!strcmp((*seqIterator)->label(), "Shear")
                  && shearButton) matrixList.push_back([S]);
            else if(!strcmp((*seqIterator)->label(), "Camera")
                  && camButton) matrixList.push_back([K]);
            else if(!strcmp((*seqIterator)->label(), "Object")
                  && objButton) matrixList.push_back([O]);
    }
    [EPK].set(1,0,0,0,
              0,1,0,0,
              0,0,1,0,
              0,0,0,1);

    //compute matrix to transform object
    for (matrixIterator = matrixList.begin();
         matrixIterator != matrixList.end();
         matrixIterator++){
        [EPK].set([EPK] * (*matrixIterator));
    }

    //set matrix to transform node
    matrixTransform->setMatrix([EPK]);
}
```

Sobald die erweiterte perspektivische Korrektur auf einen Körper angewendet wird, erfolgt eine ständige Aktualisierung des zu korrigierenden Objektes. Während der Laufzeit der Anwendung sind die Daten, die vom Nutzer über die Plug–in Oberfläche festgelegt werden und als Basiswerte für die Berechnung der Transformationsmatrix dienen, präsent. Beim Abspeichern der Szene wird die Szenengraphenstruktur in der gespeicherten OSG–Datei abgebildet. Dies beschränkt sich jedoch auf die vorhandenen Knoten der Szene. Die Daten der erweiterten perspektivischen Korrektur werden innerhalb der Data Klasse zwischengespeichert und aktualisiert. Beim Abspeichern einer Szene werden diese nicht in die OSG–Datei exportiert, weil sie kein Bestandteil des Szenengraphens sind. Somit enthält der Transformationsknoten nach dem Speichervorgang lediglich die Informationen der letzten übermittelten Geometrieveränderung und das Objekt, welches zuvor dynamisch der Kameraposition entsprechend angepasst wurde, behält nach dem Laden der zuvor gespeicherten Szene eine feste Geometrie.

Um die durch den Anwender vorgenommenen Einstellungen speichern zu können, wird der Transformationsknoten um die Informationen aus der Data–Klasse erweitert. Dazu wird eine Klasse EPKUserData angelegt. In dieser wird ein OSG–Knoten definiert der als Erweiterung des Transformationsknotens fungiert. Dieser wird in das OpenSceneGraph Node Kit über die OSG–Registrierung eingegliedert. In der Klasse EPKUserData wird festgelegt wie die Daten der EPK in die zu speichernde OSG–Datei

geschrieben und aus ihr gelesen werden. Aus der erzeugten EPKUserData Klasse wird eine Bibliothek erstellt die in der Ordnerstruktur der Arbeitsumgebung abgelegt ist. Dadurch kann eine Szene mit definiertem EPK–Transformationsknoten gespeichert werden. Die benutzerdefinierten Einstellungen stehen somit im Szenengraphen bereit und werden nach dem Laden des Plug-ins in die Arbeitsumgebung automatisch ausgelesen und die Perspektivkorrektur auf das entsprechende Objekt ausgeführt. Die Szene wird daraufhin gleich dem Zustand vor der Speicherung visualisiert.



Zusatz 9: Quellcode EPKUserData

5.3.1 Analyse des Verfahrens

Die praktische Umsetzung des Verfahrens der EPK bestätigt die Aussagen aus Abschnitt 4.5.2. In der Echtzeitumgebung ist eine Anwendung des Verfahrens möglich, jedoch kommt es bei der Ausführung auf Verbundobjekte zu Durchdringungseffekten. Ansatzpunkte diese zu mindern oder zu neutralisieren, sind in diesem Zusammenhang vielfältig. Das Anpassen des Pivotpunktes des zu korrigierenden Objektes im Kontext der Szene, sowie die Ausrichtung der Transformationsschritte der EPK vermögen die Durchdringung zu minimieren.
Im Folgenden werden Untersuchungen in der Arbeitsumgebung vorgenommen. Dabei soll das Verfahren der erweiterten perspektivischen Korrektur bezüglich dessen Anwendbarkeit im Kontext einer Szene analysiert werden. Es wird dabei auf die Rahmenbedingungen, die eine Anwendung des Verfahrens in einer frei navigierbaren Umgebung zulassen, eingegangen. Aussagen, die im Rahmen der Versuche getroffen werden, sind empirisch und bedürfen einer repräsentativen Anzahl von Probanden, um die an dieser Stelle gesammelten Erkenntnisse in weiteren Versuchen zu bestätigen oder aber zu widerlegen.

Versuche

In den Versuchen wird die erweiterte perspektivische Korrektur auf ein Objekt, welches Teil eines Verbundobjektes ist, ausgeführt. Die Szene, die diesen Untersuchungen zugrunde liegt ist eine Wand an der ein Hohlzylinder angebracht ist. Dieser ist mit vier Ausbuchtungen versehen. Es erfolgen mehrere Durchläufe mit der gleichen Versuchsanordnung. Dabei werden zwei Modifikationen am Algorithmus vorgenommen, untersucht und miteinander verglichen.

Versuchsaufbau

Die Szene besteht aus einer Kamera und zwei orthogonal zur Bildebene positionierten kubischen Objekten. An diesen Körpern ist jeweils ein Hohlzylinder, mit vier am Mantel befindlichen Prismen, ausgerichtet. Diese sind so positioniert, dass sie mit dem Kreuz, welches ein Teil der Texturierung der Wand ist, übereinstimmen. Der Körper schließt direkt an das jeweilige kubische Objekt an, ohne dieses zu durchdringen. Die Kamera befindet sich unterhalb (Position bezüglich der Y–Achse) des Pivotpunktes des zu korrigierenden Hohlzylinders. Der Kameraöffnungswinkel beträgt 135 Grad. Während

des Versuches wird die Kamera parallel zur Z–Achse bewegt. Die Ausrichtung dieser ist in positiver Z–Richtung bleibt in den Versuchen unverändert. Abbildung 5.8 skizziert den Aufbau des Versuchs. Eine räumliche Darstellung dessen zeigt Abbildung 5.9, wobei die Kameraposition ungleich der Position im Versuch ist. Sie ermöglicht lediglich eine detaillierte Sicht auf die Versuchsanordnung.

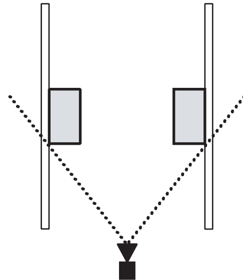


Abbildung 5.8: Skizze Versuchsaufbau

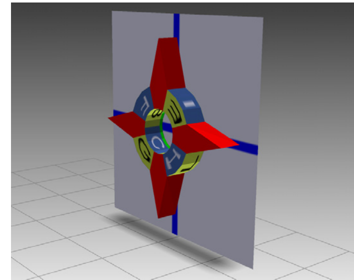


Abbildung 5.9: 3D–Darstellung Versuchsaufbau

Der Versuch umfasst drei identische Kamerafahrten. Im ersten Durchlauf wird der Algorithmus ohne Anpassungen ausgeführt. Für den zweiten Zyklus wird der Pivotpunkt des zu korrigierenden Objektes versetzt. Dazu erfolgt eine Translation des Punktes aus dem Zentrum des Objektes entlang der X–Achse an den kubischen Körper heran. Dadurch befinden sich die Pivotpunkte von Körper und kubischen Objekt auf einer Position im Raum. Im dritten Versuch bleibt der versetzte Pivotpunkt bestehen und zusätzlich werden die Rotations- und die Scherungsfaktoren entsprechend einer wahrnehmungskonformen Darstellung, des aus korrigiertem Hohlzylinder und Wand bestehenden Verbundobjektes, angepasst. In der Ausgangssituation jedes Versuches zeigt sich die Szene in einer Monoperspektive, wie die folgende Abbildung verdeutlicht. In dieser ist die perspektivische Verzerrung des Hohlzylinders zu erkennen. Zu Beginn der Versuche wird der Hohlzylinder mit Hilfe der Benutzeroberfläche des Plug–ins in einer Binnenperspektive dargestellt und daraufhin die Kamerafahrt begonnen.

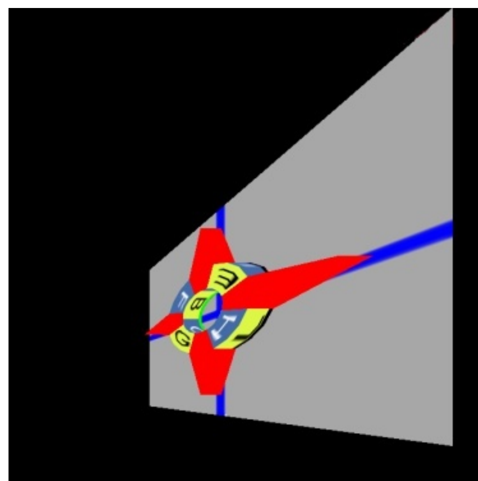


Abbildung 5.10: Monoperspektivische Darstellung der Versuchsszene

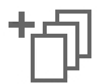
1

2

3

4

Multiperspekive	Multiperspektive mit translatierten Pivotpunkt	Multiperspektive mit translatierten Pivotpunkt und variablen Scherungs– und Rotationsfaktoren



Zusatz 10: Videos zu den Versuchen

Abbildung 5.11: Bildserien der Versuchsreihe

Nachdem bisher auf den Aufbau und Ablauf des Versuchs eingegangen wurde, stehen nachfolgend die Ergebnisse der Versuchsreihe im Mittelpunkt. Die Abbildung 5.11 zeigt in jeder Spalte eine Bildserie mit Ausschnitten aus den jeweiligen Durchläufen des Versuchs. Die in Reihe Eins der Bildserie befindlichen Abbildungen zeigen jeweils das an der Wand befindliche Objekt in einer Binnenperspektive. Die Darstellungen in den Zeilen Zwei bis Vier zeigen Ausschnitte aus der jeweiligen Kamerafahrt während des Versuchs. Die linke Spalte zeigt die Ergebnisse des ersten Durchlaufes und damit die Anwendung der erweiterten perspektivischen Korrektur ohne Veränderungen am Algorithmus. Die Ergebnisse, welche mit der Verschiebung des Pivotpunktes erzielt wurden, sind in der mittleren Spalte visualisiert und in der rechten Spalte sind die Ergebnisse mit versetztem Pivotpunkt und angepassten Scherungs– und Rotationsmatrizen zu entnehmen.

Ergebnisse

In den durchgeführten Versuchen erfolgt eine Bewegung der Kamera parallel zur Z–Achse. Die perspektivischen Verzerrungen sind an dem Hohlzylinder in der monoperspektivischen Darstellung zu erkennen. Im Gegensatz dazu, ist der korrigierte Körper in seinem Erscheinungsbild wahrnehmungskonform. Jedoch erfolgt eine Durchdringung von Objekt und Wand an der dieses positioniert ist. Dieser Effekt verstärkt sich je mehr sich der Körper von der geometrischen Mitte des Bildes entfernt. In der Versuchsreihe entsteht dies durch die Bewegung der Kamera in die Szene hinein. Die Durchdringung entspricht nicht der menschlichen Wahrnehmung, da sich zwei im Erscheinungsbild feste Körper in der Realität nicht durchdringen können. Das unveränderte Verfahren der EPK kann nicht mit dem Ziel der Wahrnehmungskonformität bei Verbundobjekten angewendet werden. Der zweite Versuch zeigt bereits in der Ausgangssituation (Abbildung Eins der Bildserie) eine deutliche Minderung der Durchdringung. Diese ist durch den Versatz des Pivotpunktes erfolgt. Das Eindringen des Körpers in die Wand erfolgt jedoch während der Kamerabewegung in der Szene. Im dritten Durchlauf ist keinerlei Durchdringungseffekt mehr sichtbar. Durch die Anpassung der Parameter ist ein wahrnehmungskonformes Abbild des Körpers vorhanden. Diese Korrektur erfolgt durch die Anpassung der Rotationsfaktoren während der Kamerabewegung. Eine Modifikation der Scherfaktoren ist in diesem Zusammenhang vernachlässigbar.

Durch die Veränderung der Rotationsfaktoren wird jedoch das Objekt in seiner Lage im Raum und damit auch bezüglich des Betrachters verändert. Dies führt bei nicht sphärischen Objekten zu einer für den Betrachter sichtbaren Veränderung der Orientierung des Körpers in der Szene. Bei der untersuchten Konstellation der Objekte in der Szene erfolgen keine Durchdringungseffekte. Jedoch ist die Orientierung des Körpers ungleich der Ursprünglichen. Dies wird in den Versuchen durch die roten Ausbuchtungen des Objektes ersichtlich, die nicht mit den blauen Kontroll-

markierungen der Wand übereinstimmen und somit einen Unterschied zur ursprünglichen Szene aufweisen.

Evaluierung

Die Umsetzung der erweiterten perspektivischen Korrektur in der Arbeitsumgebung Bildsprache LiveLab ermöglicht eine Anwendung des Verfahrens in frei navigierbaren, dreidimensionalen, virtuellen Welten. Selbst bei großem Kameraöffnungswinkel werden sphärische Objekte und Ummantelungen durch geschlossen gekrümmte Flächen wahrnehmungskonform abgebildet. Unter dem Gesichtspunkt der Kognitionskonformität ist der Einsatz der EPK dennoch nur mit Einschränkungen gegeben. Die Regel der Singularität nach GROH ist in BiLL gültig, weil die Ausführung einer Perspektivkorrektur bei Verbundobjekten zu Durchdringungseffekten führt. Diese entsprechen nicht der menschlichen Wahrnehmung und sollten aufgrund dessen vermieden werden. Ausnahmen für diese Regel sind sphärische Objekte, sowie Körper, die in mindestens einer Dimension sphärische Form aufweisen. Die Körper, welche eine derartige Geometrie aufweisen, unterliegen jedoch weiteren Einschränkungen für die Anwendung einer Perspektivkorrektur auf Verbundobjekte. Diese Körper müssen eine stetige Oberflächenstruktur sowie Texturierung aufweisen, weil sonst der Betrachter aufgrund dieser Objektattribute erkennen würde, dass eine veränderte Objektorientierung vorliegt und damit die Abbildung als nicht mehr wahrnehmungskonform empfindet. Wenn diese Einschränkungen erfüllt sind, ist eine Anwendung der erweiterten perspektivischen Korrektur ebenso bei Verbundobjekten möglich und sinnvoll.

6. Zusammenfassung

Nachfolgend wird in diesem Kapitel eine Zusammenfassung der in dieser Arbeit behandelten Themen und der verwirklichten Ziele gegeben. Dabei wird zunächst auf den Inhalt der Arbeit eingegangen (6.1) und ein Fazit aus den erreichten Resultaten gezogen (6.2). Abschließend werden weiterführende Forschungsansätze, die sich aus dieser Arbeit ergeben, vorgestellt (6.3). Diese beziehen sich sowohl auf die Plug-in Architektur, als auch auf das Verfahren der erweiterten perspektivischen Korrektur in Echtzeitanwendungen.

6.1 Inhalt

Nach einführenden Worten und der Vorstellung relevanter Begriffe aus den Bereichen Bildsprache, Computergrafik und Softwaretechnologie wurden Klassenbibliotheken, die Grundvoraussetzung für die theoretischen Überlegungen sowie praktischen Realisierungen waren, vorgestellt und betrachtet.
Daraufhin erfolgte die Darlegung verwandter Arbeiten, mit Fokus auf Komponenten- und Plug–in Architekturen sowie die Abgrenzung der beiden Begrifflichkeiten voneinander. Außerdem wurden zwei Ansätze zur Reduktion von Weitwinkelverzerrungen bei computergrafischer Zentralprojektion aufgezeigt. Dies ist zum einen die Erzeugung von multiperspektivischen Abbildungen durch den Einsatz von mehreren Kameras in einer Szene und zum anderen die Realisierung von Binnenperspektiven durch geometrische Veränderungen an Objekten mit Hilfe des Verfahrens der erweiterten perspektivischen Korrektur. Mit beiden Methoden ist die Erzeugung von multiperspektivischen Abbildungen aus dreidimensionalen Szenen umsetzbar.
In der nachfolgenden Synthese wurden Rahmenbedingungen für Echtzeitanwendungen im Allgemeinen sowie im Bezug auf die Software Bildsprache LiveLab untersucht. Des Weiteren wurden die kontextabhängigen, als auch die softwaretechnologischen Anforderungen an die Echtzeitanwendung BiLL analysiert. Ziel dieser Untersuchungen war die Herbeiführung einer Entwurfsentscheidung für eine Architektur zur möglichst einfachen Erweiterbarkeit von BiLL durch Drittentwickler. Der Entscheidung zugunsten einer Plug–in Architektur folgte eine detaillierte Konzeption zur Analyse und zum Entwurf der Arbeitsumgebung. Darüber hinaus ist eine Projektierung für die Integration des EPK–Verfahrens als Erweiterung der Basisanwendung entwickelt worden. Darauf aufbauend erfolgte die praktische Umsetzung der Architektur in der Arbeitsumgebung. Diese ist in der Arbeit erläutert und dokumentiert worden. Es folgte die Beschreibung von durchgeführten Versuchen zur Klärung der bildlichen Eigenschaften dynamischer multiperspektivischer Szenen, welche mit der vorhergehend beschriebenen Plug–in Architektur in BiLL erstellt wurden.

6.2 Fazit

Unter der Berücksichtigung von möglichen Anwendungsfällen, sowie softwaretechnologischen Rahmenbedingungen erfolgte die Umsetzung einer Plug–in Architektur auf deren Basis eine Weiterentwicklung im Bereich der dialogorientierten beziehungsweise wahrnehmungskonformen Computergrafik möglich ist. Die Umsetzung von praktischen Ergebnissen im Kontext der Bildsprache in Form von Plug–ins durch Drittentwickler, kann ohne eine intensive Einarbeitung in die Funktionsweise der Basisanwendung erreicht werden. Die Struktur der Schnittstelle und die dazugehörigen Referenzdokumente bieten die nötigen Voraussetzungen für die Plug–in Entwicklung. Weitere Aspekte der Arbeitsumgebung bleiben vor dem Entwickler verborgen. Die Umsetzung einer Erweiterbarkeit der BiLL Software wurde mit der gegebenen Architektur erfüllt. Darüber hinaus ist eine Parallelentwicklung von Plug–ins möglich.
Ein zweites zentrales Resultat der vorliegenden Arbeit ist die Portierung des Verfahrens der erweiterten perspektivischen Korrektur in die Arbeitsumgebung. Die Integration in die Anwendung erfolgt in Form eines Plug–ins und zeigt damit zugleich die Funktionstüchtigkeit der Plug–in Architektur auf. Ferner wird die Anwendbarkeit des Verfahrens in einer Echtzeitumgebung gezeigt. Die Umsetzung erfüllt die Anforderungen, die an die Applikation und die Erweiterung gestellt werden. Die umgesetzte Erweiterung ermöglicht die Durchführung von Versuchen, bei deren Auswertung neue Erkenntnisse für die Regeln zum Einsatz der erweiterten perspektivischen Korrektur und damit letztendlich auch für die Anwendung von Multiperspektive zur Erhöhung der Wahrnehmungskonformität gewonnnen werden können. Insbesondere die Verschiebung des Pivotpunktes und die Anpassung des Verfahrens an einen gegebenen Szenenkontext tragen zur Bestätigung der von GROH postulierten Regeln für den Einsatz von Binnenperspektiven in Abbildungen bei. Denn es wurde gezeigt, dass die Anwendbarkeit der erweiterten perspektivischen Korrektur gegeben ist, jedoch die Regel der Singularität (vgl. [Groh 05]) gewahrt bleiben muss, weil die sonst auftretenden Durchdringungseffekte für den Betrachter nicht wahrnehmungskonform sind.
Die Erweiterung im Zusammenhang mit der Plug–in Architektur ermöglichen es die bildstrukturellen Konzepte der Renaissancekünstler in die dreidimensionale Computergrafik zu übertragen und in einer frei navigierbaren virtuellen Welt zu nutzen.

6.3 Ausblick

Dieser Abschnitt zeigt mögliche Forschungsansätze auf, die sich aus dieser Arbeit ergeben. Weiterhin werden Bereiche konkretisiert, in denen an die bestehende Diplomarbeit angeknüpft werden kann. Abschließend werden weiterführende Lösungsansätze vorgestellt und Forschungsfelder im Bereich der Erzeugung multiperspektivischer Abbildungen aufgezeigt.

6.3.1 Weiterentwicklung der Plug–in Architektur

Die in dieser Arbeit entworfene Plug–in Architektur kann zur Erzeugung und Integration von Erweiterungen verwendet werden. Die Schnittstelle ist für die Entwicklung von Plug–ins für das bestehende Softwaresystem konzipiert. Wenn dieses jedoch verändert wird, muss eine Aktualisierung der Schnittstelle erfolgen. Darüber hinaus sind die bereits bestehenden Module in den Aktualisierungsvorgang involviert, die den modifizierten Teil der Schnittstelle nutzen. Diese Erweiterungen müssen daraufhin ebenfalls angepasst werden. Bei der Verbesserung des Kameramodells, beispielsweise durch das Hinzufügen von Postprocessing–Unterstützung, ist eine Überarbeitung der Plug–in Architektur notwendig.

6.3.2 Optimierung des Plug-ins

Das Plug-in, welches im Zuge der Arbeit erstellt wurde, ist für die Untersuchung der erweiterten perspektivischen Korrektur in einer interaktiven Umgebung nutzbar. Die Versuche, die mit dem Modul in der Arbeitsumgebung vorgenommen wurden, sind nicht repräsentativ und sollten daher mit einer aussagekräftigen Anzahl an Probanden wiederholt werden. Darüber hinaus ist das Plug–in als Hilfsmittel für die weitere Untersuchung und Erforschung des Verfahrens dienlich. Bei einer sehr spezifischen Analyse ist eine Weiterentwicklung der Anwendung jedoch notwendig. Zum einen kann dies durch die Anpassung der Bedienoberfläche an einen gegebenen Kontext erfolgen, zum anderen durch die Einführung von zusätzlichen GUI–Elementen, um eine differenziertere Ausführung des Algorithmus zu ermöglichen. Bei jeder Änderung ist zusätzlich eine Dokumentation der Modifikationen zu erstellen.

6.3.3 Lösung noch bestehender Defizite

Wie die Arbeit gezeigt hat, ist die Portierung des Verfahrens in eine Echtzeitumgebung möglich. Die Anwendung der Perspektivkorrektur unterliegt in BiLL den gleichen Beschränkungen wie in Animationen und statischen Bildern. Daher ist eine Weiterentwicklung des Verfahrens der erweiterten perspektivischen Korrektur notwendig, wenn mit Hilfe dieser, Verbundobjekte manipuliert und die derzeit auftretenden Durchdringungseffekte vermieden werden sollen. Dies kann nur unter Einbeziehung der nicht korrigierten Bestandteile des Verbundobjektes oder der Szenenstruktur im Ganzen erfolgen. Die Entwicklung eines Verfahrens der kontextsensitiven perspektivischen Korrektur ist dafür erforderlich. Im Fokus der Betrachtung ist dabei die Einbeziehung der nicht manipulierten Szenenkörper mit denen das korrigierte Objekt ein Verbundobjekt bildet.

Durch das Verschieben des Pivotpunktes und eine dynamische Anpassung der Rotationsfaktoren, werden die korrigierten Objekte in der Abbildung einer Szene ohne eine Durchdringung von anderen Körpern dargestellt. Wie die Versuche in 5.3.1 gezeigt haben, ist die Anwendung der EPK unter dem Aspekt der Wahrnehmungskonformität dennoch nicht möglich, weil das korrigierte Objekt eine falsche Orientierung im Raum aufweist. Eine Erweiterung des Verfahrens der

Perspektiv-korrektur um eine weitere Rotation muss erfolgen. Die Voraussetzung für diesen weiteren Verfahrensschritt ist die erreichte Kollisionsfreiheit nicht zu widerrufen. Die Rotation wird mit dem Ziel ausgeführt, dass Objekt in eine wahrnehmungskonforme Lage im Raum zu versetzen. Für die Durchführung der Objekttransformation ist es erforderlich eine Rotationsachse zu ermitteln. Zur Bestimmung dieser ist es notwendig die Gerade im Raum zu ermitteln, die in der Abbildung der Szene vom verschobenen Pivotpunkt des zu korrigierenden Objektes ausgehend, in den Fluchtpunkt führt. Die Rotation des Körpers muss um die Achse erfolgen die Orthogonal zu der eben beschriebenen Gerade steht. In den Versuchen in 5.3.1 wird dies zu einer Rotation des Objektes an der Wand führen, ohne jedoch in diese einzudringen. Die nachfolgende Abbildung verdeutlicht die zusätzliche Rotation im Verfahren einer kontextsensitiven perspektivischen Korrek-tur.

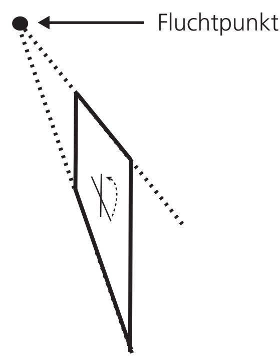


Abbildung 6.1: Zusätzliche Rotation im Verfahren zur Perspektivkorrektur

6.3.4 Aufbauende Forschungsansätze

Veränderung der Sichtbarkeit

Die Erzeugung von Binnenperspektiven in computergrafischen Abbildungen erfolgt in der vorliegenden Arbeit durch eine geometrische Veränderung des Objektes. Dies kann bei Verbundobjekten zu Durchdringungen der Objekte untereinander führen. Diese können jedoch durch eine Veränderung der Sichtbarkeiten in der Abbildung der Szene verhindert werden.

Verdeckung entsteht wenn mehrere Objekte bei der Abbildung einer 3D–Szene auf eine zweidimensionale Ebene eine Projektionsäquivalenz aufweisen. Für den Betrachter ist das zur Kamera am nächsten liegende Objekt sichtbar. Die Verdeckungsreihenfolge kann jedoch über einen Tiefensortieralgorithmus festgelegt werden. In diesem werden die Abstände zwischen Kamera und den einzelnen Objekten gespeichert. Dabei sollte der umzusetzende Algorithmus auf einem objektorientierten Verfahren beruhen, denn die Festlegung der Reihenfolge auf Basis der Objekte der Szene ist komfortabler umsetzbar als eine auf dem Z–Buffer basierende pixelorientierte Variante eines Algorithmus. Zur Lösung des Durchdringungseffektes wird die Verdeckungsreihenfolge gegenüber der Allgemeingültigen verändert und damit festgelegt, dass ein korrigiertes

Objekt die Körper verdeckt, in die es zuvor eingedrungen war. Somit ist für den Betrachter kein Eindringen eines Körpers in einen anderen erkennbar. Wenn beispielsweise in der Versuchsreihe (vgl. Abschnitt 5.3.1) die Verdeckungsreihenfolge von Wand und Hohlzylinder invertiert wird, erscheint das korrigierte Objekt immer vor der Wand und dringt für den Betrachter nie in diese ein. Durch eine vom Entwickler festgelegte Reihenfolge in einem Tiefensortieralgorithmus kann darüber hinaus die Verdeckung der weiteren Szenenbestandteile kontrolliert werden. Damit kann sichergestellt werden, dass sich die Manipulation der Verdeckung lediglich auf die beteiligten Verbundobjekte beschränkt.

Die Kollisionsdetektion

Die Kollisionsdetektion ist ein möglicher Ansatz zur Verhinderung von Durchdringungseffekten bei der Anwendung der erweiterten perspektivischen Korrektur. Dieser wird bei einem möglichen Eindringen von Körpern, die bei der Geometrieveränderung des zu korrigierenden Objektes stattfindet, angewendet. Vor einer tatsächlichen Durchdringung wird die Kollisionsdetektion ausgeführt. Diese verschiebt das Objekt entsprechend festgelegter Vorgaben. Dadurch wird ein Eindringen des Körpers vermieden. Dieser Ansatz hat den Vorteil, dass ein solches Verfahren nur in den Fällen einer tatsächlichen Kollision Anwendung findet. Jedoch ist für die Perspektivkorrektur die performante Kollisionserkennung unter Zuhilfenahme von Bounding Volumen ungeeignet (vgl. [Mezger 01]).
Generell erfolgt die Kollisionsdetektion in der Computergrafik bei dem Aufeinandertreffen von zwei Objekten, wobei mindestens eine dieser beiden Körper in Bewegung ist. Das Zusammentreffen der beiden Körper beschränkt sich auf einen kurzen Zeitraum, bevor sich die Objekte voneinander entfernen. Aufgrund dessen kann eine Kollisions-erkennung mit Bounding Volumen umgesetzt werden, weil der Kollisionszeitraum zu klein ist, um vom Auge detailliert wahrgenommen zu werden. Der Betrachter erkennt eine Kollision der Objekte obwohl nur eine Kollision der Bounding Volumen stattfindet. Für das Verfahren der erweiterten perspektivischen Korrektur ist es notwendig ein sowohl performantes als auch genaues Verfahren zur Kollisionsdetektion zu ent-wickeln, weil die Objekte durch das Verfahren neu positioniert werden und trotzdem für eine wahrnehmungskonforme Darstellung in dem Gesamtkontext der Szene eingebettet bleiben müssen.

Multiperspektivische Abbildungen durch mehrere Kameras

Ein weiterer Ansatz zur Erzeugung multiperspektivischer Abbildungen wird mit Hilfe mehrerer Kameras realisiert. Dazu werden bei der Bilderzeugung mehrere Linearperspektiven generiert. Eine Hauptkamera begrenzt die Abbildung durch die Definition des Sichtkörpers. Darüber hinaus wird für jedes Objekt, welches in einer Binnenperspektive visualisiert werden soll, eine Kamera erzeugt. Mit dieser wird das Objekt in der Frontalansicht dargestellt. Zur Erzeugung einer Abbildung wird mit jeder definierten

Kamera ein Bild erzeugt und unter Berücksichtigung von Abstraktionsregeln und benutzerdefinierter Vorgaben die Einzelbilder zu einer Abbildung zusammengefügt. Bei diesem Ansatz der Fusion von Einzelbildern ergeben sich Schwierigkeiten bezüglich der Verdeckungsreihenfolge sowie der Licht– und Schattenberechnung (vgl. [Agrawala et al. 2000]).

Für die Erzeugung von Binnenperspektiven wie sie vergleichbar durch die erweiterte perspektivische Korrektur erfolgt, können Regeln festgelegt werden, die eine Erstellung einer Abbildung aus mehreren Einzelbildern vereinfacht. Im Kontext der Reduzierung von Weitwinkelverzerrungen möchte der Betrachter die Sichtposition bei der Erzeugung von Binnenperspektiven beibehalten. Auch bei einer Bewegung durch den Raum bleibt dieser Zustand äquivalent. Die einzelnen Kameras unterscheiden sich damit lediglich durch ihre Orientierung im Raum. Dies vereinfacht die Problemstellung der Verdeckung. Die Hauptkamera legt die Reihenfolge der einzelnen Teilbilder über die Tiefeninformationen der Szene fest. In der Abbildung 6.2 sind zwei gerichtete Kameras, zur Darstellung der zwei Körper in einer Binnenperspektive, ausgerichtet. Diese befinden sich an derselben Position im Raum wie die Hauptkamera. Diese widerum bildet die nicht manipulierten Teile der 3D–Szene ab und definiert darüber hinaus die Visualisierungsabfolge, indem sie den Abstand zwischen ihr und den korrigierten Objekten berechnet. Über diese Informationen wird die Verdeckungsreihenfolge festgelegt. Das Ergebnis des Verfahrens für den in Abbildung 6.2 skizzierten Szenenaufbau ist in Abbildung 6.3 dargestellt.

In diese Methodik können zusätzlich vom Betrachter definierte Vorgaben eingebracht werden. Dies würde eine Realisierung in einer Echtzeitumgebung vereinfachen und aufwendige Berechnungen für Verdeckungsproblematiken an den Betrachter übertragen, der diese effizienter lösen kann.

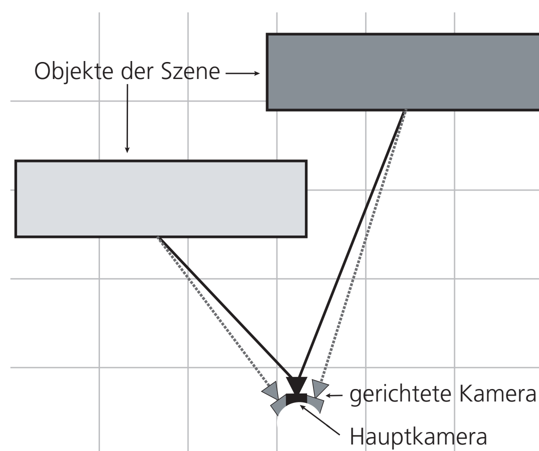


Abbildung 6.2: Multiprojektion mit mehreren Kameras (Vogelperspektive)

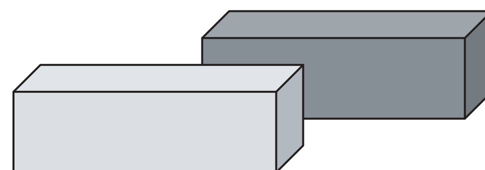


Abbildung 6.3: Abbildung mit korrekter Tiefensortierung

A Glossar

API	Siehe Programmierschnittstelle
Binnenperspektive	Es wird von einer Binnenperspektive gesprochen, wenn sich Objekte nicht dem globalen Perspektivsystem unterordnen, sondern in einer abweichenden Perspektive dargestellt werden.
Bounding Volumen	In der Computergrafik ist das Bounding Volumen eine einfache geometrische Form oder ein Körper, welcher eine komplexe Form oder einen Körper umschreibt. (vgl. [Foley et al. 94]).
Entwurfsmuster	Ein Entwurfsmuster (engl. design pattern) beschreibt eine bewährte Schablone für ein Entwurfsproblem. Es stellt damit eine wiederverwendbare Vorlage zur Problemlösung dar. Entstanden ist der Ausdruck in der Architektur, von der er für die Softwareentwicklung übernommen wurde. (vgl. [Gamma et al. 04])
Monoperspektive	Monoperspektive ist das Bild einer dreidimensionalen Szene in der alle Objekte dem perspektivischen Gesamtsystem unterworfen sind.
Multiperspektive	Multiperspektive ist das Abbild einer räumlichen Szene in dem ein oder mehrere Objekte in einer Binnenperspektive dargestellt werden.
Node Kit	Das Node Kit (dt. Knoten–Bausatz) ermöglicht die Zusammenfassung einer Menge von Knoten in einem Knoten. Dieser kann als normaler Knoten betrachtet werden, welcher einen gesamten Untergraphen verbirgt. (vgl. [Seewald 04])
Parallelprojektion	Parallelprojektion ist eine Form der Projektion. Dabei wird die Abbildung eines Punktes des Raumes an der Stelle dargestellt, an dem der zur Projektionsrichtung parallel verlaufende und durch den Ausgangspunkt gehende Projektionsstrahl die Projektionsebene schneidet. (vgl. [Pareigis 90]).

Perspektive	Dieser aus dem lateinischen stammende Begriff der Perspektive bedeutet sinngemäß „ sich von einem bestimmten Standpunkt aus ergebende Sichtweise“. Für die darstellende Geometrie und damit für die vorliegende Arbeit definiert sich der Begriff als Blickrichtung in eine Szene hinein. (vgl. [Angel 90])
Pivotpunkt	Es ist ein zu einem Objekt zugehöriger Punkt, um den die Rotation und die Skalierung ausgeführt wird. Dieser ist standardmäßig im Zentrum des Bounding Volumens eines Objektes lokalisiert. (vgl. [Foley et al. 94])
Plug–in	Der Begriff Plug–in kann sich auf eine Hardware– aber ebenso auf eine Softwarekomponente beziehen. Generell stellt ein Plug–in eine Funktionserweiterung dar, die in ein bestehendes System integriert werden kann. Bei Software handelt es sich um Module, die eine Erweiterung der ursprünglichen Funktionalität darstellen. (vgl. [Marquardt und Völter 02])
Polymorphismus	Polymorphismus ist ein Konzept der objektorientierten Softwareentwicklung. Dabei besteht die Möglichkeit, Methoden mit gleichem Namen für unterschiedliche Objekte zu implementieren. Diese Methoden können in verschiedenen Klassen unterschiedliches Verhalten aufzeigen. Welche Handlungsweise ausgeführt wird, entscheidet sich zur Laufzeit und ist abhängig vom ausführenden Objekt. (vgl. [Middendorf 02]).
Programmierschnittstelle	*Die Programmierschnittstelle (Application Programming Interface, API) definiert in ihrer Syntax und Semantik die Funktionen des Betriebssystems in Form von Systemdiensten (system services). Diese Systemdienste stehen für den Programmierer einer Anwendung im Allgemeinen in Funktionsbibliotheken […] für die jeweils benutzte Programmierumgebung bereit und sind zum Teil standardisiert* *([Schneider et al.01])*

Projektion	Die geometrische Projektion beschreibt das Abbildungsverfahren, welches Punkte im Raum auf eine festgelegte zweidimensionale Ebene abbildet. Die für die Computergrafik bedeutendsten Projektionen sind die Parallelprojektion und die Zentralprojektion. (vgl. [Pareigis 90])
Shader	Shader, auch als Schattierer bezeichnet, sind Hardware- oder Softwaremodule, die bestimmte Renderingeffekte bei der 3D–Computergrafik implementieren. Aus technischer Sicht bezeichnet der Shader den Teil von einem Renderer, der für die Ermittlung der Farbe eines Objektes zuständig ist. Der Begriff Shader wird sowohl für Hardware–Shader als auch für die darauf laufenden Programme selbst verwendet. (vgl. [Shreiner 05])
Systemraum	Der Systemraum ist nach [Panofsky 85] das primäre Gestaltungsprinzip der Renaissance und der nachfolgenden Epochen. Die Bildkomposition unterliegt dem mathematischen System der Linearperspektive. Diesem sind die meisten Bildbestandteile unterworfen. Die bildliche Struktur des Systemraums ist durch den Raum gekennzeichnet, welcher auf Grundlage des nach mathematischen Regeln funktionierenden Systems der Projektion abgebildet wird.
Vererbung	Vererbung beschreibt die Möglichkeit, ein neues Objekt von einem vorhandenen Objekt abzuleiten, wobei das neue Objekt alle Merkmale und Fähigkeiten des Alten besitzt. Dem neuen Objekt können dann weitere charakteristische Merkmale hinzugefügt werden. (vgl. [Middendorf 02]).
Zentralprojektion	Zentralprojektion ist eine Form der Projektion. Alle Projektionsstrahlen treffen sich in einem Punkt. Die Bildpunkte ergeben sich als Schnittpunkt der Projektionsstrahlen mit der Bildebene. (vgl. [Pareigis 90])

B Abkürzungsverzeichnis

API	Application Programming Interface
BiLL	Bildsprache LiveLab
DLL	Dynamic Link Library
EPK	Erweiterte Perspektivische Korrektur
FLTK	Fast Light Toolkit
fps	Frames Per Second
GUI	Graphical User Interface
LGPL	Lesser General Public License
LoD	Level of Detail
OpenGL™	Open Graphics Library
OSG	OpenSceneGraph™

C Literaturverzeichnis

[Agrawala et. al. 2000] AGRAWALA, M.; ZORIN, D.; MUNZER, T.: *Artistic Multiprojection Rendering*, In: 11th Eurographics Workshop on Rendering 2000, Brno, 2000

[Angel 90] ANGEL, E.: *Interactive Computer Graphics, A top-down approach with OpenGL™,* Addison Wesley, Boston, 1997

[Ebner 07] EBNER, T: *BiLL (Bildsprache LiveLab) Konzeption und Realisierung einer interaktiven Arbeitsumgebung für die Erforschung wahrnehmungs-realistischer Projektion,* Diplomarbeit, Technische Universität Dresden, Dresden, 2007

[FLTK] FAST LIGHT TOOLKIT: *Internetseite des Projektes* (URL: http://www.fltk.org/) Stand: 18.09.2007

[Foley et. al. 94] FOLEY, J.; VAN DAM, A.; FEINER, S.K.: *Grundlagen der Computergrafik. Einführung Konzepte Methoden*, Addison Wesley Verlag, 1994

[Franke et al. 05a] FRANKE, I.S.; ULRICH, A.; ZITZMANN, M.: *An Approach Overcoming the Distance between Cyber and Culture*, In: 3rd Global Conference Cybercultures, Exploring Critical Issues, Prague, 2005, 11. 13. August

[Franke et al. 05b] FRANKE, I. S.; SCHINDLER, J.; ZAVESKY, M.: *Multiperspektive versus Ergonomie*, 50. IWK, Technische Universität Ilmenau, Ilmenau, 2005, 19.-23. September, S. 483 ff.

[Franke et al. 06] FRANKE, I. S.; ZAVESKY, M.; RIEGER, R.: *The Power of Frustum Die Macht der geometrischen Mitte*, In: Neue Medien und Technologien der Informationsgesellschaft (NMI 2006), Berlin, 2006, 19.-21. Juli

[Franke et al. 07] FRANKE, I.S.; ZAVESKY, M; DACHSELT, R.: *Learning from Painting: Perspective-dependent Geometry Deformation for Perceptual Realism,* Eurographics Symposium, 2007,S.117-120

[Gamma et al. 04] GAMMA, E.; HELM, R.;JOHNSON, R.; VLISSIDES, J. *Entwurfsmuster: Elemente wieder verwendbarer objektorientierter Software,* Addison-Wesley, 2004,

[Griffel 98] GRIFFEL, F.: *Componentware. Konzepte und Techniken eines Softwareparadigmas*, Dpunkt Verlag, 1998

[Griffel 99] GRIFFEL, F.: *Komponenten – Softwarebausteine des nächsten Jahrtausends?*, in: OBJEKTspektrum 1/99 (siehe auch: http://www.sigs-datacom.de)

[Groh 05] GROH, R.: *Das Interaktionsbild – Zu den bildnerischen und theoretischen Grundlagen der Interfacegestaltung*, TUDpress Verlag der Wissenschaften, Dresden, 2005

[Groh et al. 06] GROH, R.; FRANKE, I. S.; ZAVESKY, M.: *With a Painter's Eye – An aproach to an Intelligent Camera*, In: Proceedings of Th e Virtual 2006, Isle of Rosenön, Stockholm, 2006, 14.-16. September

[Heinmann und Councill 01] HEINENMANN, G.; COUNCILL, W.: *Component Based Software Engineering: Putting the Pieces Togehter*, Addison-Wesley, 2001

[Hollmann 06] HOLLMANN, A.: *Semantische Beleuchtung mittels Farbkorrektur*, Komplexpraktikum, Technische Universität Dresden, 2006

[Java Beans] JAVABEANS: *Internetseite des Entwicklers:* (URL: http://java.sun.com/products/javabeans/) Stand: 18.09.2007

[Kim 06] KIM S.: *Narrative Strukturierung von Bildern durch intelligente Panelsetzung*, Komplexpraktikum, Technische Universität Dresden, 2006

[König 05] KÖNIG, N.: *Gestalterisch geordnete Computergrafik – Transformationsprinzip in OpenGL*, Diplomarbeit, Technische Universität Dresden , Dresden, 2005

[LGPL] GNU LESSER GENERAL PUBLIC LICENSE: *Internetseite über Lizenzbestimmungen* (URL: http://www.gnu.org/licenses/lgpl.html) Stand: 18.09.2007

[Marquardt und Völter 02] MARQUARDT, K.; VÖLTER, M.: *Plug-ins: Applikationsspezifische Komponenten.* JavaSpektrum, 2002

[Mezger 01] MEZGER, J.: *Effiziente Kollisionsdetektion in der Simulation von Textilien, Diplomarbeit,* Universität Tübingen, Tübingen, 2001

[Middendorf et al. 02] MIDDENDORF, S.; SINGER, R.; HEID, J.: *Programmierhandbuch und Referenz für die JavaTM-2-Plattform,* Standard Edition, 3. Auflage 2002

[Panofsky 85] PANOFSKY, E.: *Aufsätze zu Grundlage der Kunstwissenschaften*, Berlin, 1985

[Orlamünder und Mascolus 04] ORLAMÜNDER, D.; MASCOLUS, W.: *Computergrafik und OpenGL*, Technische Universität Dresden, Fachbuchverlag Leipzig 2004

[Open Producer] OPEN PRODUCER: *Internetseite des Projektes (* URL: http://andesengineering.com/Producer) Stand: 18.09.2007

[OSG] OPENSCENEGRAPH™: *Internetseite des Projektes* (URL: http://www.openscenegraph.org) Stand: 18.09.2007

[Panofsky 85] PANOFSKY, E.: *Aufsätze zu Grundlagen der Kunstwissenschaften*, Berliner Wissenschaftsverlag, Berlin, 1985

[Pareigis 90] Pareigis, B.: *Analytische und projektive Geometrie für die Computer-Graphik,* Teubner Stuttgart, 1990.

[Parnas 72] PARNAS, D.: *On the Criteria to Be Used in Decomposing Systems Into Modules,* Communications of the ACM, 1972

[Ruess 03] RUESS: *Simulation und Bildanalyse mit Java,* Seminar, 2003

[Schumacher 03] SCHUMACHER, J.: *Eine Plug-in-Architektur für Renew – Konzepte Methoden, Umsetzung,* Diplomarbeit, Universität Hamburg, 2003

[Seewald 04] SEEWALD, S.: *Grafik-APIs zur Softwareentwicklung für die computergestützte Sehschulung*, Diplomarbeit, Technische Universität Dresden, Dresden, 2004

[Schneider et al. 01] SCHNEIDER, U. ET AL.: *Taschenbuch der Informatik*, 4. aktualisierte Auflage, Fachbuchverlag Leipzig, Leipzig, 2001

[Shreiner 04] SHREINER, D.: *OpenGL Reference Manual. The Official Reference Document to OpenGL*, Version 1.2, Addison-Wesley, 2004

[Shreiner 05] SHREINER, D.: *OpenGL Programming Guide. The Official Guide to Learning OpenGL,* Version 2, Addison-Wesley, 2005

[Szyperski 02] SZYPERSKI, C.: *Component Software. Beyond Object-Oriented Programming*, Addison-Wesley Verlag, 2002,

[Tzscheutschler und Clemente 06] TZSCHEUTSCHLER, R.; CLEMENTE, M.: *Farbgestützte Navigation,* Komplexpraktikum, Technische Universität Dresden, 2006

[Wernecke 94] WERNECKE, J.: *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor,* Addison-Wesley, 1994

[Wolf 06] WOLF, J.: *C++ von A bis Z Das umfassende Handbuch (Galileo Computing*, Galileo Press, 2006

[Wojdziak 06] WOJDZIAK, J.: *Präsentationen in 3D*, Belegarbeit, Technische Universität Dresden, Dresden, 2006

[Zavesky 06] ZAVESKY, M.: *Die erweiterte perspektivische Korrektur -Ein geometrisches Verfahren zur dialogorientierten computergrafischen Abbildung dreidimensionaler Szenen*, Belegarbeit, Technische Universität Dresden, Dresden, 2006

[Zavesky 07] ZAVESKY, M.: *Dynamische Multiperspektive - Untersuchung der Eigenschaften von perspektivisch korrigierten Abbildungen in dynamischen Kontexten*, Diplomarbeit, Technische Universität Dresden, Dresden, 2007

D Abbildungsverzeichnis

Abbildung 1.1: Projekt „Looking Glass“ von Sun® Microsystems 2
Abbildung 2.1: dreidimensionales Koordinatensystem 8
Abbildung 2.2: schematische Darstellung eines Sichtkörpers (nach [Angel 1990]) 9
Abbildung 2.3: Die geometrische Mitte (nach [Franke et al. 06]) 9
Abbildung 2.4: Szenengraph zur Visualisierung eines Fahrrads [Seewald 04]............. 10
Abbildung 2.5 Subgraphen innerhalb eines Szenengraphs [Seewald 04] 11
Abbildung 2.6: Szenengraph im Kontext einer Anwendung [Seewald 04] 12
Abbildung 2.7: Rendering Pipeline [Franke et al. 05a] 12
Abbildung 2.8: Funktionsprinzip der Open Producer Kamera (nach [Open Producer]). 14
Abbildung 3.1: Skizze einer allgemeinen Komponentenarchitektur 18
Abbildung 3.2: Plug–in Architektur eines Browsers (nach [Ruess 03]) 20
Abbildung 3.3: Reduktion der Weitwinkelverzerrungen bei gekrümmten Flächen 22
Abbildung 3.4: Ausgangssituation [Franke et al. 07] 24
Abbildung 3.5: Positionsabhängige Scherung von Objekten [Franke et al. 07]........... 24
Abbildung 3.6: Rotation zu Beibehaltung der Objektansicht [Franke et al. 07]........... 24
Abbildung 4.1: BiLL – Das Viewerfenster.. 30
Abbildung 4.2: BiLL – Das Editorfenster 30
Abbildung 4.3: Architektur von BiLL 32
Abbildung 4.4: Wege–Markierung [Groh 05].. 33
Abbildung 4.5: Narrative Panelsetzung [Kim 06] 33
Abbildung 4.6: Kamerafahrt auf rundem Pfad 34
Abbildung 4.7: Multipanorama runder Kamerapfad [Franke et al. 05b]...................... 34
Abbildung 4.8: Farbperspektive der Tiefe [Tzscheutschler und Clemente 06]............. 34
Abbildung 4.9: Farbperspektive im Post-process [Hollmann 06]................................. 34
Abbildung 4.10: schematischer Aufbau der BiLL Schnittstelle.................................... 42
Abbildung 4.11: bidirektionale Bindung.. 42
Abbildung 4.12: zwei unidirektionale Bindungen.. 42
Abbildung 4.13: schematische Darstellung der Plug-in Architektur............................ 44
Abbildung 4.14: Architektur des Plug–in Managers 46
Abbildung 4.15: Sequenzdiagramm des Ladevorgangs eines Plug–ins....................... 47
Abbildung 4.16: Ausführung einer Plug–in Funktion in der Arbeitsumgebung 48
Abbildung 4.17: Vorgang zum Entladen eines Plug–ins... 48
Abbildung 4.18: Entwicklung und Integration eines Plug–ins in BiLL 49
Abbildung 4.19: Szenengraph ohne Transformationsknoten...................................... 51
Abbildung 4.20: Szenengraph mit Transformationsknoten... 51
Abbildung 5.1: Plug–in Manager in BiLL 59
Abbildung 5.2: Skizze der Struktur des Plug-in Konzepts 61
Abbildung 5.3: Installationsroutine der Arbeitsumgebung... 64
Abbildung 5.4: Benutzeroberfläche des Plug–ins in BiLL.. 65
Abbildung 5.5: Szenendarstellung mit 36 Grad Kameraöffnungswinkel..................... 67
Abbildung 5.6: Szenendarstellung mit 120 Grad Kameraöffnungswinkel................... 67
Abbildung 5.7: schematischer Aufbau des Verfahrens der EPK [Zavesky 07]............... 68
Abbildung 5.8: Skizze Versuchsaufbau 72

Abbildung 5.9: 3D–Darstellung Versuchsaufbau 72
Abbildung 5.10: Monoperspektivische Darstellung der Versuchsszene 72
Abbildung 5.11: Bildserien der Versuchsreihe 73
Abbildung 6.1: Zusätzliche Rotation im Verfahren zur Perspektivkorrektur 79
Abbildung 6.2: Multiprojektion mit mehreren Kameras (Vogelperspektive) 81
Abbildung 6.3: Abbildung mit korrekter Tiefensortierung 81

E Tabellenverzeichnis

Tabelle 1: Technologiespezifikation der BiLL Arbeitsumgebung 31
Tabelle 2: Übersicht der Analyse von Komponenten– und Plug–in Architekturen........ 39
Tabelle 3: Referenz der Schnittstellenklasse Plug_Object 56
Tabelle 4: Referenz der Schnittstellenklasse BiLLInterface 58

F Zusatzelemente

Zusatz 1: OSG Referenz 14
Zusatz 2: Open Producer Referenz 14
Zusatz 3: FLTK Referenz 15
Zusatz 4: BiLL starten 31
Zusatz 5: BiLL starten 54
Zusatz 6: Interface Referenz 58
Zusatz 7: BiLL Setup 64
Zusatz 8: Quellcode der EPK 68
Zusatz 9: Quellcode EPKUserData 71
Zusatz 10: Videos zu den Versuchen 73