BOOGA
Ein Komponentenframework für Grafikanwendungen

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Christoph Streit
von Jaberg, BE

Leiter der Arbeit: Prof. Dr. H. Bieri
Institut für Informatik
und angewandte Mathematik,
Universität Bern

BOOGA
Ein Komponentenframework für Grafikanwendungen

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Christoph Streit

von Jaberg, BE

Leiter der Arbeit: Prof. Dr. H. Bieri
Institut für Informatik
und angewandte Mathematik,
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, den 22. Mai 1997 Der Dekan:

Prof. Dr. H. Bunke

Zusammenfassung

Die vorliegende Arbeit befasst sich mit dem Entwurf und der Realisierung eines flexiblen Grafiksystems für die Unterstützung der verschiedensten Teilgebiete der Computergrafik. Beschrieben wird das Grafiksystem BOOGA, welches sich durch grosse *Flexibilität*, gute *Erweiterbarkeit* der Grundfunktionalität und leichte *Erlernbarkeit* auszeichnet. Diese generellen Eigenschaften liessen sich erst durch das neu eingeführte Konzept des *Komponentenframeworks* und den konsequenten Einsatz der Objekttechnologie erreichen. Die Arbeit zeigt die zugrundliegenden Prinzipien von BOOGA detailliert auf, beschreibt wichtige Eigenschaften des Systems und präsentiert illustrative Beispielanwendungen.

Dank

Diese Dissertation ist am Institut für Informatik und angewandte Mathematik der Universität Bern unter der Leitung von *Prof. Dr. Hanspeter Bieri* entstanden. Ich möchte mich bei ihm an dieser Stelle herzlich für seine wertvolle Unterstützung bedanken.

Die in diesem Text dokumentierten Ergebnisse sind in enger Zusammenarbeit mit *Stephan Amann* erarbeitet worden. Seine und meine Arbeit sind unzertrennbar miteinander verknüpft. Als wichtige Ergänzung sei deshalb auf die Dissertation von Stephan Amann [Ama97] hingewiesen, deren zusätzliche Lektüre ein wesentlich vollständigeres Bild ermöglicht.

Sehr viele Personen haben an BOOGA mitgearbeitet und wesentliche Teile dazu beigetragen. Einen ganz herzlichen Dank gebührt *Bernhard Bühlmann*, dem ersten Anwender. Auch *Andrey Collison*, *Thierry Matthey*, *Thomas Teuscher*, *Pascal Habegger*, *Richard Bächler*, *Daniel Möri*, *Beat Liechti*, *Peter Sagara*, *Thomas Wenger*, *Thomas von Siebenthal*, *Lorenz Ammon* und *Roland Balmer* haben mit ihren Erweiterungen BOOGA entscheidend weitergebracht. *Andrey Collison*, *Dr. Eric Dubuis*, *Bruno Grossniklaus* und *Jean-Guy Schneider* waren mir beim Korrekturlesen dieses Textes sehr behilflich.

Besonderen Dank gebührt meiner Lebenspartnerin, *Christine Reinhard*. Ihre Unterstützung hat sehr viel zum erfolgreichen Abschluss der Arbeit beigetragen.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In der Gruppe für Computergeometrie und Grafik am Institut für Informatik und angewandte Mathematik der Universität Bern stieg ab 1991 der Bedarf für ein Grundsystem, das die vielfältigen Forschungsarbeiten im Bereich der Computergrafik und Bildverarbeitung unterstützen kann. Bereits erstellte Software konnte in vielen Fällen nicht als Grundlage für Weiterentwicklungen dienen, da unter anderem keine gemeinsamen Klassenbibliotheken verwendet wurden oder der Aufwand für eine Codeintegration viel zu hoch gewesen wäre. Die meisten Realisierungen kamen deshalb Neuentwicklungen gleich.

Gesucht wurde also ein Grundsystem, das die Wiederverwendung von Code unterstützt aber auch einen hinreichend grossen Satz von vorgefertigten Teilen anbietet, der im Laufe der Zeit immer weiter ausgebaut werden kann. Die zu diesem Zweck angebotenen Grafiksysteme unterstützten diese Forderungen teilweise, bedingten aber durch ihre Komplexität einen beträchtlichen Einarbeitungsaufwand. Flexible Systeme, die ein hinreichend grosses Anwendungsgebiet abzudecken vermögen, waren zudem kaum vorhanden. Meist konzentrierten sie sich auch lediglich auf ein Teilgebiet der Computergrafik, wie zum Beispiel die fotorealistische Darstellung von geometrischen Objekten. Auch bezüglich Erweiterungen der Grundfunktionalität wiesen viele der damals erhältlichen Grafiksysteme Einschränkungen auf.

In einer universitären Umgebung muss ein Grafiksystem Besonderes leisten, da man möglichst umfassend in den verschiedenen Forschungsbereichen unterstützt werden möchte. Zudem sollte es auch möglich sein, eigene Erweiterungen, zum Beispiel neue Algorithmen oder Verfahren, nahtlos integrieren zu können. Die Anforderungen an ein solches System sind also sehr hoch. Aus praktischer Sicht, d.h. aus der Sicht des Anwenders des Grafiksystems, sind die in der folgenden Liste geforderten Eigenschaften besonders wichtig:

- *Unterstützung verschiedener Teilgebiete der Computergrafik*

  Die Aktivitäten der Gruppe für Computergeometrie und Grafik decken die unterschiedlichsten Teilgebiete der Computergrafik ab. Im wesentlichen sind dies die synthetische Bilderzeugung, Modellierung im 2D- und 3D-Raum, sowie Bildbearbeitungs- und Bildanalyseverfahren. Das gesuchte Grafiksystem muss diese Teilgebiete hinreichend durch flexible Konzepte unterstützen.

- *Wiederverwendbarkeit und Erweiterbarkeit*

  Das Grafiksystem soll als Basis für neue Anwendungen dienen und gleichzeitig neu erstellte Elemente nahtlos integrieren können. Die gesamte Entwicklung steht also im Zeichen der Wiederverwendung, die auf mehreren Ebenen berücksichtigt werden muss. Gefordert ist der wiederholte Einsatz bereits erstellter Applikationen oder Applikationsfragmenten, die Integration neuer Abstraktionen in bestehende Designstrukturen unter Ausnützung der vorgegebenen Mechanismen und die Wiederverwendung von Analyse- und Design-Resultaten.

- *Verwendung der Objekttechnologie*

  In vielen Bereichen der Informatik nimmt die Entwicklung neuer Systeme immer grössere Ausmasse an. Die Erstellung neuer Applikationen sind ohne die Wiederverwendung bereits erstellter Software kaum mehr denkbar. Die Objekttechnologie bietet die Möglichkeit, existierenden Code zu erweitern und an neue Gegebenheiten anzupassen. Daher wird in der Softwareentwicklung heute beinahe ausschliesslich der objektorientierte Ansatz eingesetzt. Als besonders flexibel und mächtig haben sich Frameworks herausgestellt. Ein Framework im Kontext des objektorientierten Software-Engineerings ist eine Menge von Klassen, die einen abstrakten

Entwurf für die Lösung einer Familie verwandter Probleme darstellen [JF88].

Die Verwendung der Objektorientierung und der Einsatz von Frameworks bieten viele Vorteile. Aus diesem Grunde soll das gesuchte Grafiksystem auf diesen Technologien aufbauen.

- *Minimierung des Einarbeitungsaufwandes*

  Der Einarbeitungsaufwand ist ein entscheidender Faktor für die Akzeptanz eines Grafiksystems und sollte deshalb möglichst klein sein. In einer universitären Umgebung mit sehr kurzen Projektlaufzeiten ist es nicht vertretbar, dass ein zu grosser Teil der zur Verfügung stehenden Zeit für die Einarbeitungsphase verwendet wird. Deshalb sollten angepasste Dokumentationskonzepte oder für den Anwendungsbereich optimierte Software-Architekturen zur Verfügung stehen.

1.2 Das Projekt Booga

Die Suche nach einem Grafiksystem, das die obigen Anforderungen zu erfüllen vermag, verlief ergebnislos. Viele Systeme konzentrierten sich lediglich auf einen kleinen Teilbereich der Computergrafik. Sie waren zudem selten für Erweiterungen ausgelegt oder unterstützten nur die eingeschränkte Anpassung bestehender Mechanismen und Abstraktionen. Aus diesem Grund wurde 1994 der Entschluss zugunsten eines eigenen Systems gefällt, das unsere Anforderungen umfassend erfüllen sollte. Als Resultat ist das Grafiksystem Booga[1] entstanden. Das Grundsystem wurde in den letzen drei Jahren von Stephan Amann und dem Autor der vorliegenden Dissertation entworfen und implementiert. Mehrere Projekte konnten bereits unter Verwendung von Booga durchgeführt werden. Das Grafiksystem hat sich aber auch in der Lehre bewährt.

1.3 Inhalt der Arbeit

Die vorliegende Arbeit zeigt die zugrundliegenden Prinzipien von Booga detailliert auf, präsentiert wichtige Eigenschaften des Sy-

[1] Berne's Object-Oriented Graphics Architecture

stems und illustrative Beispielanwendungen. BOOGA erfüllt alle gestellten Anforderungen an ein Grafiksystem und hat sich bereits in vielen Anwendungen bewährt. Der Aufbau der Arbeit gliedert sich grob in zwei Teile:

Grundlagen

Die Forderungen an leichte Wiederverwendbarkeit und Erweiterbarkeit des Grundsystems lassen sich besonders gut mit dem objektorientierte Paradigma erfüllen. Kapitel 2 stellt die wichtigsten Elemente der objektorientierten Softwareentwicklung vor und gibt konkrete Hinweise zum Aufbau von Frameworks.

Das Kapitel 3 führt kurz in den Problembereich, d.h. die Computergrafik, ein. Deren verschiedene Teilgebiete werden charakterisiert und mögliche Klassifizierungsschemas näher vorgestellt.

Im Kapitel 4 werden Kriterien erarbeitet, die es uns erlauben Grafiksysteme bezüglich deren Aufbau, Funktionsumfang und Design und Implementierung zu vergleichen. Diese Kriterien werden auf einige konkrete Grafiksysteme angewendet. Die daraus abgeleiteten Aussagen führen dann zu neuen Lösungsansätzen für die Entwicklung des eigenen Grafiksystems.

BOOGA

Die Kapitel 5 bis 9 stellen BOOGA vor, ein objektorientiertes Grafiksystem. BOOGA definiert ein umfassendes Modell für verschiedene Gebiete der Computergrafik und ist deshalb ausgezeichnet als Forschungsplattform geeignet. Die gewählte Schichtenarchitektur garantiert einen einfachen und schrittweisen Zugang zur Funktionalität, erlaubt aber gleichwohl eine hohe Flexibilität bei der Erweiterung und Anpassung des Systems. Schliesslich zeigen einige Beispielsanwendungen die in BOOGA eingesetzten Konzepte auf und illustrieren gleichzeitig den zugehörigen Entwicklungsprozess für neue Applikationen.

In Kapitel 10 werden schliesslich einige persönliche Schlussbemerkungen des Autors über die erreichten Resultate, mögliche Erweiterungen und gemachte Erfahrungen gezogen.

Vom Leser wird erwartet, dass er grundlegende Kenntnisse der Computergrafik aber auch der Objekttechnologie besitzt. Die

Designstrukturen von BOOGA werden mit Hilfe der grafischen Notation von Booch dokumentiert [Boo94]. Als Implementierungssprache wurde C++ eingesetzt [Str95]. Alle Algorithmen sind deshalb direkt in dieser Sprache formuliert. Um sie nachvollziehen zu können, werden mindestens passive Kenntnisse von C++ vorausgesetzt. Einige Konzepte von BOOGA sind allerdings ziemlich komplex und setzen sehr gute Kenntnisse dieser Sprache aber auch der Objekttechnologie voraus.

Die in diesem Text dokumentierten Ergebnisse sind in enger Zusammenarbeit mit *Stephan Amann* erarbeitet worden. Seine und meine Dissertation bilden zusammen unsere Gesamtpräsentation von BOOGA, mit einer Aufteilung nach wesentlichen Aspekten. Als wichtige Ergänzung sei deshalb auf die Arbeit von Stephan Amann [Ama97] hingewiesen. Er präsentiert unter anderem einen sehr detaillierten Überblick der verschiedenen Konzepte der Softwareentwicklung. Im Zentrum der Diskussion steht die Wiederverwendung der Resultate aus der Analyse-, Design- und Implementierungsphase. Die dokumentierten Erkenntnisse hatten einen wesentlichen Einfluss auf die Entwicklung von BOOGA. Einen weiteren Schwerpunkt besitzt seine Arbeit bei der Vorstellung eines Vorgehensmodells für die Applikationsentwicklung mit Hilfe von BOOGA, welches motiviert und an einigen Beispielen illustriert wird. Weitere Teile sind der Präsentation von Dokumentationskonzepten gewidmet. Anhand dieser Ausführungen wird ein Dokumentationsansatz für BOOGA abgeleitet, der die verschiedenen Vorschläge zu einem umfassenden System verbindet.

Objektorientierte Softwareentwicklung

2

Das eigentliche Ziel dieser Arbeit ist der Entwurf eines flexiblen Frameworks für Grafikanwendungen. In diesem Kapitel werden nun die Grundlagen des objektorientierten Software-Engineerings aufgearbeitet. Im Vordergrund stehen Begriffsbildung und die Diskussion über eine für unseren Anwendungsbereich geeignete Software-Architektur. Diese sollte gleichzeitig flexibel und erweiterbar sein, die Wiederverwendung unterstützen und zudem einen einfachen Einstieg ermöglichen. Das objektorientierte Paradigma verspricht, genau diese Anforderungen zu erfüllen. Dieses Kapitel dient aber keinesfalls als eine Einführung in die Objekttechnologie. Hierfür sei zum Beispiel auf Booch [Boo94] oder Meyer [Mey88] verwiesen.

2.1 Einleitung

Die Wurzeln der Objektorientierung liegen beinahe 30 Jahre zurück, bei SIMULA 67 [ND81], einer Sprache speziell für die Programmierung von diskreten Simulationen entworfen. Das Konzept der *Objekte*, die eine direkte Modellierung von Elementen aus der realen Welt erlauben, ist wohl der eigentliche Grund für den Erfolg der Objektorientierung. Aber auch die Wiederverwendung von Artefakten[1],

[1] Unter einem Artefakt wird im Software-Engineering ein *Stück Software* verstanden. Dies kann sowohl ein Codefragment, ein Entwurfsmuster, eine Implementation oder auch eine Dokumentation sein [Kru92, Met95].

unterstützt durch Polymorphismus und Vererbung, haben sicherlich dazu beigetragen.

Anlässlich der NATO Software Engineering Conference von 1969 schlug McIlroy [McI69] vor, Bibliotheken von wiederverwendbaren Komponenten zu schaffen. Dieser Vorschlag wurde aufgenommen und wird heute durch diverse Konzepte konkretisiert. Dabei kann eine Wiederverwendung auf sehr unterschiedlichen Stufen erreicht werden, wie die nächsten Abschnitte aufzeigen werden.

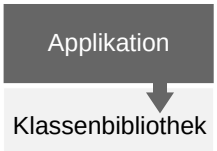
2.2 Begriffsbildungen

Präzise, allgemein anerkannte Definitionen der wichtigsten Begriffe im Kontext des objektorientierten Software-Engineerings fehlen. Die folgenden Ausführungen werden deshalb die für diese Arbeit zentralen Begriffe einführen und entsprechend erläutern. Die Definitionen sind durch die Sichtweise und Erfahrungen des Autors geprägt und unterscheiden sich deshalb teilweise von Definitionsversuchen anderer Autoren. Wo dies notwendig ist, wird im Text darauf hingewiesen.

2.2.1 Überblick

Wir konzentrieren uns auf die Begriffe *Klassenbibliothek*, *Design Pattern*, *Framework* und *Komponente*, die in Abbildung 2.1 visualisiert und in einen Zusammenhang gebracht werden. Während Design Patterns ausschliesslich abstrakte Designstrukturen beschreiben, beinhalten die anderen Elemente auch Programmfragmente oder ganze Applikationsteile, die direkt wiederverwendet werden können.

2.2.2 Klassenbibliotheken



Klassenbibliotheken entsprechen weitestgehend den von den prozeduralen Sprachen bekannten Subroutinenbibliotheken. Sie stellen eine Menge von lose gekoppelten Klassen dar, die ohne weitere Anpassungen direkt verwendet werden können. Weiter zeichnet sich eine Klassenbibliothek dadurch aus, dass immer die Applikation durch Methodenaufrufe Dienste der Bibliothek in Anspruch nimmt.

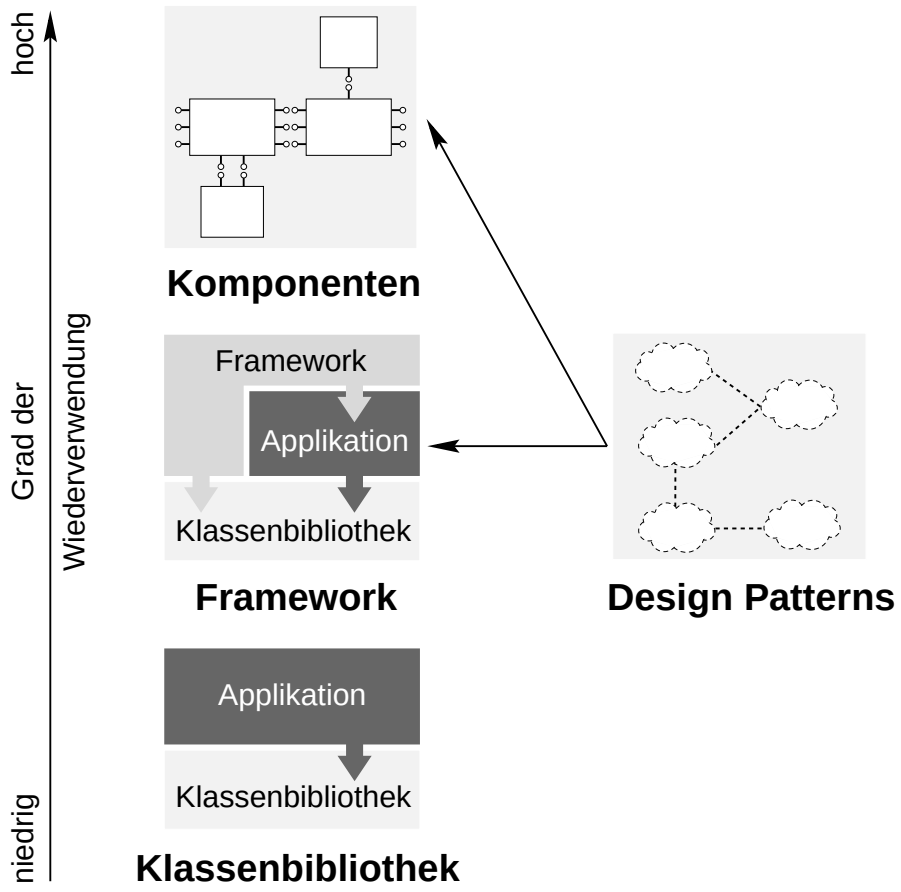


Abbildung 2.1 *Überblick der verschiedenen Konzepten des Software-Engineerings im Bereich der Objektorientierung.*

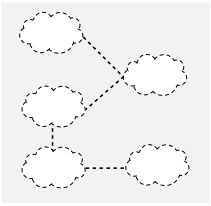
In umgekehrter Richtung findet keine Kommunikation statt, d.h. die Bibliothek ruft in der Regel keine Methoden der Applikation auf.

Typische Beispiele sind Sammlungen von Datenbehältern, wie sie unter anderem in der *Standard Template Library*, einem zukünftigen Bestandteil der C++ Standard Library, zu finden sind. Aber auch Bibliotheken für Kommunikationsaufgaben oder die grafische Präsentation von Zahlenmaterial sind zum Teil als Klassenbibliotheken aufgebaut.

Der Einfluss der Klassenbibliotheken auf die Architektur eines Systems ist gering und beschränkt sich auf die Wiederverwendung von Programmcode. Der Grad der Wiederverwendung ist klein, nämlich auf einzelne Objekte begrenzt. Für den Anwender hat dies den positiven Effekt, dass die Einarbeitungszeit im allgemeinen sehr kurz ausfällt, da kein Wissen über komplexe Wechselwirkungen zwischen den verschiedenen Klassen der Bibliothek erarbeitet werden muss.

Dies ist wohl der Hauptgrund, dass Klassenbibliotheken sehr häufig wiederverwendet werden.

2.2.3 Design Patterns



Ein Design Pattern wird von Gamma et al. [GHJV95] als *"Solution to a problem in a context"* charakterisiert und entspricht dem destillierten Wissen von erfahrenen Softwarearchitekten. Die angesprochene Lösung für ein Problem beschreibt ein Design, bestehend aus den beteiligten Klassen, sowie deren Verantwortlichkeiten, Abhängigkeiten und Zusammenspiel. Die Lösung entspricht aber nie einem konkreten Design oder einer fertigen Implementation für ein spezifisches Problem. Vielmehr liefert ein Design Pattern ein Lösungsmuster, das für die verschiedensten Problemstellungen adaptiert werden kann. Zum Beispiel gibt das Design Pattern *Composite* [GHJV95, Seite 163ff] konkrete Vorschläge für die flexible Modellierung rekursiver Datenstrukturen.

Im Unterschied zu anderen Artefakten dieses Kapitels, liegt einem Design Pattern kein Programmcode zugrunde. Es manifestiert sich als wesentliches Element einer Architektur. Booch [Boo96b] geht sogar noch einen Schritt weiter indem er sagt: *"The best patterns disappear once they manifest themselves in a system"*.

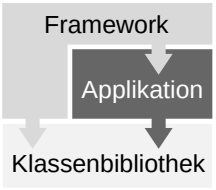
Design Patterns werden häufig auch als *Micro-Architekturen* bezeichnet, da sie keine Strukturen für ganze Applikationen vorschlagen, sondern lediglich Lösungen für Teilprobleme anbieten. Durch den Einsatz der Patterns wird *Design wiederverwendet*, was aus modernen Entwicklungsprojekten nicht mehr wegzudenken ist. Allerdings bestehen auch Gefahren. So führt der extensive Einsatz von Design Patterns nicht automatisch zu einer sehr guten Architektur, sondern zu schwer verständlichen, komplizierten Strukturen, die kaum mehr zu überblicken sind. Deshalb sollte der Ratschlag von Gamma und Weinand nicht ungehört verklingen:

> *"Don't overpattern! Design as flexible as needed, not as flexible as possible." [GW96]*

Verschiedene Sammlungen von Patterns wurden bereits veröffentlicht. Die bekannteste stammt von Gamma et al. [GHJV95], mit über 20 Design Patterns aus den verschiedensten Anwendungsbereichen. Das Werk von Buschmann et al. [BMR+96] ist eine weitere

Quelle für Design Patterns, wie auch die jährliche *PLoP* (Pattern Languages of Programs) Konferenz, die seit 1994 stattfindet.

2.2.4 Frameworks



Der Begriff des Frameworks wurde von Johnson und Foote [JF88] als *"A set of classes that embodies an abstract design for solutions to a family of related problems"* umschrieben. Im Gegensatz zur Klassenbibliothek, ist ein Framework also nicht nur eine lose Klassensammlung, sondern eine Menge von miteinander verwebten Klassen.

Die Motivation für die Entwicklung von Frameworks fusst in der Beobachtung, dass einzelne Anwendungsgebiete und Anwendungsfamilien inhaltliche und strukturelle Ähnlichkeiten aufweisen. Diese werden mit Hilfe eines Frameworks abstrahiert. Die Anwendungsentwicklung besteht nun darin, die abstrakten Klassen eines Frameworks durch anwendungsspezifische, konkrete Klassen zu ersetzen. Frameworks erleichtern es daher, die Entwicklungsaktivitäten auf die problemspezifischen Fragestellungen zu fokussieren.

Im Unterschied zu einer traditionellen Applikation ist ein grosser Teil des Kontrollflusses im Framework selbst enthalten, d.h. Applikationscode wird vom Framework aufgerufen (*inversion of control*, siehe auch Abbildung 2.1). Dadurch wird die Architektur der Applikation teilweise vorgegeben, was die Wiederverwendung von Design *und* Implementation zur Folge hat.

Johnson und Foote unterschieden in ihrer Arbeit zwei Typen von Frameworks, die *White-Box-* und *Black-Box-Frameworks*. In einem White-Box-Framework wird mit Hilfe des Vererbungsmechanismus die Adaption für eine konkrete Problemstellung vorgenommen. Dies hat zur Folge, dass die Architektur und Implementation sehr genau bekannt sein müssen, bevor überhaupt Anpassungen und Erweiterungen vorgenommen werden können. Typische Frameworks dieser Kategorie, wie zum Beispiel ET++ [WGM88], bringen einen enormen Einarbeitungsaufwand für den Anwender mit sich, erlauben aber andererseits auch sehr flexible Eingriffe in deren Struktur. Bei Black-Box-Frameworks werden durch Komposition aus vorgegebenen Komponenten die gewünschte Anwendung erstellt. Der Benutzer hat die Wahl aus einer Menge von vorgefertigten Klassen, die er an dafür vorgesehenen Orten einsetzen kann. Der Einarbeitungs-

aufwand für diesen Frameworktyp ist wesentlich geringer, allerdings ist die erreichbare Flexibilität auch entsprechend eingeschränkter.

Ein häufig verwendeter Ansatz ist die Kombination der White- und Black-Box-Sichtweise auf ein Framework. Solange man mit vorgefertigten Klassen auskommt, verwendet man den Black-Box-Ansatz. Werden neue Element benötigt, die nicht durch Komposition realisiert werden können, wird zur White-Box-Sicht gewechselt. Dieses Vorgehen kann durch unterschiedliche Schnittstellen für die beiden Sichtweisen (*Framework* und *Client API*) zusätzlich unterstützt werden (siehe Abbildung 2.2). Das Framework API erlaubt die Spezialisierung des Frameworks, indem der Benutzer eigene Klassen und Objekte entwickelt und integriert. Auf diese Weise kann er das Verhalten des Frameworks an die eigenen Bedürfnisse anpassen. Das Client API hingegen ermöglicht die Verwendung der Funktionalität des Frameworks als Ganzes.

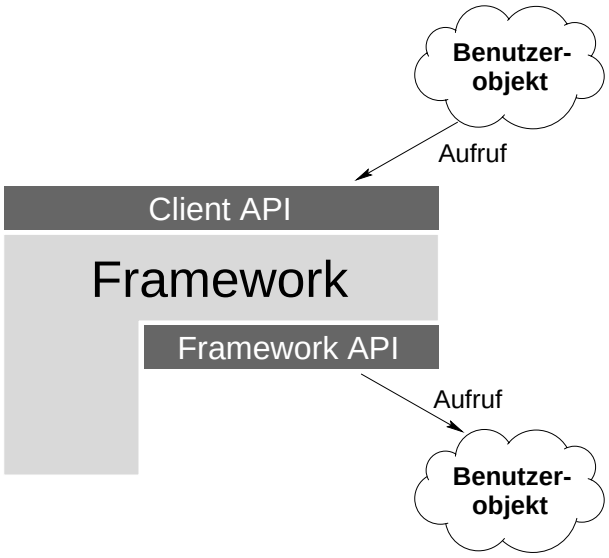


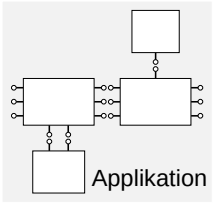
Abbildung 2.2
Schnittstellen eines Frameworks [Tal94].

Ein weiteres wichtiges Merkmal von Frameworks sind die *Hot Spot*s [Pre95]. Diese bezeichnen diejenigen Stellen im Framework, die für Erweiterungen vorgesehen sind. Sie repräsentieren somit die variablen Aspekte des Entwurfs.

Der Einfluss von Frameworks auf die Gesamtarchitektur eines Systems ist gross, stellen sie doch gewissermassen generische Applikationen dar, die lediglich durch applikationsspezifische Elemente konkretisiert werden müssen. Ist ein eingesetztes Framework für den betrachteten Anwendungsfall ausgelegt, d.h. es hat denselben *Design*

Center [GW96], so ist das Wiederverwendungspotential enorm. Der Entwurf eines Framework ist allerdings bis heute ein sehr schwieriges Unterfangen. Booch spricht sogar von 80% missglückter Frameworkprojekte [Boo96a]. Es gibt aber auch einige Beispiele, die das Potential von Frameworks überzeugend demonstrieren. Für einen Erfolg ist es allerdings unabdingbar, dass der Anwendungsbereich sehr gut verstanden wird und Erfahrung im Design von Frameworks besteht.

2.2.5 Komponenten



Die magischen Worte des Software-Engineerings lauten zur Zeit *komponentenorientierte Softwarekonstruktion*. Damit wird die Idee von McIlroy [McI69] wieder aufgegriffen, Applikationen aus vorgefertigten Elementen zusammenzubauen.

Die Motivation für diese Art der Applikationsentwicklung liegt in der Tatsache begründet, dass der Einsatz eines Frameworks mit einer nicht zu unterschätzenden Komplexität verbunden ist. Die durch den Frameworkansatz erreichte Flexibilität wird aber in vielen Fällen gar nicht benötigt oder ist sogar nicht erwünscht. Betrachtet man die Struktur einer frameworkbasierten Applikation (Abbildung 2.3, links), so fällt die starke Vernetzung und die geringe Granularität der einzelnen Abstraktionen (symbolisiert durch Kreise) auf. Gruppiert man die einzelnen Objekte zu grösseren Gebilden (grau schattierte Ovale) und versteckt die Details vor dem Anwender, so nehmen die Abhängigkeiten ab und die Struktur der Applikation ist wesentlich einfacher zu verstehen (Abbildung 2.3, rechts). Durch dieses Vorgehen erhalten wir neue Artefakte, die auf einer wesentlich höheren Abstraktionsebene angesiedelt sind. Man kann sie als Black-Boxes betrachten, die eine in sich abgeschlossene Teilaufgabe lösen und deren Implementation für den Benutzer unsichtbar bleibt. Erfüllen diese Black-Boxes gewisse Bedingungen (siehe weiter unten), so nennen wir sie *Komponenten*.

Eine einheitliche und anerkannte Definition des Komponentenbegriffs existiert zur Zeit leider noch nicht. Eine möglicher Ansatz leitet sich aus dem Vorgehen bei der Applikationsentwicklung ab. Applikation werden in diesem Kontext durch Komposition und Konfiguration vorgefertigter, in einer Bibliothek abgelegter Komponenten unter Verwendung eines *Kompositionsmechanismus* ge-

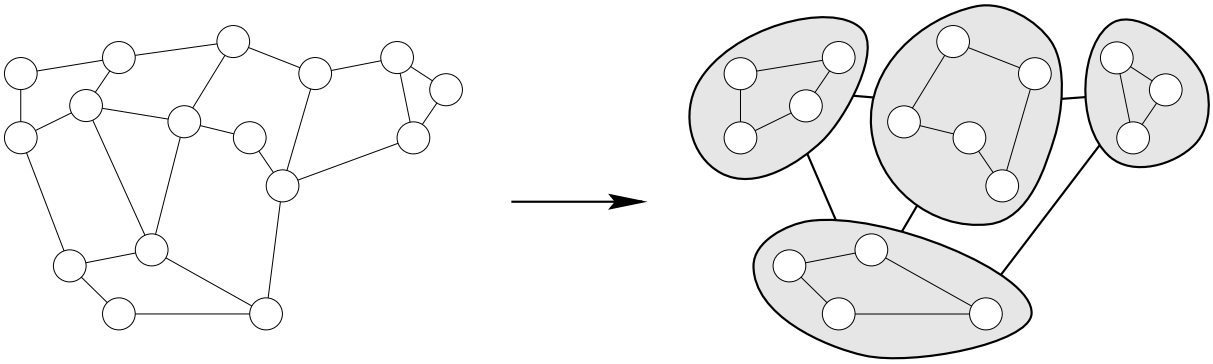


Abbildung 2.3
Vereinfachung einer frameworkbasierten Applikationsstruktur durch das Gruppieren von Objekten, wodurch Implementationsdetails versteckt werden [Wei95].

bildet. Im Idealfall spielt es keine Rolle, in welcher Programmiersprache oder für welches Betriebssystem die einzelnen Komponenten verfasst wurden. Bedingung hierfür ist ein Kompositionsmechanismus, der Programmiersprachen- und Betriebsystem-unabhängig arbeitet.[2] Damit Applikationen auf die beschriebene Art und Weise gebaut werden können, müssen Komponenten einigen Anforderungen gerecht werden:

Komponenten sind Black-Boxes.

Eine Black-Box gibt keine Implementationsdetails preis und ist nur über ihr Interface ansprechbar. Die Wiederverwendung geschieht durch Komposition, die Anpassung an die jeweiligen Gegebenheiten einer Applikation durch Konfiguration.

Komponenten haben ein Dienst- und Konfigurationsinterface.

Das Dienstinterface (service interface) erlaubt den Zugang zur eigentlichen Funktionalität der Komponenten, während das Konfigurationsinterfaces eine Anpassung des Verhaltens an die jeweiligen Bedürfnisse erlaubt.

Komponenten lösen abgeschlossene Teilaufgaben.

Damit die Abhängigkeiten zu anderen Komponenten

[2] CORBA (*Common Object Request Broker Architecture* [OMG95]) liefert eine Umgebung, die genau dies zu leisten vermag.

möglichst gering ausfallen, sollten Komponenten abgeschlossene Teilaufgaben behandeln. Dadurch kann eine Komponente für sich alleine stehen und repräsentiert im Extremfall eine eigenständige Applikation. Es ist aber darauf zu achten, dass die Interfaces einfach und überschaubar gehalten werden sollten, um einen Einsatz so einfach wie möglich zu gestalten.

Komponenten werden für die Wiederverwendung entworfen und implementiert.

Eine gute Komponente sollte in verschiedenen Applikationen wiederverwendet werden können. Um dieses Ziel zu erreichen, muss bereits bei ihrer Realisierung auf die Wiederverwendung geachtet werden. So sollte eine Komponente zum Beispiel keine applikationsspezifischen Teile enthalten.

Komponenten können miteinander kombiniert werden.

Die Mächtigkeit des Komponentenansatzes wird durch die Kombination mehrerer Komponenten erreicht. Dadurch können komplexe Aufgaben durch das Zusammenfügen von einfachen, vorgefertigten Elementen gelöst werden. Damit die Zusammenarbeit in der Praxis funktioniert, müssen die Komponenten allerdings auf einem gemeinsamen Kompositionsmechanismus aufbauen. Im einfachsten Fall sind alle Komponenten in derselben Programmiersprache implementiert, die dann gleichzeitig als Kompositionsmechanismus dienen kann.

Durch den Einsatz von Komponenten resultiert eine Architektur, die keinen monolitischen Charakter mehr aufweist. Die Applikation ist eine lose gekoppelte Sammlung von Komponenten, die sich bei Verwendung eines geeigneten Kompositionsmechanismus in unterschiedlichen Adressbereichen oder sogar auf entfernten Computersystemen aufhalten können. Die jeweilige Architektur wird daher massgeblich durch den gewählte Kompositionsmechanismus beeinflusst.

Die durch den Komponentenansatz erreichbare Flexibilität ist im Vergleich zu den in den vorangegangenen Abschnitten diskutierten Konzepten eingeschränkt. Erfüllt eine Komponente nicht exakt die an sie gestellten Anforderungen und lässt sie sich auch nicht entsprechend konfigurieren, so kann sie nicht eingesetzt werden.

Ein Ansatz, der die einfache Handhabung von Komponenten mit der Flexibilität und Erweiterbarkeit von Frameworks kombiniert,

wäre wünschenswert. Im Abschnitt 2.4 wird aus diesem Grunde ein kombiniertes Modell vorgestellt.

2.3 Vergleiche

Die in den vorangegangenen Abschnitten vorgestellten Konzepte des objektorientierten Software-Engineerings haben sehr unterschiedliche Eigenschaften. Ein Vergleich kann deshalb auf verschieden Kriterien beruhen. Zum einen interessiert sicherlich der Grad der Wiederverwendbarkeit, d.h. die Granularität der Wiederverwendung[3]. Abbildung 2.4 zeigt genau dieses Potential in grafischer Form auf. Offensichtlich bietet der Einsatz von Komponenten das grösste Potential, während Klassenbibliotheken nur in geringem Ausmass die Entwicklung beeinflussen.

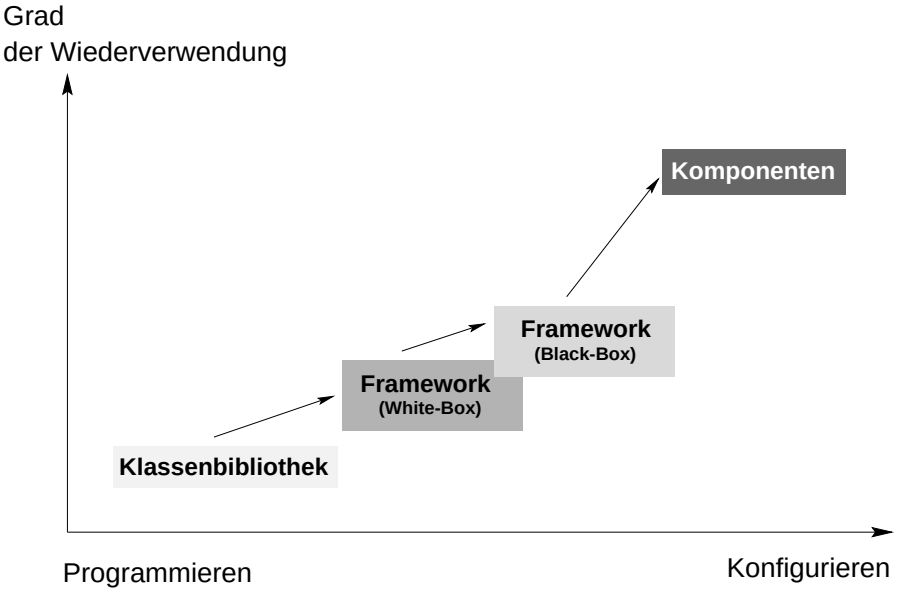


Abbildung 2.4 *Der Grad der Wiederverwendung ist bei den verschiedenen Konzepten des objektorientierten Software-Engineerings sehr unterschiedlich. [Gam96]*

Eine weitere Untersuchung kann bezüglich dem Einsatzzeitpunkt eines Konzeptes im gesamten Entwicklungsprozess vorgenommen werden (siehe Abbildung 2.5). Je früher der Einsatz, desto grösser die resultierende Einsparung beim Entwicklungsaufwand. Komponenten werden bereits von der Analysephase an berücksichtigt, bieten sie doch schlüsselfertige Lösungen für ganze Teilprobleme

[3] Die grösste Granularität der Wiederverwendung wird durch den Einsatz bereits erstellter Applikationen erreicht, während das Wiederverwenden von Codefragmenten einer kleinen Granularität entspricht.

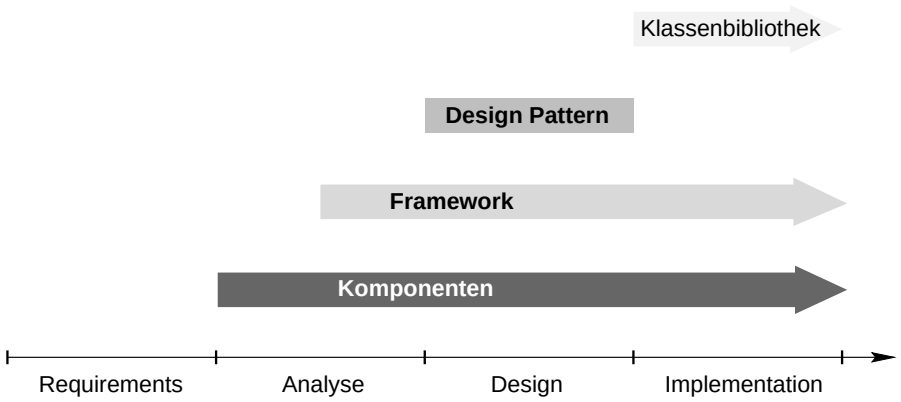


Abbildung 2.5 *Vergleich der Software-Engineering Konzepte bezüglich des Einsatzzeitpunktes im Entwicklungsprozess.*

und beeinflussen zudem die zugrundeliegende Architektur massgeblich. Design Patterns spielen im Unterschied dazu ausschliesslich während der Design Phase eine gewichtige Rolle.

Abschliessend sind in der Tabelle 2.1 die drei wichtigsten Merkmale der verschiedenen Konzepte zusammengetragen. Die *Granularität* quantifiziert den Grad der Wiederverwendung, die *Flexibilität* beschreibt die Adaptierbarkeit auf eine Problemstellung und die *Komplexität* steht für den Lern- und Einarbeitungsaufwand eines Anwenders. Ein $-$ entspricht einer negativen Ausprägung, o bedeutet neutral und $+$ ist eine positive Wertung.

Konzept	Granularität	Flexibilität	Komplexität
Klassenbibliothek	$-$	o	$+$
Design Pattern	o	$++$	$-$
Framework	$+$	$++$	$--$
Komponente	$++$	$--$	$++$

Tabelle 2.1 *Zusammenstellung verschiedener Merkmale der betrachteten Software-Engineering Konzepte.*

2.4 Ein kombiniertes Modell

Jedes der vorgestellten Konzepte hat seine Vor- und Nachteile. Durch eine Kombination der verschiedenen Ansätze lassen sich die Vorteile nutzen, ohne mit den jeweiligen Nachteilen leben zu müssen. Unser Vorschlag für ein kombiniertes Modell sieht eine Architektur aus drei Schichten vor, bestehend aus einer Klassenbibliothek, einem Framework und einem Komponentenaufsatz (siehe

Abbildung 2.6). Das Resultat ist eine komponentenorientierte Frameworkstruktur oder kurz ein *Komponentenframework*.

Abbildung 2.6
Ein Komponentenframework ist die Kombination von Klassenbibliothek und Framework mit einem Komponentenaufsatz.

Bei der Applikationsentwicklung mit einem Komponentenframework bedient sich der Anwender mit Vorteil der vorgefertigten Komponenten der obersten Ebene. Dies garantiert den grössten Nutzen und ist zudem die einfachste Art mit dem System umzugehen. Reicht die angebotene Funktionalität nicht aus, wird zur Frameworkschicht gewechselt. Hier können neue Komponenten erstellt werden, unterstützt durch die vorgegebenen Mechanismen des Frameworks. Weitergehende Anpassungen, die nicht durch das Konfigurationsinterface der Komponenten möglich sind, werden ebenfalls hier behandelt. Die Frameworkschicht baut ihrerseits auf einer Klassenbibliothek auf. Auch auf dieser Stufe sind Erweiterungen und Anpassungen denkbar. Natürlich kann die Frameworkschicht auch auf mehreren Klassenbibliotheken, ausgelegt für unterschiedliche Problembereiche, aufbauen.

Komponentenframeworks bieten einen guten Kompromiss zwischen Flexibilität und Komplexität. Die volle Flexibilität für Erweiterungen und Anpassungen steht durch den Zugang zu den verschiedenen Schichten jederzeit zur Verfügung. Wird diese nicht benötigt, können auf einfache Art und Weise Applikationen durch Komposition und Konfiguration von Komponenten erstellt werden. Dem Einsteiger erlaubt dieses Konzept eine schrittweise Annäherung an die Funktionalität, wodurch ein rasches und effizientes Arbeiten mit dem System ermöglicht wird.

Hiermit sind die für diese Arbeit relevanten Begriffe des objektorientierten Software-Engineerings eingeführt. In den folgenden Kapiteln werden sie unter anderem für den Vergleich von Grafiksystem herangezogen. Der Ansatz des Komponentenframeworks bildet zudem die Basis für die Architektur des Systems, das in den Kapiteln 5 bis 9 näher beschrieben wird.

3 Grafiksysteme

Das folgende Kapitel gibt eine kurze Einführung in den Bereich der Grafiksysteme. Diskutiert werden die verschiedenen Teilgebiete der Computergrafik und einige Anforderungen an ein konkretes Grafiksystem aus Anwendersicht.

In dieser Arbeit wird unter dem Begriff *Computergrafik* all das verstanden, was im weitesten Sinne mit einer bildlichen Darstellung mit Computerhilfe in Zusammenhang gebracht werden kann. Sei dies beispielsweise die Erstellung von Bildern aus verschiedenen, nicht notwendigerweise in bildlicher Form vorliegenden Daten, oder die Manipulation einer geometrischen Beschreibung. *Grafiksysteme* sind Bibliotheken oder Frameworks (siehe das vorangegangene Kapitel), die Teilgebiete der Computergrafik unterstützen.

3.1 Teilgebiete der Computergrafik

Die ISO[1] definiert Computergrafik als die *Summe aller Methoden und Techniken, Daten für die grafische Ausgabe vorzubereiten.* Unter diese Definition fallen allerdings immer noch vielfältige Teilgebiete und es fällt schwer eine geeignete Klassifizierung zu finden, die die einzelnen Disziplinen etwas detaillierter abzudecken vermag.

[1] International Standards Organization

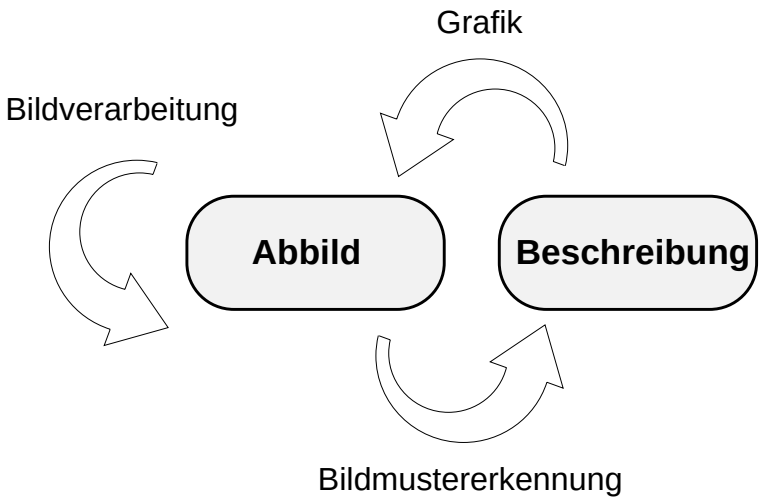


Abbildung 3.1
Teilgebiete der Computergrafik und deren Zusammenhänge nach Pavlidis [Pav82].

Verschiedene Ansätze zur Klassifizierung wurden deshalb von diversen Autoren vorgeschlagen. So unterscheidet etwa Pavlidis [Pav82][2] die Teilgebiete *Grafik* (Erstellung von Bildern aus Daten in nicht bildlicher Form), *Bildverarbeitung* (Problemstellungen mit Eingaben und Ausgaben in bildlicher Form) und *Bildmustererkennung* (Methoden zur Beschreibung von Bildvorlagen). Auf der Seite der verwendeten Datenstrukturen führt diese Unterteilung zu den Elementen *Abbild* und *Beschreibung*, wie dies auch in Abbildung 3.1 dargestellt wird.

Eine vollständigeres Modell stammt von Meier [Mei86] und wird in Abbildung 3.2 schematisch dargestellt. Dieser Vorschlag diente als Ausgangspunkt für den Entwurf und die Realisierung eines eigenen Grafiksystems, das mit einem einheitlichen Modell die verschiedenen Teilgebiete integrieren sollte.

Als Unterteilungskriterium diente Meier die Unterscheidung in *Bildraum* und *Objektraum*. Er bezeichnet den Objektraum als den physikalischen Raum. Der Bildraum hingegen, hilft räumliche Objekte durch grafische Darstellungen zu veranschaulichen. Er entspricht einer grossen, möglicherweise mehrdimensionalen Matrix.

Die *geometrische Datenverarbeitung* befasst sich mit der Darstellung, Speicherung und Verarbeitung geometrischer Informationen.

[2] Die Klassifizierung von Pavlidis geht auf einen Vorschlag von Rosenfeld [NR72] zurück.

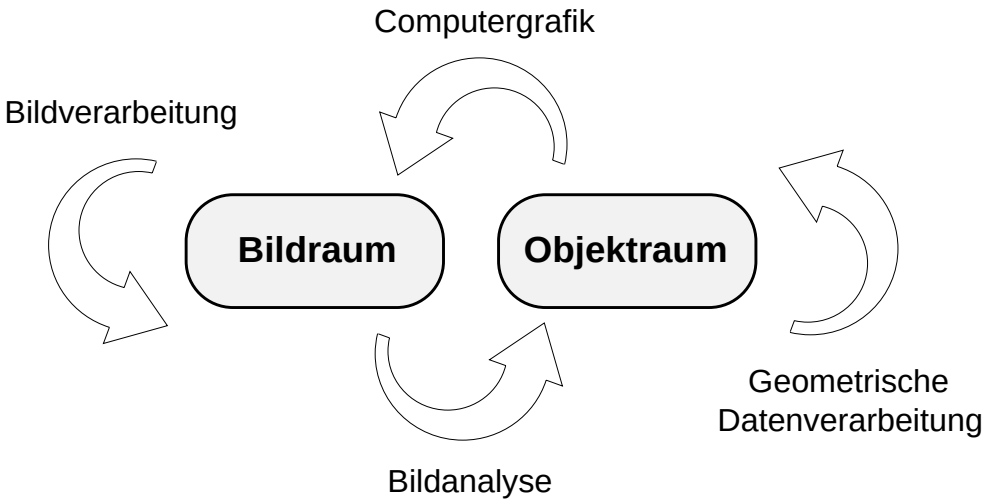


Abbildung 3.2
Überblick der in dieser Arbeit betrachteten Teilgebiete des Grafikbereichs [Mei86].

Im Vordergrund stehen mathematische Verfahren zur Beschreibung der Gestalt von Objekten, sowie die Berechnung von geometrischen und topologischen Eigenschaften.

In der *Computergrafik* steht die Umwandlung von Daten des Objektraumes in grafische Daten des Bildraumes im Zentrum. Aus einem Objektmodell wird eine bildhafte Darstellung erzeugt, die auf einem grafischen Ausgabegerät sichtbar gemacht werden kann.

Die *Bildverarbeitung* beschäftigt sich vor allem mit der Transformation digitaler Bilder. Typische Algorithmen in diesem Bereich erlauben etwa die Elimination von Bildstörungen oder die Hervorhebung von Kanten.

Schliesslich werden mit den Methoden der *Bildanalyse* Bilder mit dem Ziel verarbeitet, eine Beschreibung der abgebildeten Objekte zu gewinnen. Als Resultat wird ein Modell erzeugt, das dem im Bild dargestellten Objekt entspricht.

Unberücksichtigt bleibt bei der Klassifizierung nach Meier aber die Tatsache, dass geometrische Information des Objektraumes unterschiedliche Dimension aufweisen kann (meist auf den 2D- und 3D-Fall beschränkt). Auch kann der Objektraum typische Bildraumelemente, wie beispielsweise mit einer Textur attributierte geometrische Objekte, enthalten. Trotzdem ist dieser Ansatz für unsere Zwecke nützlich und diente sogar als Basis für die Entwicklung des

zugrundeliegenden Konzeptes von BOOGA (siehe Kapitel 5). Die Klassifizierung nach Meier ist allerdings noch keine hinreichende Grundlage für ein Grafiksystem. Der folgende Abschnitt stellt deshalb einige Anforderungen aus Anwendersicht auf, die es bei der Realisierung zu berücksichtigen gilt.

3.2 Anforderungen an ein Grafiksystem

Der Begriff des Grafiksystems wurde in den bisherigen Erläuterungen oft verwendet. Er lässt sich wie folgt umschreiben:

> *Ein Grafiksystem implementiert Lösungen, d.h. Datenstrukturen und Algorithmen, für ein oder mehrere Teilgebiete der Computergrafik.*

Somit ist ein Grafiksystem Bestandteil einer ganzen Applikation. Solche Systeme können als Bibliotheken oder Frameworks konzipiert werden, wie dies in Kapitel 4 ab der Seite 45 an einigen Beispielen aufgezeigt wird. Bevor aber ein Konzept und eine konkrete Architektur eines Grafiksystems definiert werden können, müssen erst verschiedene Anforderungen präzisiert sein. Zum Beispiel stellt die direkte Unterstützung möglichst vieler Teilgebiete der Computergrafik hohe Ansprüche an ein solches System, müssen doch die unterschiedlichsten Datenstrukturen und Algorithmen unter einem Konzept integriert werden.

Die folgende Liste stellt einige Anforderungen und Fragestellungen auf, die es für ein konkretes Grafiksystem zu berücksichtigen gilt. Die Aufzählung erfolgt aus Anwendersicht und erhebt keinen Anspruch auf Vollständigkeit.

Welche Anwendungsgebiete sollen abgedeckt werden?
: Alle Grafiksysteme haben ein klar definiertes Anwendungsgebiet. Sei das nun die Unterstützung der geometrischen Modellierung oder die Aufbereitung von Fotografien mit den Methoden der Bildverarbeitung. Sollen mehrere der in der Klassifizierung von Meier angesprochenen Bereiche unterstützt werden, muss eine entsprechende Architektur als gemeinsame Basis definiert werden.

Soll zwischen dem 2D- und 3D-Fall unterschieden werden?

Viele CAD[3]- oder GIS[4]-Applikationen arbeiten nur mit 2D-Datensätzen. Andere Applikationen, wie Virtual-Reality-Systeme, benötigen auch eine Unterstützung im 3D-Bereich.

Einige Systeme, wie beispielsweise OpenGL [Arc92], betrachten den 2D- als degenerierten 3D-Fall und setzen die z-Koordinate auf einen konstanten Wert. Allerdings kann der 2D- nicht immer als Spezialfall des 3D-Falles behandelt werden, da hierfür viele spezialisierte Algorithmen existieren, die keinerlei Entsprechungen in höheren Dimensionen kennen.

Erweiterbarkeit und Flexibilität.

Ein Grafiksystem sollte für unser Empfinden nicht starr konzipiert sein, sondern flexible Erweiterungen erlauben. Dies ermöglicht eine nachträgliche Anpassung an neue Gegebenheiten, die beim Entwurf nicht berücksichtigt wurden:

> *"Good frameworks can be used for things that the designers never dreamed of." [Joh96]*

Diese Eigenschaft ist für eine Forschungsplattform unabdingbar, da gerade hier der Einsatz für neue Anwendungsgebiete im Vordergrund steht. Allerdings wird Erweiterbarkeit und Flexibilität nicht automatisch erreicht, sondern muss durch entsprechende Konzepte (siehe Kapitel 4 ab Seite 34) aktiv unterstützt werden.

Lokale Erweiterbarkeit.

Ein grösseres Grafiksystem erlaubt Erweiterungen und Anpassungen in unterschiedlichen Bereichen. So können beispielsweise neue Algorithmen, neue Darstellungsmethoden oder neue Objekttypen hinzugefügt werden. Entscheidend ist dabei, dass Erweiterungen in einem Gebiet nicht Anpassungen anderer Teile erfordern, d.h. die lokale Erweiterbarkeit sollte unterstützt werden.

Leichte Erlernbarkeit.

Flexible Grafiksysteme mit einem grossen Funktionsumfang

[3] Computer Aided Design
[4] Geographische Informationssysteme

haben eine inhärente Komplexität. Ein Anwender ist gezwungen einen grossen Einarbeitungsaufwand zu leisten, bevor er mit dem System effizient arbeiten kann. Diesem Umstand muss deshalb bereits in der Entwurfsphase Rechnung getragen werden. Zum Beispiel kann durch ein Schichtenmodell eine schrittweise Annäherung an die Funktionalität des Grafiksystems ermöglicht werden, die am Anfang viele Details und komplexe Zusammenhänge versteckt. Eine weitere Unterstützung kann durch die Bereitstellung von Standardfällen erreicht werden, so dass ein Anwender nur in seltenen Fällen das System an die eigenen Wünschen anpassen muss. Weinand nennt dieses Vorgehen *Supporting defaults for the 95% cases* [Wei96], was sich als gutes Hilfsmittel für den Umgang mit der Komplexität eines umfangreichen Grafiksystems erwiesen hat.

Im nachfolgenden Kapitel werden nun Vergleichskriterien für Grafiksysteme erarbeitet und an einigen ausgewählten Beispielen angewendet. Die Analyse der architektonischen Eigenschaften konkreter Systeme liefert zudem die technischen Grundlagen, um die eben aufgestellten Anforderungen zu erfüllen. In unserem Fall dienten die Ergebnisse als Basis für den Entwurf von BOOGA, den wir im Kapitel 5 ab Seite 61 näher besprechen.

4 Vergleich von Grafiksystemen

Das vorliegende Kapitel stellt zwei Möglichkeiten für den Vergleich von Grafiksystemen vor. Zum einen ist dies der ISO Standard *Computer Graphics Reference Model* (CGRM), der den Versuch unternimmt, eine abstrahierte Beschreibung eines Grafiksystems zu definieren. Der Standard liefert eine eindeutige Begriffsdefinition und ermöglicht durch die Abbildung bestehender Systeme auf den CGRM Standard auch deren Vergleichbarkeit.

Als weiteres Kriterium für die Beurteilung von Grafiksystemen können das zugrundeliegende Design und die Implementationstechnik betrachtet werden. Abschnitt 4.2 befasst sich deshalb ausgiebig mit Design- und Implementierungsvarianten von Grafiksystemen im Wandel der Zeit. Dabei werden Parallelen zu den Fortschritten im Software-Engineering offensichtlich.

Abschliessend werden die erarbeiteten Kriterien dazu verwendet, einige ausgewählte Grafiksysteme zu analysieren und deren Anwendungsgebiete aufzuzeigen. Aus dieser Analyse werden einige Vorschläge für den Aufbau von Grafiksystemen abgeleitet, die bei der Realisierung von Booga (siehe Kapitel 5) berücksichtigt wurden.

4.1 CGRM

Der ISO Standard *Computer Graphics Reference Model (CGRM)* [ISO92, Pro95, Bei94] ist eine formale Beschreibung eines Grafiksystems, d.h. es dient keineswegs als Referenz für die Implementierung

eines konkreten Systems, wie das die Standards GKS [ISO85] für den 2D- oder PHIGS [ISO89] für den 3D-Bereich tun. Der CGRM Standard verfolgt die nachstehenden Ziele:

- Definition eines *Modells* für Grafiksysteme,
- Erarbeitung einer konsistente *Namensgebung*,
- Kriterien für den *Vergleich* vom Grafiksystemen bereitstellen und
- Anhaltspunkte für die Spezifikation neuer Systeme geben.

CGRM orientiert sich am Modell einer Computergrafik-Applikation wie in Abbildung 4.1 dargestellt. Applikationen dieses Typs verwenden Grafik-Subsysteme, die in CGRM in Anlehnung an die klassische Darstellungs-Pipeline in fünf Schichten, den *Environments*, unterteilt werden. Für die Benennung der einzelnen Environments, aber auch Prozess- oder Datenelementen, wurden konsistente und unverwechselbare Begriffe gewählt.

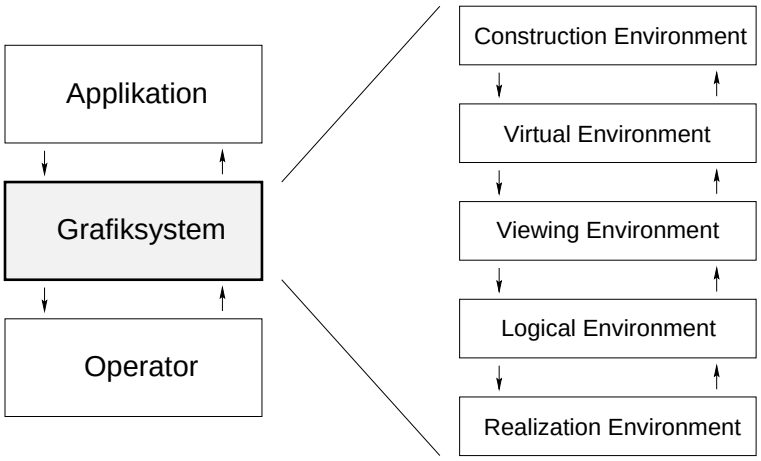


Abbildung 4.1 *Das ISO Computer Graphics Reference Model (*CGRM*) entspricht einem Modell eines Grafiksystems.*

Im folgenden werden diejenigen Teile des Standards detailliert vorgestellt, die für die Charakterisierung bestehender Systeme eine wichtige Rolle spielen werden.

4.1.1 CGRM Environments

Die fünf Environments stellen formal unabhängige Systeme mit klar definierten Ein- und Ausgabewerten und internem Zustand

dar. Grundsätzlich kommunizieren alle Environments mit der direkt über- oder untergeordneten Ebene. Alle Environments sind konzeptuell vorhanden, können aber ohne Funktion sein, d.h. Daten passieren eine solche Ebene unverändert.

Construction Environment

Das Construction Environment stellt die Schnittstelle zwischen applikationsspezifischen und grafischen Daten dar. Die Umgebung enthält ein *Modell*, das Begriffe und Masse aus dem Umfeld der Applikation verwendet. So könnte das Modell eines Autos vier Räder enthalten, deren Abmessungen in Metern angegeben sind. Zudem muss die Geometrie nicht vollständig definiert sein, wodurch andere Umgebungen beliebige Transformationen durchführen können. Die Veränderung des Modells selbst kann allerdings nur in dieser Umgebung vorgenommen werden.

Virtual Environment

Im Virtual Environment wird das Modell des Construction Environments in eine *Szene* transformiert. Im Unterschied zur übergeordneten Umgebung ist die Geometrie vollständig definiert. Lediglich nicht geometrische Attribute, wie beispielsweise die Farbe, müssen noch nicht vollständig spezifiziert sein.

Viewing Environment

Informationen zu Blickrichtung und Standort eines Betrachters wird verwendet, um die Szene aus dem Virtual Environment in ein *Abbild* überzuführen. Perspektivische Transformationen können angewendet werden, um zum Beispiel die Dimension von 3D nach 2D zu reduzieren. Alternativ kann nach der perspektivischen Transformation auch ein 3D-Abbild der Szene resultieren, dessen Tiefeninformation erst im Realization Environment zur Berechnung von verdeckten Geradensegmenten oder Flächen herangezogen wird.

Das erzeugte Abbild liegt in einer idealisierenden Form vor, d.h. die Eigenschaften des Ausgabegerätes werden in keiner Weise berücksichtigt, was die hauptsächliche Aufgabe der beiden nachfolgenden Umgebungen ist.

Grafische Eingaben aus dem Logical Environment können unter anderem von einem 2D- in ein 3D-Koordinatensystem transformiert werden.

Logical Environment

Der Übergang vom Viewing zum Logical Environment kann auch als Übergang vom geräteunabhängigen zum geräteabhängigen Teil der Verarbeitung umschrieben werden. Alle verbleibenden Attribute werden in dieser Umgebung definiert und ein grafisches *Bild* wird für die abschliessende Darstellung erzeugt. Unter anderem werden Attribute des Ausgabegerätes wie minimale Streckendicke oder die Verfügbarkeit von Farben berücksichtigt.

Grafische Eingaben stammen aus dem Realization Environment und werden in eine geräteunabhängige Form gebracht.

Realization Environment

Das Realization Environment komplettiert den grafischen Ausgabeprozess, indem das Bild der Szene an ein Ausgabemedium geschickt wird.

In dieser Umgebung werden grafische Eingaben direkt vom Operator empfangen. Sie ist somit die einzige Umgebung, in der ein Benutzer Eingaben machen kann.

4.1.2 Aufbau eines Environments

Umgebungen sind identisch aufgebaut (siehe Abbildung 4.2) und setzen sich aus Datenelementen zur Speicherung von Information und Prozesselementen zur Manipulation der Informationen zusammen. Alle Datenelemente einer Umgebung besitzen Schnittstellen, um Schreib- und Lesezugriffe auf Metafiles zu ermöglichen.

Im folgenden werden die einzelnen Prozess- und Datenelemente kurz charakterisiert:

Prozesselemente

Absorption

Der Absorption-Prozess empfängt Ausgabeeinheiten der übergeordneten Umgebung und wendet auf diese geometrische und andere Transformationen an, um sie in Einheiten seiner Umgebung zu konvertieren. Eine solche Konvertierung ist zum Beispiel die Zerlegung eines Ausgabeobjektes der höheren Umgebung in ein oder mehrere Primitive der aktuellen Umgebung.

Manipulation

Der Manipulation-Prozess stellt die Verbindung von Eingabe und Ausgabe innerhalb einer Umgebung her.

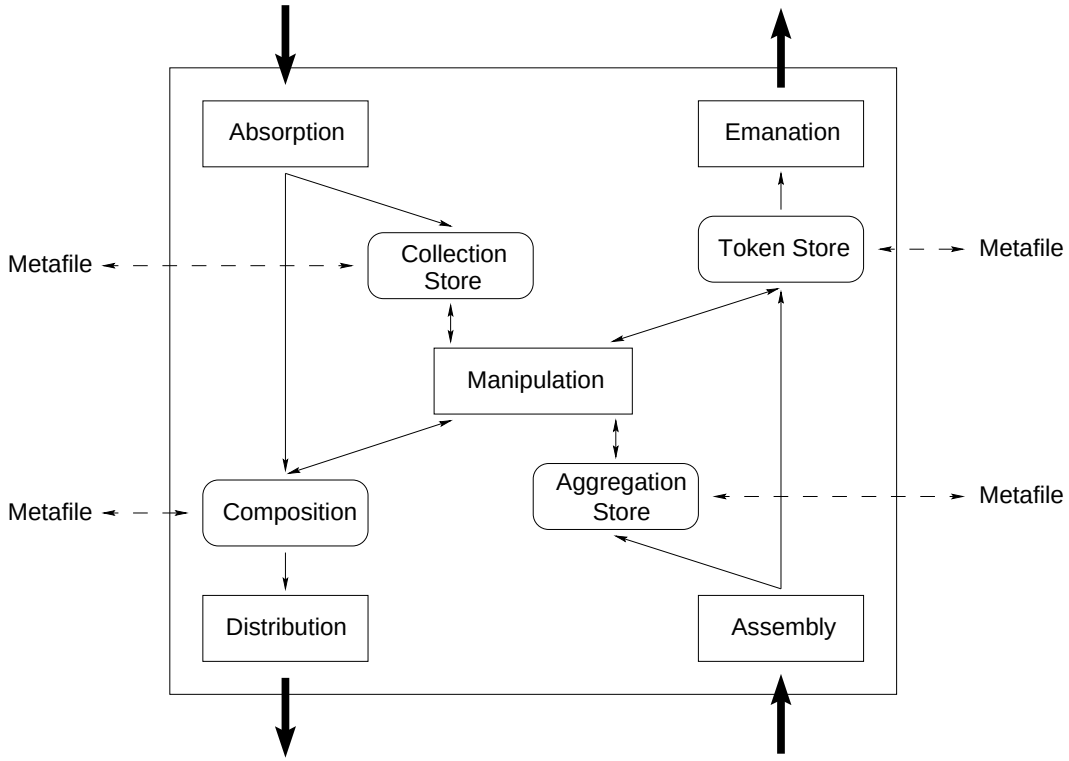


Abbildung 4.2
Der interne Aufbau eines CGRM-*Environments.*

Distribution
Der Distribution-Prozess ist zuständig für die Realisierung von Filteroperationen.

Assembly
Der Assembly-Prozess hat die Aufgabe, Eingabedaten aus mehreren Quellen zu sammeln und in Token zusammenzufassen, um sie dann im Token oder Aggregation Store einzufügen.

Emanation
Der Emanation-Prozess verarbeitet Tokens, die durch eine Transformation oder Konvertierung für die übergeordnete Umgebung verarbeitbar werden.

Composition ***Datenelemente***
Die Composition ist eine räumlich strukturierte Menge vom Ausgabeprimitiven.

Collection Store
Im Collection Store werden grafische Ausgabeelemente manipuliert.

Token Store
Der Token Store ist der Speicher für Eingabedaten.

Aggregation Store
Im Aggregation Store werden Eingabeelemente manipuliert.

Die eben erwähnten Begriffe können für alle Environments gleichermassen verwendet werden. Spricht man allerdings von einer einzelnen dieser fünf Umgebungen, so werden anstelle der Begriffe Absorption, Emanation, Composition und Token Store spezifische Bezeichnungen verwendet, die der jeweiligen Umgebung besser angepasst sind (siehe Abbildung 4.3).

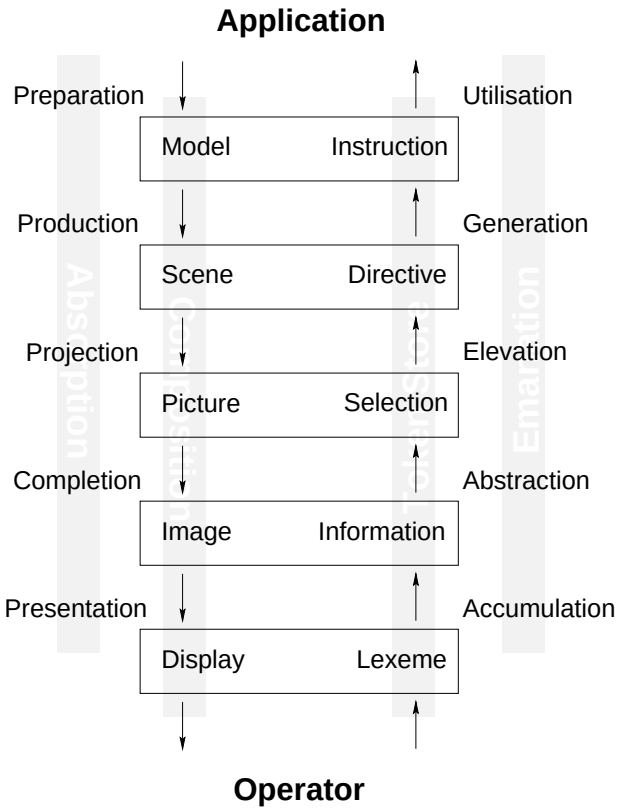


Abbildung 4.3
Die Elemente Absorption, Emanation, Composition und Token Store haben in den einzelnen Umgebungen spezifische Bezeichnung, um die jeweilige Bedeutung im Kontext besser zu umschreiben.

4.1.3 CGRM und PHIGS

Das *Programmers Hierarchical Interactive Graphics System* (PHIGS) [ISO89] ist ein von der ISO standardisiertes 3D-Grafiksystem. PHIGS erlaubt die Erzeugung und Verwaltung komplexer Modelle und trennt diese Operationen von der eigentlichen Visualisierung. Der CGRM Standard beschreibt die Abbildung der Darstellungs-Pipeline von PHIGS auf das Modell eines Grafiksystems aus CGRM. Abbildung 4.4 zeigt diese Abbildung grafisch auf. Das Beispiel dient zur Illustration der Inhalte der fünf Environments von CGRM an einem konkreten und bekannten Grafiksystem.

4.1.4 Beurteilung

Der CGRM Standard definiert ein Modell eines Grafiksystems, bestehend aus fünf Schichten mit einem jeweils identischen und vollständigen Aufbau. Mit Hilfe des Modells und den zugehörigen, klar definierten Begriffen lassen sich die meisten der existierenden Systeme einheitlich beschreiben und miteinander vergleichen (siehe auch Abschnitt 4.3). Der Aufbau als Schichtenmodell mit klarer Aufgabenverteilung und Schnittstellen zur Aussenwelt, erlaubt die Austauschbarkeit von Umgebungen und den Zugriff auf Daten, die zwischen diesen Umgebungen ausgetauscht werden. Ein einzelnes Environment kann also als Black-Box angesehen werden, das nach Bedarf durch eine alternative Implementierung ersetzt werden kann.

CGRM ist besonders gut geeignet, um herkömmliche Grafiksysteme zu beschreiben. Die mehrstufige CGRM-Pipeline entspricht in etwa den Transformationen und Verarbeitungsschritten einer Rendering-Pipeline. Unter anderem wird in [ISO92] die Abbildung von PHIGS (siehe auch den vorangegangenen Abschnitt) und GKS nach CGRM präzisiert. Andere Darstellungsmethoden, wie zum Beispiel das Raytracing, lassen sich aber nicht so leicht in dieses Schema pressen.

Der CGRM Standard bietet einen brauchbaren Ansatz zum Vergleich von Grafiksystemen. Die herkömmliche Aufteilung der Darstellungs-Pipeline in fünf Schichten wird allerdings durch neuere Grafiksysteme in Frage gestellt und zum Teil anders gelöst (siehe Abschnitt 4.3). Auch uns scheint CGRM als Vorgabe für den

Entwurf eines Grafiksystems als zu wenig flexibel. Die Auseinandersetzung mit den Ideen von CGRM hatte aber sicherlich einige positive Einflüsse, ist doch schon die Erkenntnis, dass eine traditionelle Modellierung der Darstellungs-Pipeline zu wenig flexibel ist äusserst wichtig. ISO Standards werden alle 5 Jahre überprüft und gegebenenfalls angepasst. Es ist zu hoffen, dass neue Anforderungen einfliessen werden und so der CGRM Standard in Zukunft als gemeinsame Grundlage vieler Grafiksysteme dienen kann.

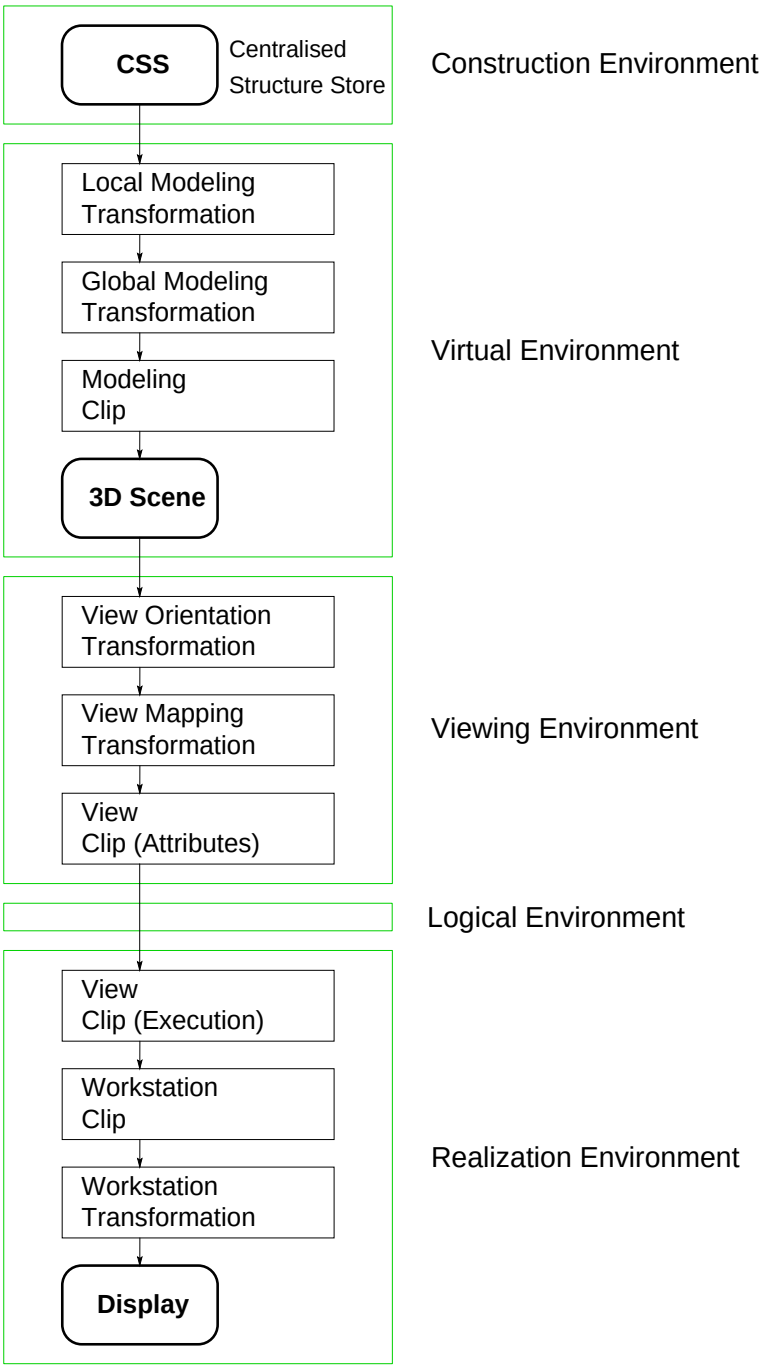


Abbildung 4.4 *Die Abbildung der* PHIGS *Darstellungs-Pipeline auf die fünf Environments des* CGRM *Standards.*

4.2 Design und Implementierung von Grafiksystemen

Die für den Entwurf und die Implementierung von Grafiksystemen verwendeten Techniken spiegeln die Fortschritte im Software-Engineering Bereich wieder. Dies liegt vor allem daran, dass sich Grafikanwendungen besonders gut eignen, Konzepte des Software-Engineerings an einer anschaulichen und allgemein verständlichen Problemstellung aufzuzeigen. Die folgenden Abschnitte orientierten sich deshalb an den verschiedenen Abstraktionsmechanismen, die das Software-Engineering hervorgebracht hat (siehe auch Kapitel 2). Beginnend mit *Grafikbibliotheken* werden verschiedene Grundmechanismen von *Grafikframeworks* diskutiert, um dann den prinzipiellen Aufbau eines *komponentenorientierter Grafiksystems* vorzustellen. Der letzte Ansatz ist zur Zeit der Erstellung dieser Arbeit noch kaum verwendet worden. Es ist aber zu erwarten, dass entsprechende Systeme bald erhältlich sein werden. Der gewählte Aufbau dieses Abschnitts repräsentiert gleichzeitig eine historische Entwicklung. Trotzdem sind auch heute noch alle vorgestellten "Spielarten" von Grafiksystemen anzutreffen oder werden sogar neu entwickelt.

Die vorgeschlagene Unterteilung erlaubt einen Vergleich von Grafiksystemen auf Basis deren Designmerkmalen und Implementierungstechniken. Dadurch lassen sich Aussagen über Wiederverwendung, Einarbeitungsaufwand oder Erweiterbarkeit machen, im Unterschied zur Beurteilung von Grafiksystemen mit Hilfe des CGRM Standards, durch den vor allem der Funktionsumfang beschrieben werden kann.

4.2.1 Grafikbibliotheken

Wie in Abschnitt 2.2.2 dargelegt, ist eine Bibliothek eine Sammlung von lose gekoppelten Funktionen oder Klassen. Im Unterschied zu einem Framework enthält eine Bibliothek keinerlei Kontrollfluss, der vollständig von der Applikation spezifiziert werden muss. Typischer Vertreter einer 3D-Grafikbibliothek ist das im Abschnitt 4.3.1 diskutierte OpenGL. Bibliotheken im 2D-Bereich sind Xlib [CGG+88, SGFR90] oder SRGP [FvDFH90].

Grafikbibliotheken zeichnen sich durch die folgenden Merkmale aus:

Keine Verwaltung von Szeneninformation
Eine Grafikbibliothek bietet keine Datenstrukturen und Funktionen zur Verwaltung von Szeneninformationen, sondern entspricht einem *Immediate Rendering* Modell.

Realisierung der Darstellung
Grafikbibliotheken realisieren die Schichten Viewing, Logical und Realization des CGRM Standards, d.h. ihr Schwerpunkt liegt bei der Darstellung von grafischen Objekten. Häufig sind die Bibliotheken für die optimale Ausnutzung von Grafikhardware ausgelegt.

Lokale Beleuchtungsmodelle
Globale Beleuchtungsmodelle benötigen den Zugriff auf die gesamte Szeneninformation, um die Berechnung des Lichtaustausches zwischen geometrischen Objekten ermitteln zu können. Grafikbibliotheken verwalten keine Szeneninformationen und implementieren deshalb ausschliesslich lokale Beleuchtungsmodelle.

Eingeschränkte Erweiterbarkeit
Erweiterungen der Funktionalität lassen sich lediglich durch Einkapselung von Funktionen und Datenstrukturen, d.h. durch eine zusätzliche Softwareschicht erreichen und sind nicht durch entsprechende Konzepte der Grafikbibliothek unterstützt. Beispiel einer solchen Erweiterung ist GLUT [Kil94], welches auf OpenGL (siehe Abschnitt 4.3.1) aufsetzt und unter anderem die Verarbeitung von NURBS Objekten ermöglicht.

Basis für Grafikframeworks
Grafikbibliotheken bilden häufig die unterste Schicht (Realisierungsschicht) von Grafikframeworks, die auf den Darstellungsmöglichkeiten der Bibliothek aufbauen und höhere Konzepte wie Szenenmodellierung realisieren. Beispiele solcher Frameworks sind OpenInventor (siehe Abschnitt 4.3.3) und GROOP [KW93].

4.2.2 Grafikframeworks

Als Grafikframework wird ein System bezeichnet, das in sich die Elemente

- Szenen-*Modellierung*,
- Szenen-*Darstellung* und
- Szenen-*Traversierung*

vereinigt und im Sinne eines Frameworks (siehe Abschnitt 2.2.4) Mechanismen für das Zusammenspiel der drei Elemente vorgibt.

In der Literatur können die verschiedenen Entwicklungsschritte nachvollzogen werden, die zu heutigen Systemen geführt haben. Die folgenden Ausführungen orientieren sich daran und verfeinern die relevanten Teilaspekte eines stark vereinfachten objektorientierten Grafiksystems. Die Diskussion konzentriert sich auf die Evolution eines objektorientierten Systems, da sich

- die aktuelle Forschung beinahe ausschliesslich mit objektorientierten Systemen beschäftigt [BW89, WK90, LBdMP95],
- das objektorientierte Paradigma dem mentalen Modell von grafischen Objekten weitgehend entspricht und
- der objektorientierte Ansatz eine effiziente Erweiterbarkeit ermöglicht.

Objekthierarchie

Heckbert [Hec89] schlug vor, dass geometrische Objekte als Basisabstraktionen mit *identischen Interfaces* modelliert werden sollten. Ein Grafiksystem lässt sich dadurch unabhängig von den konkret vorhandenen geometrischen Objekten realisieren, indem lediglich eine Abhängigkeit zum gemeinsamen Interface erlaubt wird. Diese Idee wurde zuerst für Raytracing-Systeme realisiert und lässt sich heute in allen objektorientierten Grafiksystemen wiederfinden. Das Konzept erlaubt die Erweiterung eines Grafiksystems um neue geometrische Objekte, ohne die Grundfunktionalität anzupassen, falls die neuen geometrischen Objekte das vorgegebene Interface implementieren. Diese weit verbreitete Realisierungsvariante impliziert eine Klassenhierarchie von geometrischen Objekten, zum Beispiel mit einer abstrakten Basisklasse `Object` wie in Abbildung 4.5 dargestellt. Jede von `Object` abgeleitete Klasse repräsentiert ein geometrisches Objekt und implementiert eine Menge von Methoden,

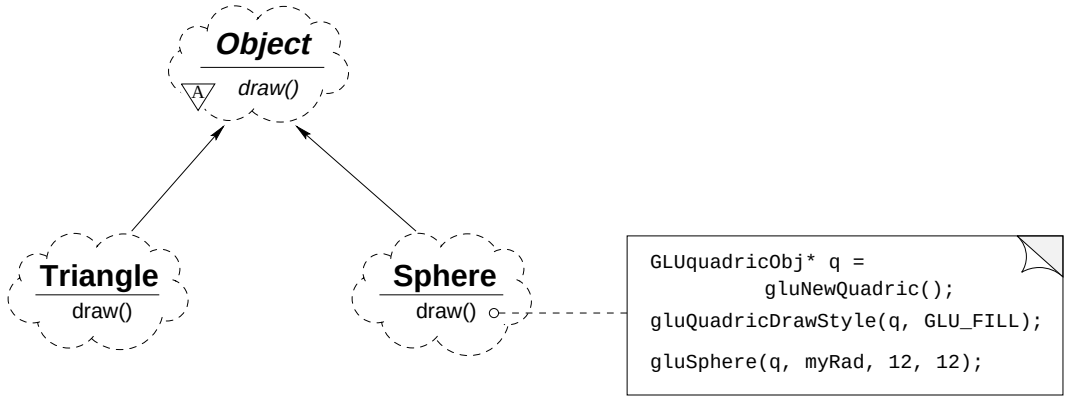


Abbildung 4.5
Hierarchie von geometrischen Objekten eines Grafiksystems mit gemeinsamem Interface.

die für das jeweilige Grafiksystem wichtig sind. In unserem vereinfachten Beispiel ist dies die Methode `draw()`, die für die Darstellung verantwortlich ist. In Abbildung 4.5 ist die Implementation dieser Methode für die Klasse `Sphere` unter Verwendung der Grafikbibliothek OpenGL dargestellt.

Aggregate

Kirk und Arvo [KA88] haben die Idee der identischen Interfaces für alle geometrischen Objekte in einem Grafiksystem verallgemeinert, indem sie Strukturierungsmechanismen als eigenständige Objekte, sogenannte *Aggregate*, auffassen und in die Klassenhierarchie integrieren. Typische Aggregate sind zum Beispiel einfache listenbasierte Behälter oder aufwendige Octree-Strukturen [Hun78, RR78]. Abbildung 4.6 zeigt die um ein Aggregat-Objekt erweiterte Klassenhierarchie. Eine Implementation basiert üblicherweise auf dem weit verbreiteten Design Pattern *Composite* [GHJV95, Seite 163ff]. Die `draw()` Methode ist häufig durch Delegation realisiert, wie bei der in Abbildung 4.6 gezeigten Variante.

Das Konzept erlaubt die beliebige Verschachtelungen von Aggregatstrukturen und ermöglicht das Zusammenfassen von Objekten zu Gruppen. Dadurch wird der strukturierten Aufbau einer Szene erreicht, wie Abbildung 4.7 an einem Beispiel[1] demonstriert.

[1] Das Beispiel stammt ursprünglich von Andrey Collison ist aber unseren Bedürfnissen angepasst worden.

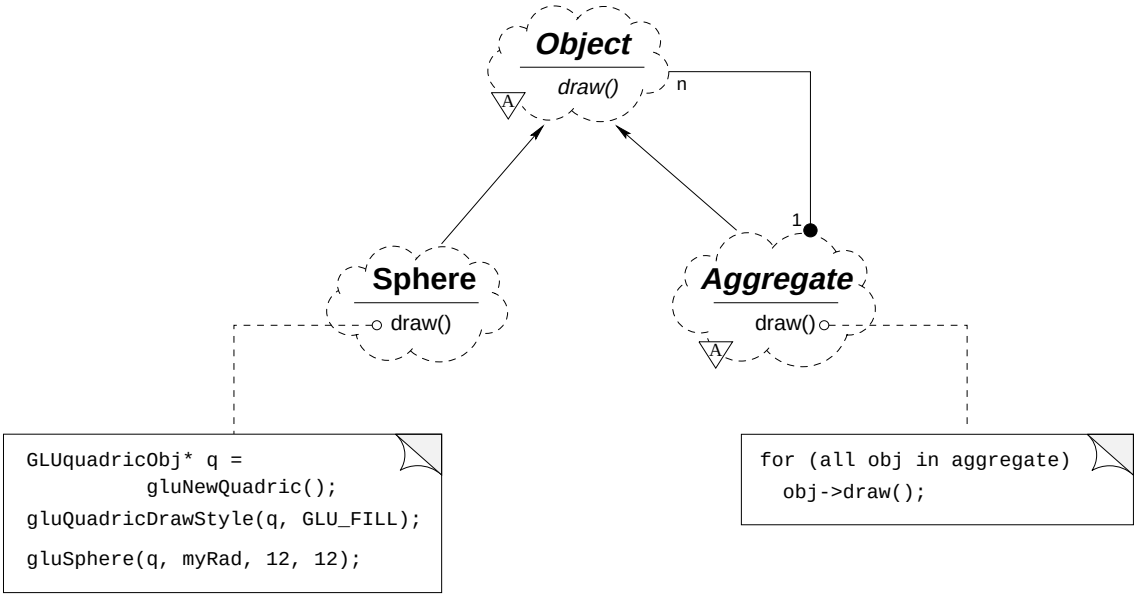


Abbildung 4.6
Objektbehälter (Aggregate) als eigenständige Grafikobjekte implementiert.

Darstellungsstrategie

Die bis hierher entwickelte Hierarchie von geometrischen Objekten erlaubt eine flexible Modellierung von Szenen. Sie ist aber äusserst unflexibel bezüglich der zugrundeliegenden Darstellungsstrategie, die in unserem Beispiel durch die Möglichkeiten von OpenGL eingeschränkt ist. Sollen alternative Darstellungstechniken verwendet werden, so müssten alle `draw()` Methoden angepasst oder zusätzliche Methoden für die jeweilige Darstellungstechniken hinzugefügt werden, zum Beispiele eine `drawWithMyTechniqe()`-Methode für alle Darstellungstechniken. Beide Varianten haben den Nachteil, dass jeweils die gesamte Objekthierarchie angepasst werden muss und sind deshalb bezüglich Erweiterbarkeit und Flexibilität ungeeignet. Als Ausweg bietet sich eine von der Hierarchie der geometrischen Objekte getrennte Darstellungshierarchie an. Abbildung 4.8 zeigt den prinzipiellen Aufbau eines solchen Systems.

Die abstrakte Klasse `GDevice` bildet die Basis aller Darstellungsstrategien und definiert Methoden für alle im System vorkommenden geometrischen Objekte. Diese Methoden werden in den davon abgeleiteten Spezialisierungen überschrieben. Die in Abbildung 4.8 dargestellte Klasse `OpenGLDevice` verdeutlicht dieses Vorgehen.

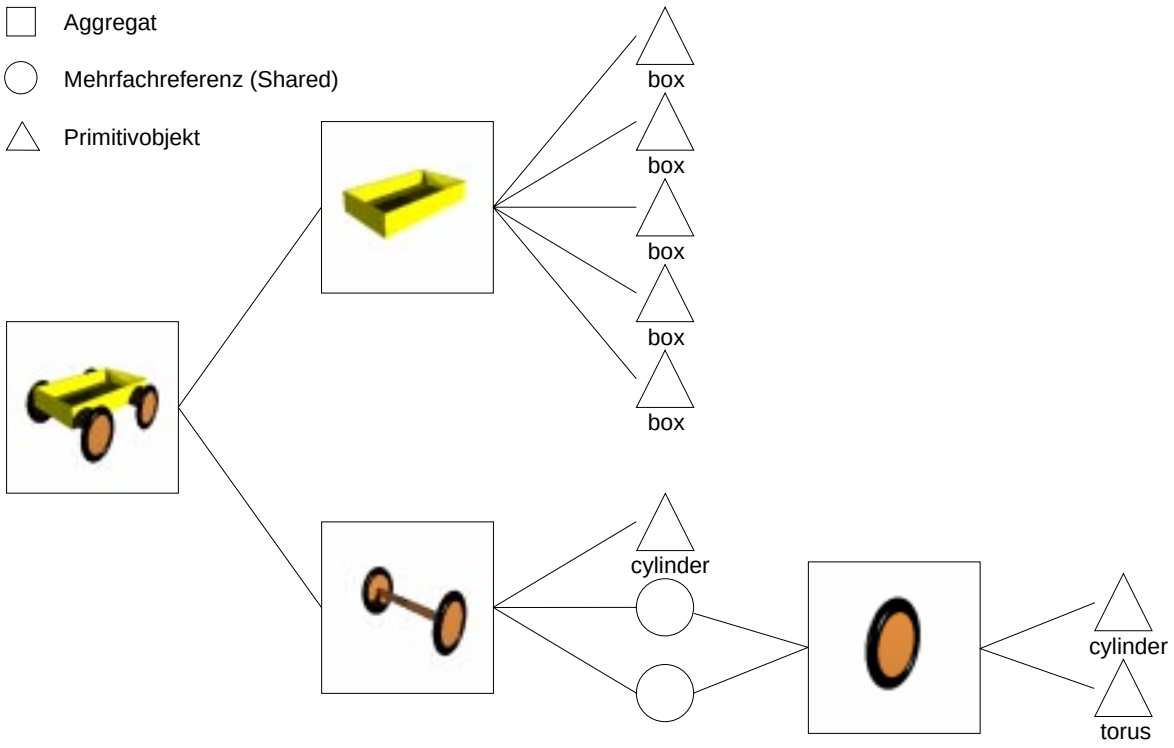


Abbildung 4.7
Aggregate können zur logischen Gliederung einer Szene dienen.

Dieselbe Technik wird unter anderem in den Systemen GRAMS [Egb92, Egb95] und HOOPS [CG94] eingesetzt.

Eine konkrete Implementation folgt dem *Visitor* Design Pattern [GHJV95, Seite 331ff], welche sich nur dann als geeignet erweist, wenn die Hierarchie der geometrischen Objekte statisch bleibt. Wird sie durch einen Objekttyp erweitert, müssten alle von `GDevice` abgeleiteten Klassen mit einer neuen `drawNewObj()`-Methode ergänzt werden. Im allgemeinen ist aber eine solche Einschränkung nicht tolerierbar, da häufig mit neuartigen Objekttypen experimentiert wird. Um eine unabhängige Erweiterung beider Vererbungshierarchien zu gewährleisten, müssen deshalb Varianten des Visitor Patterns entwickelt werden, wie dies für das BOOGA-System getan wurde (siehe Kapitel 8).

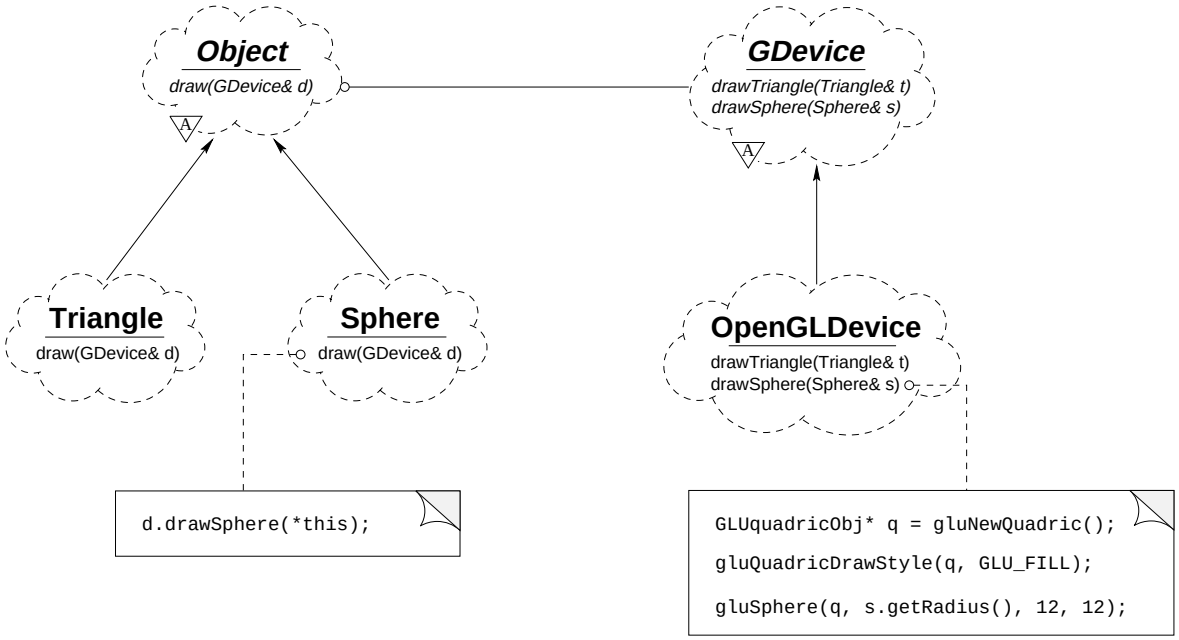


Abbildung 4.8
Explizite Modellierung der Darstellungsstrategie durch eigene Abstraktionen, wodurch die geometrischen Objekte unabhängig einer spezifischen Darstellungstechnik werden.

Traversierungsstrategie

Unterschiedliche Darstellungstechniken benötigen eine jeweils angepasste Traversierung der Objekte, die in einer gegebenen Szene enthalten sind. So werden bei einer Darstellung mit Hilfe von OpenGL alle geometrischen Objekte der Szene in Preorder-Reihenfolge besucht. Für eine interaktive Applikation müsste hingegen diese Traversierung dahingehend modifiziert werden, dass bei jeder Benutzereingabe abgebrochen wird, um möglichst rasch entsprechend reagieren zu können. Es scheint unmöglich alle Anforderungen an die Traversierungsstrategie vorauszusehen und entsprechend zu implementieren, weshalb diese als eigene Abstraktion realisiert werden sollte. Dieses Konzept führt zu einer Dreiteilung von geometrischen Objekten (Hierarchie `Object`), deren Darstellung (Hierarchie `GDevice`) und deren Traversierung (Hierarchie `Traversal`) wie in Abbildung 4.9 dargestellt.

Abbildung 4.10 zeigt den Ablauf bei der Darstellung einer einfachen Szene. Die Implementierung entspricht dem *Iterator* Design Pattern [GHJV95, Seite 257ff].

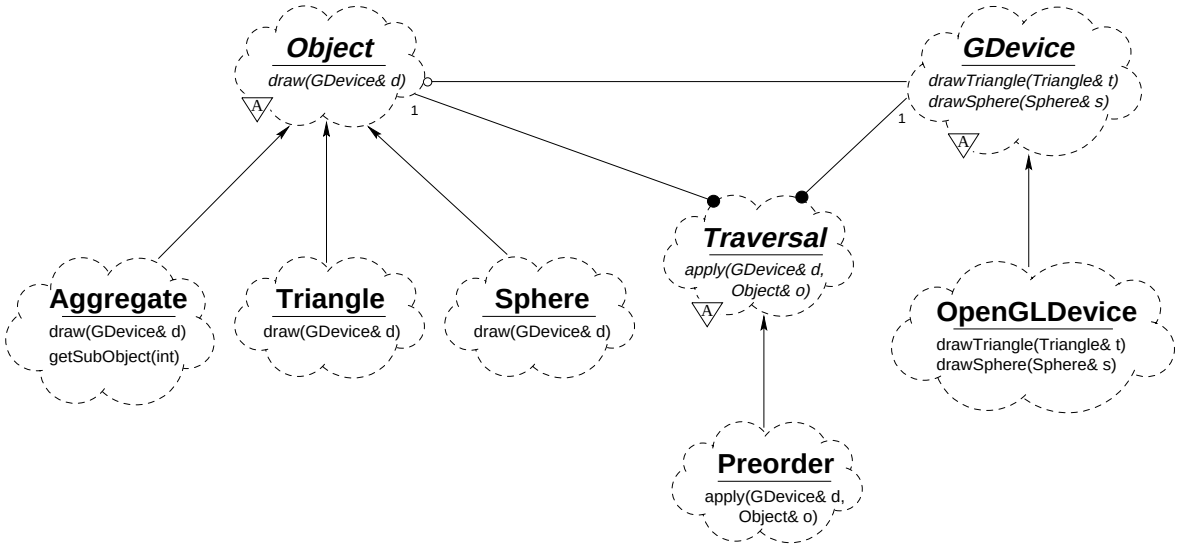


Abbildung 4.9
Durch eine Aufteilung in Objekt-, Darstellungs- und Traversierungshierarchien wird ein Höchstmass an Flexibilität erreicht.

Das resultierende System erlaubt nun die flexible Erweiterung der unterstützten geometrischen Objekte, der Darstellungs- und Traversierungsstrategie. Um Anpassungen vornehmen zu können, ist allerdings ein gründliches Verständnis dieser Mechanismen unerlässlich und entspricht einer White-Box-Erweiterung des Systems. Der nächste Abschnitt führt nun eine zusätzliche Abstraktionsebene ein, die ein Black-Box-Sicht auf ein Grafiksystem erlaubt.

4.2.3 Komponentenorientierte Grafiksysteme

Wie in Abschnitt 2.2.4 diskutiert, entwickeln sich White-Box-häufig zu Black-Box-Frameworks mit einer höhere Abstraktionsstufe. Diese Tendenz ist bei Grafikframeworks noch kaum ausgeprägt. Nur wenige Systeme bieten einen entsprechenden Zugang, wie zum Beispiel OpenInventor (Abschnitt 4.3.3) oder der angehende ISO Standard PREMO (Abschnitt 4.3.4). Nichtsdestotrotz bietet die zusätzliche Abstraktionsschicht wesentliche Vorteile und es ist zu erwarten, dass neuere Entwicklungen entsprechende Zugänge anbieten werden.

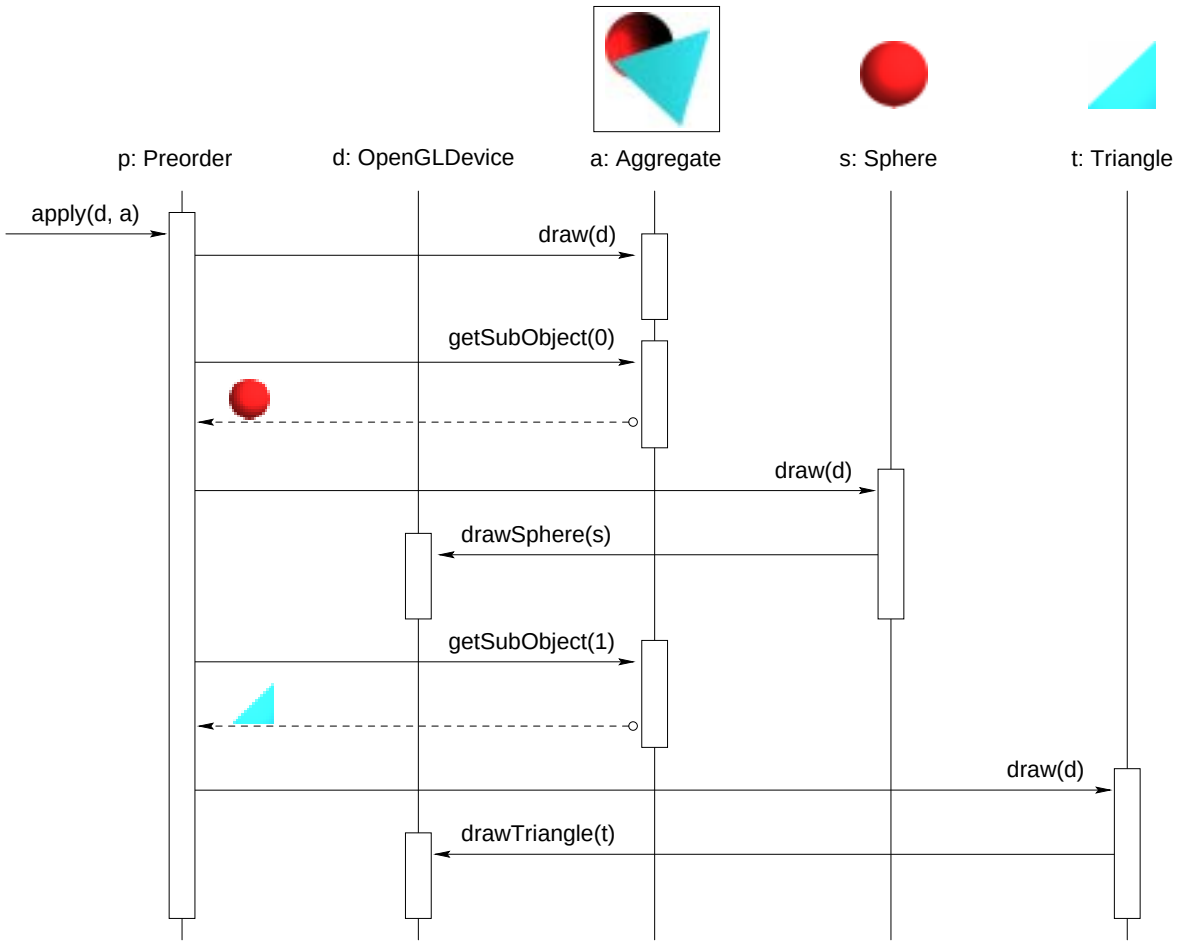


Abbildung 4.10
Interaktionsdiagramm der Darstellung einer einfachen Szene.

Visualisierungssysteme sind wesentlich innovativer und bieten bereits seit langem auch visuelle, auf Komponenten beruhende Kompositionsmechanismen an. Abbildung 4.11 zeigt die grafische Beschreibung einer Applikation am Beispiel von KHOROS [KR94, YAK95].

Grundsätzlich besitzt ein komponentenorientiertes Grafiksystem eine Schichtenarchitektur bestehend aus *Bibliotheksschicht* (Grafikbibliothek), *Frameworkschicht* (White-Box-Grafikframework) und *Komponentenschicht* (Black-Box-Grafikframework). Abbildung 4.12 zeigt den typischen Aufbau schematisch auf.

Die drei Schichten bieten unterschiedliche Zugänge zum Grafiksy-

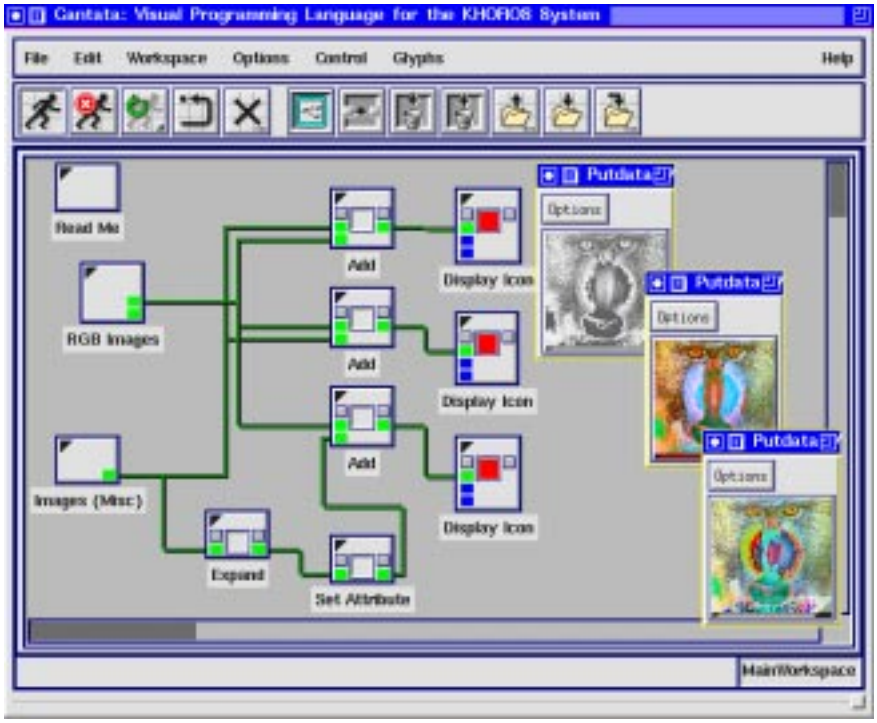


Abbildung 4.11 *Visuelle, komponentenorientierte Applikationsentwicklung am Beispiel von* KHOROS.

stem und haben klar umrissene Aufgabenbereiche:

Bibliotheksschicht
: Realisierung der Darstellung, entspricht den Environments Viewing, Logical und Realization des CGRM Standards. Ein Benutzer des Grafiksystems wird kaum mit dieser Ebene in Berührung kommen, ausser er nimmt Erweiterungen der Frameworkschicht vor.

Frameworkschicht
: Sie realisiert Objekt-, Darstellungs- und Traversierungshierarchien, entspricht also den Construction und Virtual Environments des CGRM Standards. Eine Erweiterung auf dieser Ebene entspricht der White-Box-Erweiterung eines Frameworks, was Kenntnisse der internen Mechanismen voraussetzt.

  Häufig wird zwischen Frameworks- und Bibliotheksschicht ein Adapterkonzept eingesetzt [GHJV95, *Bridge* Design Pattern, Seite 151ff], das eine beliebige Austauschbarkeit der zugrundeliegenden Darstellungsmöglichkeiten erlaubt, die durch die verwendete Grafikbibliothek gegeben sind. Dieser Ansatz wird

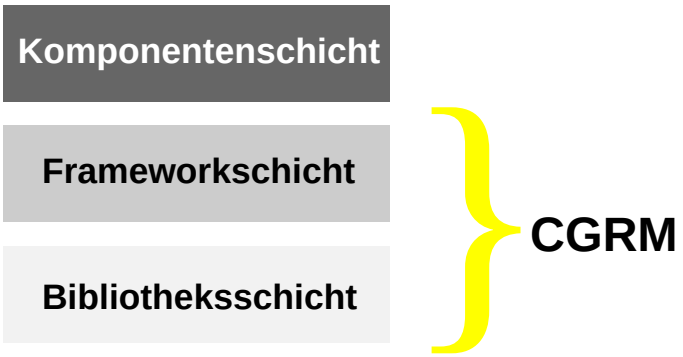


Abbildung 4.12 *Schematischer Aufbau eines komponentenorientierten Grafiksystems.*

unter anderem von ET++ [WGM88, Gam91] und dessen Multimedia-Erweiterung MET++ [AE93] verfolgt.

Eine weitere Aufgabe der Frameworkschicht ist die Unterstützung bei der Entwicklung neuer Black-Box--Abstraktionen der Komponentenebene.

Komponentenschicht

Sie stellt verschiedene Applikationskomponenten zur Verfügung, die durch einen Kompositionsmechanismus zu Applikationen zusammengefügt werden. Die vorgefertigten Komponenten bauen auf der Frameworkschicht auf und können sehr unterschiedliche Granularitäten aufweisen. Zum Beispiel enthält eine Komponente die Funktionalität eines gesamten Grafiksystem oder realisiert lediglich ein einzelnes Environment des CGRM Standards.

Die Vorteile eines komponentenorientierten Grafikframeworks liegen vor allem in den unterschiedlichen Schichten, die einen jeweils angepassten Zugang zur Funktionalität erlauben (siehe auch Kapitel 2 ab Seite 17). Erst die Kombination von Black- und White-Box-Sicht auf ein Grafiksystem erlauben das einfache Zusammenbauen von Applikationen aus bestehenden Komponenten *und* die Erweiterung und Anpassung bestehender Mechanismen.

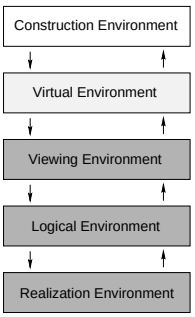
4.3 Beurteilung konkreter Grafiksysteme

Die vorangegangenen Abschnitte haben zwei Möglichkeiten für den Vergleich von Grafiksystemen eingeführt. Die folgenden Ausführungen verwenden die erarbeiteten Kriterien für die Untersuchung von vier typischen Vertretern solcher Systeme. Die Auswahl enthält mit OpenGL (Abschnitt 4.3.1) und OpenInventor (Abschnitt 4.3.3) die zwei zur Zeit wohl bekanntesten Grafiksysteme. Weniger bekannt ist Generic-3D (Abschnitt 4.3.2) und PREMO (Abschnitt 4.3.4). Sie besitzen aber innovative Merkmale, die auch den Entwurf von Booga beinflusst haben.

Die Reihenfolge der Beschreibung orientiert sich am zugrundeliegenden Design und der verwendeten Implementierungstechnik. Beginnend mit Grafikbibliotheken werden typische Grafikframeworks und ein Vertreter der komponentenorientierten Architekturen vorgestellt. Als Orientierungshilfe wird die Einordnung des jeweils untersuchten Grafiksystems im Randbereich dargestellt. Zum einen ist dies die Zuordnung zu den Kategorien Grafikbibliothek, Grafikframework oder komponentenorientiertes Framework. Zum anderen wird eine Abbildung auf die fünf Environments des CGRM Standards grafisch dargestellt. Während dunkel eingefärbte Environments eine Unterstützung der Konzepte der entsprechenden Umgebung symbolisieren, kennzeichnet eine helle Farbe das teilweise und keine Einfärbung das Fehlen entsprechender Mechanismen im betrachteten Grafiksystem.

4.3.1 OpenGL

Grafikbibliothek



OpenGL [Arc92, NDW93] ist ein typischer Vertreter einer Grafikbibliothek. Ursprünglich ausschliesslich für Computersysteme von Silicon Graphics ausgelegt und für deren Grafikhardware optimiert, hat sich OpenGL zum Defacto-Standard gemausert, der auf allen gängigen Plattformen und Betriebssystemen lauffähig ist. Die folgende kleine Beispielapplikation soll ein Gefühl für die Programmierung mit OpenGL geben. Sie initialisiert zuerst die virtuelle Kamera und zeichnet dann einen Torus, einen Kegel und eine Kugel. In Abbildung 4.13 ist das Resultat dieser Applikation dargestellt.

```
#include <GL/glut.h>

void display()
{
  // Viewing Parameter einstellen.
  glViewport(0, 0, 500, 400);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-2.5, 2.5, -1.8, 1.8, -10.0, 10.0);

  glClearColor(1.0, 1.0, 1.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // Globale Modelltransformation auf den
  // Transformationsstack schieben.
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glRotatef(20.0, 1.0, 0.0, 0.0);

  // Zeichne Torus.
  glPushMatrix();
    glTranslatef(-0.75, 0.5, 0.0);
    glRotatef(90.0, 1.0, 0.0, 0.0);
    glutSolidTorus(0.275, 0.85, 15, 15);
  glPopMatrix();

  // Zeichne Kegel.
  glPushMatrix();
    glTranslatef(-0.75, -0.5, 0.0);
    glRotatef(270.0, 1.0, 0.0, 0.0);
    glutSolidCone(1.0, 2.0, 15, 15);
  glPopMatrix();

  // Zeichne Kugel.
  glPushMatrix();
    glTranslatef(0.75, 0.0, -1.0);
    glutSolidSphere(1.0, 20, 20);
  glPopMatrix();

  // Globale Modelltransformation vom
  // Transformationsstack entfernen.
  glPopMatrix();
  glFlush();
}
```

OpenGL kann lediglich Punkte, Strecken und konvexe Polygone verarbeiten. Alle anderen geometrischen Objekte müssen aus diesen Primitiven zusammengesetzt werden. Die grundlegende Datenstruktur ist der *Vertex*, der zur Definition von einzelnen Punkten, Anfangs- und Endpunkten einer Strecke oder als Eckpunkt eines Polygons dient. Jeder Vertex enthält Positions-, Normalen-, Farb- und Texturinformation und wird in einem klar definierten Schema über mehrere Stufen von Modell- zu physischen Bildschirmkoordi-

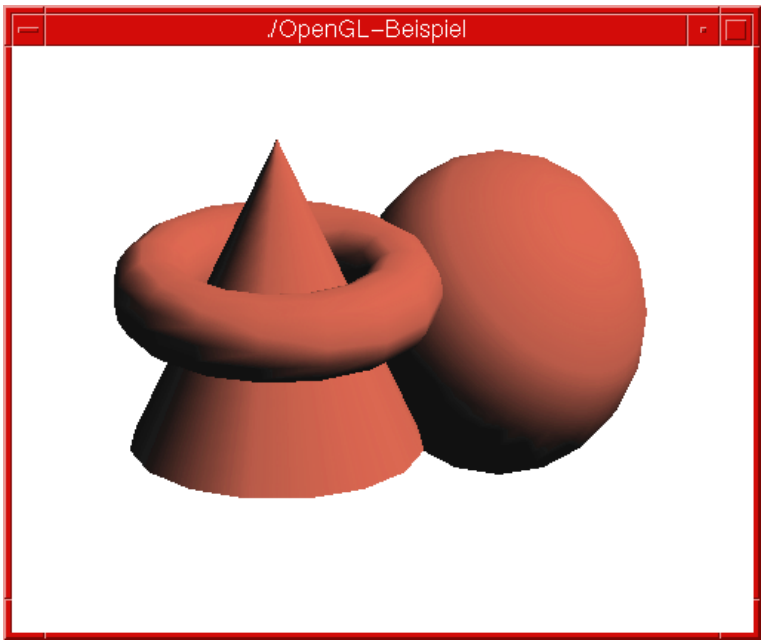


Abbildung 4.13
Eine einfache grafische Ausgabe realisiert mit der Grafikbibliothek OpenGL.

naten transformiert. OpenGL erlaubt eine direkte Manipulation der grundlegenden Mechanismen der *Rendering Engine*. So kann zum Beispiel die verwendete Beleuchtungsgleichung den eigenen Bedürfnissen angepasst werden. Die Qualität der Darstellung geht dabei allerdings nie über ein Gouraud Shading hinaus.

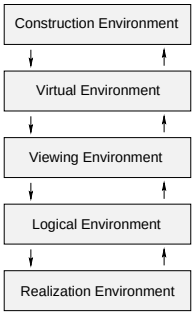
Da OpenGL nur einen sehr eingeschränkten Satz von geometrischen Objekten unterstützt, existieren viele Bibliotheken die weitere Objekttypen aber auch andere Funktionen zur Verfügung stellen, wie die Kopplung von OpenGL an ein spezifisches Windowing-System. Die bekanntesten sind die *OpenGL Utility Library* (GLU, [Arc92]) und das *OpenGL Utility Toolkit* (GLUT, [Kil94]), die unterem anderem Befehle zur Darstellung von komplexeren geometrischen Objekten wie Kugeln, Zylindern aber auch NURBS-Flächen anbieten. Für Abbildung 4.13 wurde GLUT verwendet, um "Nicht-Standardobjekte" zu erzeugen.

Verglichen mit dem Modell eines Grafiksystems des CGRM Standards fehlen in OpenGL das Construction Environment vollständig und das Virtual Environment ist durch die Unterstützung von Displaylisten nur in Ansätzen vorhanden. Dieser Vergleich zeigt auf, dass sich OpenGL auf die Darstellung von geometrischen Objekten beschränkt und dient deshalb oft als Grafikbibliothek von Grafikframeworks [SC92, KW93]. Prominentester Vertreter ist sicherlich OpenInventor (siehe Abschnitt 4.3.3), welches im we-

sentlichen ein Programmiermodell für OpenGL einführt und dessen Fähigkeiten voll auszunutzen vermag.

4.3.2 Generic-3D

Grafikframework



Generic-3D [BB95, Bei96] ist ein neuartiger Typ eines *generischen Grafikframeworks*. Das Framework bietet lediglich generische Funktionen eines Grafiksystems an. Somit kann es nur als Basis für eine konkretes System dienen, das durch instanzieren und konfigurieren erzeugt werden muss. So besitzt Generic-3D zum Beispiel keine geometrischen Objekte. Diese werden, wie in Abbildung 4.14 dargestellt, im Baukastenprizip aus den vorgefertigten Elementen für ein konkretes System zusammengebaut. Alle anderen Elemente eines Grafiksystems können auf ähnliche Art und Weise gebildet werden.

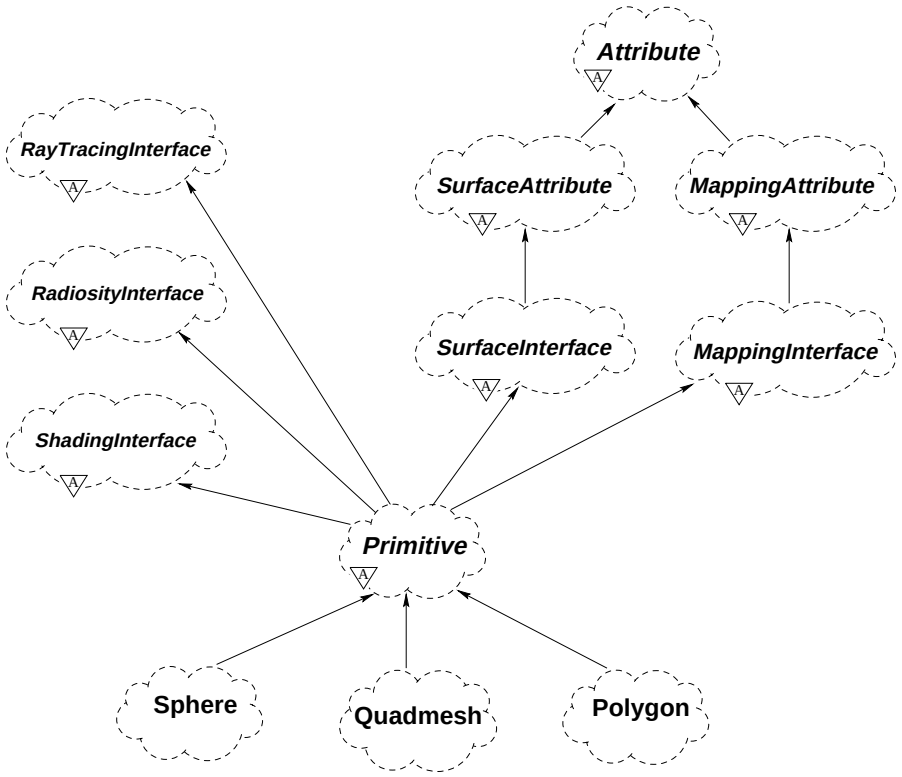


Abbildung 4.14 *Konfiguration von geometrischen Objekten durch das Instanzieren von generischen Vorlagen in* Generic-3D.

Zur Zeit existieren mehrere Implementationen von Grafiksystemen, die aus Generic-3D erzeugt wurden:

GX ein erweiterbarer Raytracing-Kernel,

GG ein einfaches GIS-System,

EGR GF ein kommerzielles System speziell abgestimmt für die europäische Möbelindustrie,

MAF ein verteiltes Multimedia-Framework und

EGR TIGER eine interpretierte OpenGL-Umgebung, die in der Lehre eingesetzt wird.

Implementiert ist Generic-3D in C++ unter intensiver Nutzung von Templates, um das generische Konzept zu realisieren. Die vollständigen Sourcen sind unter

`ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/GENERIC/`

frei erhältlich.

Generic-3D ist nur schwer mit dem CGRM-Modell eines Grafiksystems vergleichbar. Führt man nun trotzdem eine Abbildung auf die Environments des CGRM Standards durch, so sind zumindest Teile der Funktionalität der ersten drei Umgebungen abgedeckt (Construction, Virtual und Viewing Environments). Eine konkrete Instanzierung von Generic-3D kann aber über Elemente aller Evironments verfügen.

Beurteilung

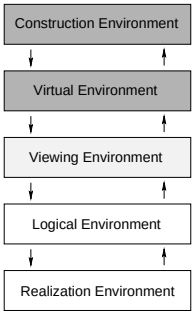
Obwohl der Generic-3D zugrundeliegende Ansatz im Grafikbereich[2] noch kaum eingesetzt wurde und sehr innovativen Charakter hat, sind die restlichen Merkmale des Systems eher konventionell. Zum Beispiel ist die Darstellungsstrategie direkt in den aus den generischen Strukturen erzeugten geometrischen Objekte realisiert (siehe Abbildung 4.14, Rendering Interfaces), anstatt eigene Abstraktionen für die Darstellung und Szenentraversierung anzubieten.

[2]Der generische Ansatz von Generic-3D folgt weitestgehend dem Muster der STL Bibliothek. Die *Standard Template Library* (STL) [MS96], integraler Bestandteil des kommenden ANSI C++ Standards, ist eine Sammlung von generischen Behältern und Algorithmen. STL ist ein bekanntes Beispiel eines erfolgreichen, generischen Frameworks.

Die Idee eines generischen Grafikframeworks ist vielversprechend und es ist zu hoffen, dass dieser Ansatz auch in andere Grafiksysteme einfliessen wird.

4.3.3 OpenInventor

Grafikframework (mit Komponenten)



OpenInventor [Ope94, Wer94b, Wer94a, SC92, Str93a] ist ein leistungsfähiges, in C++ realisiertes Framework für 3D-Grafik und Animation, das OpenGL zur Generierung der Ausgabe verwendet. Die eigentliche Darstellung wird mit Hilfe von OpenGL realisiert. Ähnlich CGRM ist auch OpenInventor in Schichten aufgebaut (Abbildung 4.15).

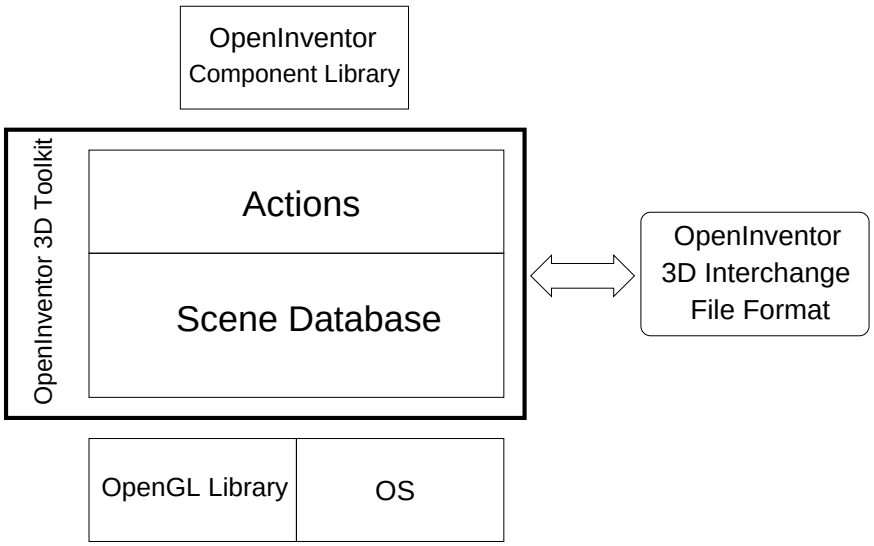


Abbildung 4.15 OpenInventor *folgt einem Schichtenmodell.*

Grundsätzlich werden die Elemente *Database Primitives*, *Actions* und *Components* unterschieden:

Database Primitives

Die Datenbasis setzt sich aus verschiedenen *Nodes* zusammen, die als azyklischer, gerichteter Graph organisiert sind, d.h. mehrfache Referenzierung von Objekten ist möglich. Nodes repräsentieren geometrische Objekte (Primitive und Aggregate), Transformationen und Elemente zur Konfiguration der Darstellung. Der folgende C++ Code baut eine einfache Szene bestehend aus 8 Nodes auf:

```
SoSeparator *root, *sep1, *sep2;
SoBaseColor *b1, *b2;
SoSphere    *sphere;
SoTransform *xf;

// Teilgraph mit Kugel erzeugen.
sep1 = new SoSeparator;
b1 = new SoBaseColor;
b1->rgb.setValue(1.0, 0.2, 0.2);
xf = new SoTransform;
xf->translation.setValue(0.0, 3.0, 0.0);
sphere = new SoSphere;
sphere->radius = 0.3;
sep1->addChild(b1);
sep1->addChild(xf);
sep1->addChild(sphere);

// Teilgraph mit Wuerfel erzeugen.
sep2 = new SoSeparator;
b2 = new SoBaseColor;
b2->rgb.setValue(0.2, 0.2, 1.0);
sep2->addChild(b2);
sep2->addChild(new SoCube);

// Alles zusammensetzen.
root = new SoSeparator;
root->ref();
root->addChild(sep1);
root->addChild(sep2);
```

Abbildung 4.16 stellt dieselbe Szene grafisch dar und folgt dabei der von den Autoren von OpenInventor vorgeschlagenen grafischen Notation, durch die auch komplexe Szenengraphen anschaulich präsentiert werden können.

Das Beispiel zeigt deutlich, dass geometrische Objekte lediglich Attribute zur Beschreibung ihrer *geometrischen Form* beinhalten. Alle anderen Attribute sind als eigene Abstraktionen realisiert. Dies entspricht einem konsequenten Einsatz objektorientierter Konzepte, wie es auch in Vision vorgeschlagen wird [SS95]. Allerdings hat dieses Vorgehen den Nachteil, dass der Programmier wissen muss wo im Szenengraph das jeweils gültige Attribut steht, bevor er es verändern kann. Zudem ist die Reihenfolge beim Einfügen von Nodes wichtig, beeinflusst doch zum Beispiel eine Transformation *alle* nachfolgenden Knoten.

Actions

Nachdem eine Szene aufgebaut wurde, können Operationen (Actions) darauf angewendet werden. Auf diese Weise wird die Darstellung realisiert (`SoGLRenderingAction`), ein Picking durchgeführt

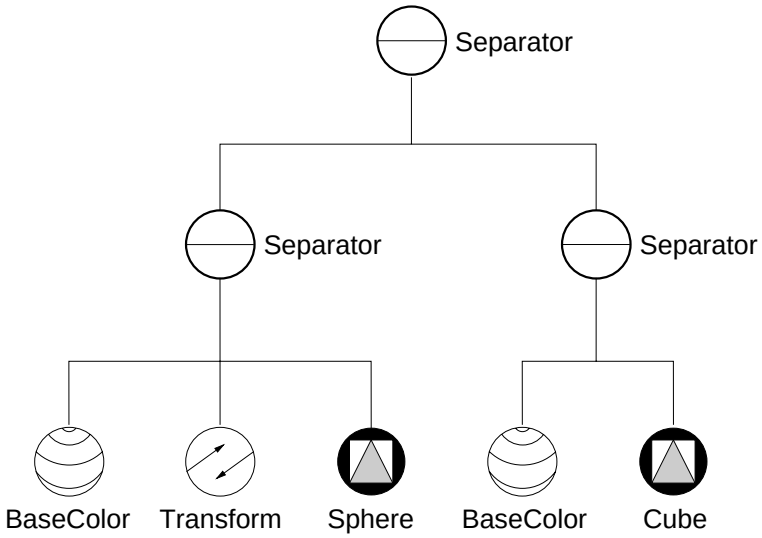


Abbildung 4.16
Ein einfacher OpenInventor *Szenengraph.*

(`SoRayPickAction`) oder auch die Szene in eine Datei geschrieben (`SoWriteAction`).

Actions werden auf alle Nodes in einem Szenengraph angewendet. Die dafür notwendigen Traversierungsalgorithmen sind allerdings nicht als eigene Abstraktionen realisiert, sondern werden von den Nodes selbst ausgeführt.

Components

Elemente der Komponentenschicht von OpenInventor sind vorgefertigte, interaktive Applikationsteile. Sie bestehen immer aus den zwei Elementen Benutzerinterface und Darstellungsbereich. Beispiele für Komponenten sind Material- und Lichtquelleneditor oder verschiedene Szenenviewer. Die folgende Applikation verarbeitet eine externe Szenendefinition und stellt die darin enthaltenen Objekte mit Hilfe einer solchen Viewer-Komponeten in einem Fenster dar:

```
int main(int argc, char* argv[])
{
  // Umgebung initialisieren.
  Widget appWindow = SoWin::init(argv[0]);

  // Szenenbeschreibung einlesen.
  SoInput in;
  SoSeparator* root = SoDB::readAll(&in);
  root->ref();

  // Viewer-Komponente erzeugen und Szene darstellen.
  SoWinExaminerViewer* viewer =
                  new SoWinExaminerViewer;
  viewer->setSceneGraph(root);
```

```
    viewer->build(appWindow);
    viewer->show();
    SoWin::show(appWindow);
    SoWin::mainLoop();
    return 0;
}
```

Abbildung 4.17 zeigt das Resultat dieser kleinen Anwendung.

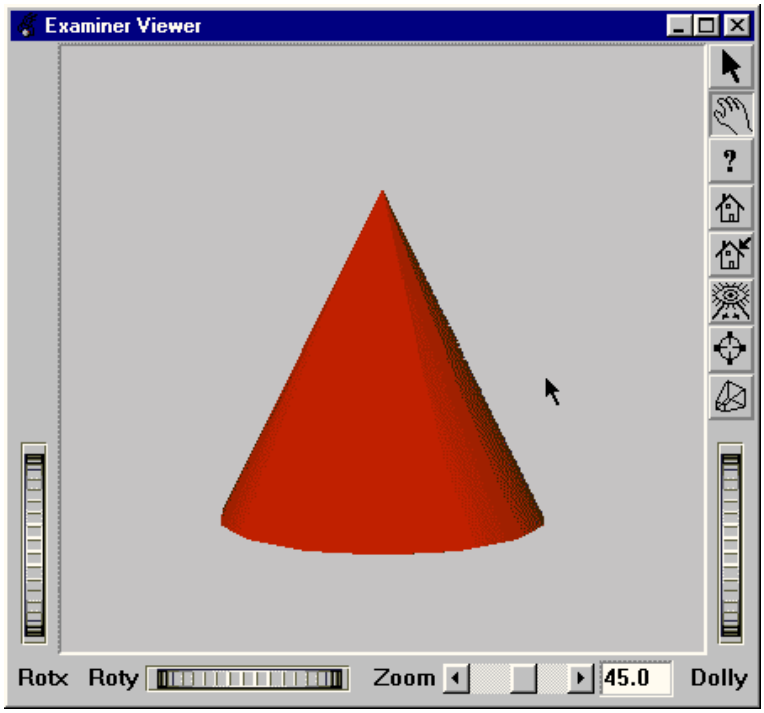


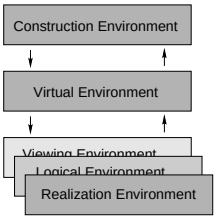
Abbildung 4.17
Darstellung der im Text beschriebenen OpenInventor-*Anwendung.*

OpenInventor definiert auch ein 3D-Dateiformat zur persistenten Speicherung von Szenenbeschreibungen, welches als Grundlage für die Entwicklung der Virtual Reality Modeling Language (VRML, [BPP96]) gedient hat.

OpenInventor bietet Funktionen, die den Construction und Virtual Environments des CGRM Standards entsprechen. Die anderen Schichten werden durch den Einsatz von OpenGL als Darstellungsbibliothek abgedeckt. Diese Trennung ist ein typisches Merkmal von Grafikframeworks (siehe dazu auch Abschnitt 4.2.2).

4.3.4 PREMO

Komponenten-orientiertes Framework



PREMO (Presentation Environment for Multimedia Objects) [ISO9x, DDtHR94, Wis95] ist ein neuer ISO Standard. Zum Zeitpunkt der Entstehung dieser Arbeit war die in 1992 begonnene Entwicklung allerdings immer noch im Fluss. Das Ziel ist die Definition einer standardisierten Umgebung für die Präsentation von multimedialen Daten. Unterstützt werden sollen die Medientypen Computergrafik, Animation, synthetisch erzeugte Bilder, Audio, Text, Bilder, bewegte Bilder inklusive Video, Bilder von Bildverarbeitungsoperationen und alle anderen darstellbaren Medientypen oder Kombinationen davon.

Zentrales Merkmal des Entwurfs ist die Verwendung des objektorientierten Ansatzes. Im Unterschied zu bisherigen Standards erlaubt dies eine flexible Erweiterung und Anpassung der Basisfunktionalität.

Die fünf Environments des CGRM Standards wurden in PREMO zu drei Gruppen zusammengefasst, dem

Modelling Environment,
: das dem Construction Environment des CGRM Standards entspricht und applikationsspezifische Modellierungskomponenten beinhaltet, dem

Virtual Environment,
: mit Szenen als Sammlung von Primitiven zur Darstellung bestimmt, welche zum Beispiel durchaus mit akustischen Ausgabegeräten (Sound Renderering) erfolgen kann. Die Szene ist zudem der Mittler zwischen verschiedenen Modellierungs- und Darstellungskomponenten, die ihrerseits im

Presentation Environment
: enthalten sind.

Abbildung 4.18 zeigt die beschriebenen Zusammenhänge grafisch auf. Einige Beispiele von Komponenten des Modelling und Presentation Environments sind angedeutet und die Mittlerrolle der Szene zwischen diesen Environments wird offensichtlich.

Abbildung 4.19 zeigt den prinzipiellen Aufbau einer PREMO Applikation. *Modeller* sind Objekte, die als Eingabeparameter applikationsspezifische oder PREMO-Primitive erhalten und daraus eine

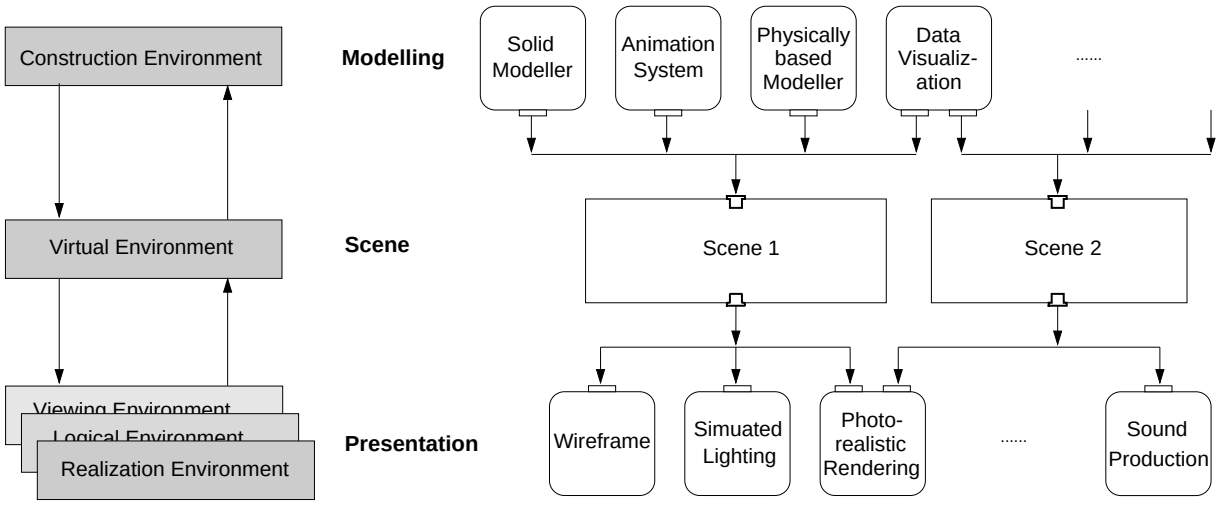


Abbildung 4.18
Die Environments des CGRM *Standards, wie sie in* PREMO *aufgefasst werden.*

Scene erzeugen, die nun ausschliesslich PREMO Primitive enthält. *Renderer* sind aus konzeptioneller Sicht ebenfalls Modeller, die allerdings als Eingabeparameter nur PREMO Primitive akzeptieren. Üblicherweise ist ein Renderer mit einem Ausgabegeräte assoziiert und wandelt die Eingabeprimitive in ein geeignetes Format um, so dass eine Präsentation durch das Ausgabegerät möglich wird.

Durch das Zusammenfügen mehrerer Modeller und Renderer können ganze Netzwerke gebildet werden, wie dies in Abbildung 4.20 am Beispiel eines Systems, das Audio- und Grafikdaten verarbeitet demonstriert wird. Solche Systeme werden in PREMO *MRI Netzwerke* (Modelling, Rendering and Interaction) genannt. Eine Verfeinerung des Beispiels ist in Abbildung 4.21 zu sehen. Die zusätzliche Eingabeverarbeitung (grau hinterlegt), erlaubt die interaktive Manipulation verschiedener Komponenten. Das *Router* Element leitet den Eingabestrom an die richtige Adresse, abhängig vom aktuellen Zustand des Systems.

Wie man aus den Abbildungen entnehmen kann, basiert PREMO auf einem Komponentenmodell, in dem die verschiedenen Modeller und Renderer die Rolle der Komponenten einnehmen. Im PREMO Standard wird der Begriff der Komponenten benutzt, allerdings auf völlig andere Weise aufgefasst:

> *"Components are collections of object and non-object*

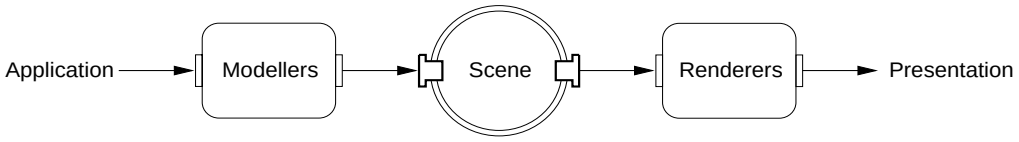


Abbildung 4.19
Der prinzipielle Aufbau einer Applikation in PREMO.

> *types that together provide certain* services *to the system." [DDtHR94]*

"Komponenten" werden also als Subsysteme (Klassenbibliotheken) aufgefasst, was mit der Komponentendefinition aus Abschnitt 2.2.5 nicht in Einklang zu bringen ist. Zur Zeit sind drei dieser "Komponenten" in einer ersten Fassung definiert:

1. **Foundation Component**
   Entspricht einer universellen und teilweise generischen Klassenbibliothek, die neben Aggregatstrukturen (Mengen, Listen, Stacks, ...) auch verschiedene Basisabstraktionen für die darauf aufbauenden Komponenten definiert.

2. **Multimedia System Services Component (MSS)**
   Definiert eine Menge von Standarddiensten und -objekten, die als Basis für die Entwicklung von interaktiven und verteilten Multimediaanwendungen nützlich sind.

3. **Modelling, Rendering, and Interaction Component (MRI)**
   Beschreibt Objekttypen für die Modellierung und Präsentation von und Interaktionen mit multimedialen Daten. Zusätzlich werden alle Elemente zur Beschreibung von Szenen, wie geometrische Primitive und Transformationen, definiert.

Beurteilung

Die Ziele von PREMO sind sehr hoch gesteckt und die Spezifikation ist noch nicht besonders weit gediehen. Lediglich die grundlegendsten Interfaces sind definiert und konkrete Realisierungen sind noch ausstehend.

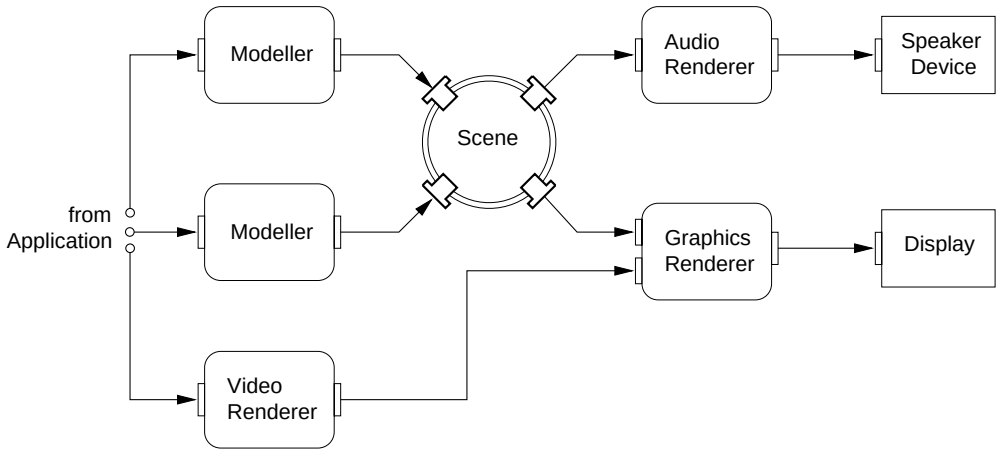


Abbildung 4.20
Ein einfaches MRI (Modelling, Rendering and Interaction) Netzwerk in PREMO.

Eine Implementation würde wohl aus einer Reihen von Frameworks bestehen, die die vielen Teilaspekte von PREMO abzudecken vermögen. Dies hat einerseits den Vorteil, dass eine klare Trennung der verschiedenen Bereiche vorgenommen würde, führt aber andererseits zu einem erhöhten Einarbeitungsaufwand.

Zur Zeit wird noch diskutiert, wie generische Renderer implementiert werden müssen, die mit *allen* Medientypen umzugehen vermögen [Wis95]. Die bisher gemachten Vorschläge basieren auf bekannten Ideen, wie die Konvertierung in alternative Repräsentationen, was allerdings bei all den verschiedenen Medientypen äusserst schwierig zu realisieren sein wird.

Da PREMO einen objektorientierten Ansatz verfolgt, werden Anpassungen und flexible Erweiterungen möglich und sind sogar erwünscht. Dies steht im krassen Gegensatz zu bisherigen Standards wie GKS oder PHIGS, die häufig als zu starr empfunden wurden.

Die Idee der Applikationsnetzwerke von Modeller, Szenen und Renderer ist vielversprechend und typisch für komponentenorientierte Frameworks. Im Fall von PREMO scheinen die einzelnen Elemente allerdings etwas zu grobkörnig zu sein, entspricht eine einzelne Renderer-Komponente doch zum Beispiel einer kompletten PHIGS Implementation.

Wann eine fertige Spezifikation vorliegen wird, ist zur Zeit noch unklar und es wird sich bis dahin mit Bestimmtheit noch sehr viel

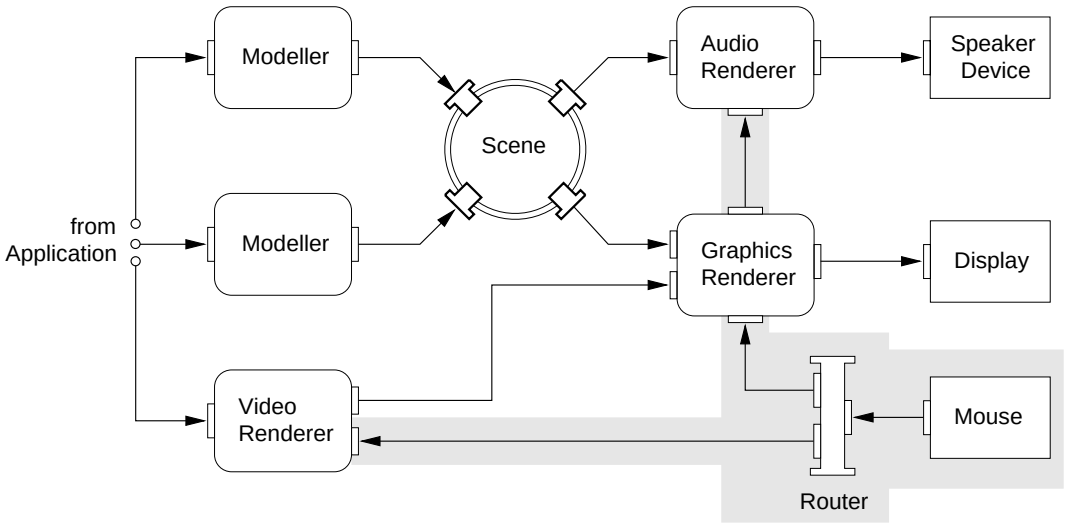


Abbildung 4.21
Ein MRI Netzwerk mit Interaktionselementen.

an den heutigen Vorschlägen ändern, wie dies auch schon in den letzten Jahren der Fall gewesen ist.

4.4 Schlussfolgerungen

Der Vergleich von Grafiksystemen ist schwierig, da die jeweiligen Merkmale sehr unterschiedlich sind. Nichtsdestotrotz lohnt sich die Untersuchung bestehender Systeme. Erst auf diese Weise können grundlegende Konzepte identifiziert werden, die sich bei der Realisierung von Grafiksystemen als geeignet erwiesen haben. Der CGRM Standard ist hier kaum hilfreich. Durch die starre Aufteilung in die fünf Environments kann auf der einen Seite ein existierendes Systems recht gut bezüglich der unterstützten Elemente der Darstellungs-Pipeline beschrieben werden. Viele Methoden lassen sich andererseits aber nur schwer auf dieses Schema abbilden. Der verallgemeinerte Ansatz von PREMO lässt sich da viel einfacher in einem konkreten System umsetzen.

Der konsequente Einsatz der Objekttechnologie, scheint für die Realisierung von Grafiksystemen Vorteile zu bieten. Ein gutes Grafiksystem muss mehrere gegensätzliche Anforderungen erfüllen. So soll es in möglichst vielen Bereichen flexibel erweiterbar sein, aber gleich-

zeitig auch einfach erlernbar bleiben. Diese Anforderungen können durch eine geeignete Architektur des Systems erfüllt werden:

Drei Basisabstraktionen
: Durch eine Dreiteilung in Objekt-, Darstellungs- und Traversierungshierarchie (siehe Abschnitt 4.2) wird ein Höchtsmass an Flexibilität erreicht. Mittels Vererbung lassen sich beliebige Erweiterungen oder Anpassungen vornehmen. Entsprechend implementiert, lassen sich die drei Hierarchien unabhängig voneinander erweitern, d.h. eine lokale Erweiterbarkeit ist möglich.

  Die angesprochene Flexibilität hat allerdings auch ihre Schattenseiten. Durch die drei Hierarchien entsteht eine recht komplexe Struktur, die doch eine grosse Einstiegshürde bildet. Das übliche Vorgehen zur Reduzierung der Komplexität ist die Einführungen einer Schichtenarchitektur.

Schichtenarchitektur
: Eine Unterteilung in Bibliotheks-, Framework- und Komponentenschicht, die sowohl eine White- und Black-Box-Sichtweise auf das Grafiksystem erlauben, bietet zum einen volle Flexibilität aber auch eine abstrahierte, vereinfachte Sicht auf die verschiedenen Elemente des Systems. Der Einstieg kann stufenweise erfolgen und erlaubt ein effizientes Arbeiten mit dem System innert nützlicher Frist. Zur Zeit sind im Grafikbereich allerdings nur einige wenige Systeme erhältlich, die dieses Konzept verwirklicht haben. Dies liegt vor allem an der Schwierigkeit, ein geeignetes Komponentenmodell zu finden.

Der obige Vorschlag entspricht im wesentlichen einem Komponentenframework, wie es schon im Kapitel 2 vorgeschlagen wurde. Die Zuordnung der drei Basisabtraktionen (Objekt-, Darstellungs- und Traversierungshierarchie) zu Bibliotheks-, Framework- oder Komponentenschicht muss allerdings erst noch genauer betrachtet werden und kann nicht direkt aus den bisher durchgeführten Untersuchungen abgeleitet werden. Das folgende Kapitel wird sich mit dieser Fragestellung beschäftigen.

Hiermit sind alle Konzepte eingeführt und alle Grundlagen erarbeitet, die für die Vorstellung von BOOGA in den nun folgenden Ka-

piteln benötigt werden. Der Ansatz des Komponentenframeworks bildet die Basis für die Softwarearchitektur, deren vermutete gute Eigenschaften sich in der Praxis bestätigt haben.

Grobdesign von BOOGA

5

Die vorangegangenen Kapitel haben die Grundlagen für die Beschreibung von BOOGA (*Berne's Object-Oriented Graphics Architecture*) [ASB96] präsentiert. Dieses Kapitel gibt nun einen Überblick über die grundlegenden Konzepte und stellt die logische und physische Architektur vor. Im weiteren gehen die Kapitel 6 bis 8 im einzelnen auf die verschiedenen Schichten von BOOGA ein, zeigen die wichtigsten Abstraktionen und Mechanismen auf und geben Einblicke in Teile der Implementierung.

5.1 Das Konzept

In Kapitel 1 wurde bereits detailliert auf die von BOOGA verfolgten Ziele eingegangen. Das wichtigste ist die Bereitstellung einer Forschungsplattform für die unterschiedlichen Gebiete der Computergrafik, wie sie in Kapitel 3 klassifiziert wurden. Weiter soll BOOGA möglichst flexibel an neue Gegebenheiten angepasst werden können und dennoch einen einfachen und effizienten Einstieg erlauben. Die letztere Anforderung impliziert die Architektur eines Komponentenframeworks (siehe Abschnitt 2.4, Seite 17), während die erste eine geeignete Klassifizierung des Grafikbereichs voraussetzt. Die Klassifizierung ist für ein einheitliches Modell unabdingbar, welches als Basis für die Architektur des Systems dient.

Die Klassifizierungen nach Meier (siehe Kapitel 3) geht von der Unterscheidung in Bild- und Objektraum aus. Dieses Schema ori-

entiert sich am Aufbau von Standardanwendungen im Grafikbereich. So fällt es nicht schwer, die klassische Darstellungspipeline von Rendering-Applikationen auf das Modell von Meier abzubilden. Auch die Algorithmen der Bildverarbeitung oder anderer Bereiche lassen sich leicht integrieren. Schwieriger ist aber die Einordnung von Anwendungen, die ausschliesslich im Objektraum arbeiten und aus 2D-Eingabedaten 3D-Strukturen erzeugen, wie dies zum Beispiel bei Rekonstruktionsverfahren häufig der Fall ist.

Das letzte Beispiel führt zu einem weiteren Ansatz: Die Unterscheidung wird bezüglich der Dimension der vorliegenden Datenstrukturen vorgenommen. Bild- und Objektraum werden in diesem Modell nicht unterschieden, solange die Objekte dieselbe Dimension besitzen. Diese Unterscheidung trägt auch dem Umstand Rechnung, dass viele Algorithmen im Grafikbereich für eine bestimmte Dimension ausgelegt sind. So sind beispielsweise gewisse Verfahren der Algorithmischen Geometrie auf den 2D-Bereich spezialisiert und kennen keine Entsprechung in höheren Dimensionen. Abbildung 5.1 zeigt ein aus diesem Ansatz abgeleitetes Konzept, das die Grundlage von BOOGA ist. Die Darstellung zeigt alle möglichen *Übergänge* zwischen den Datenstrukturen gleicher oder verschiedener Dimension. Die Begriffe sind bewusst sehr allgemein gewählt, um semantische Einschränkungen zu vermeiden. Die folgende *unvollständige* Auflistung gibt einige Beispiel für mögliche Operationen, die durch die verschiedenen Übergänge repräsentiert werden können:

Operation 3D → 3D
: Einlesen einer Szenenbeschreibung, Modellierungsoperationen, Berechnung von Formfaktoren für das Radiosity-Verfahren, Vereinfachung von Mesh-Strukturen, Simulationen.

Operation 3D → 2D
: Bilderzeugende Verfahren wie Raytracing oder Z-Buffering, Sketch-Rendering, grafische Visualisierung dreidimensionaler Datenstrukturen, Kameratransformationen.

Operation 2D → 2D
: Einlesen einer Szenenbeschreibung, Algorithmen der Bildverarbeitung, wie zum Beispiel Kantendetektion oder Farbreduktion, Algorithmen der Algorithmischen Geometrie wie das Finden einer Konvexen Hülle, Modellierungsoperationen, Darstellung von Rasterobjekten auf einem Ausgabegerät.

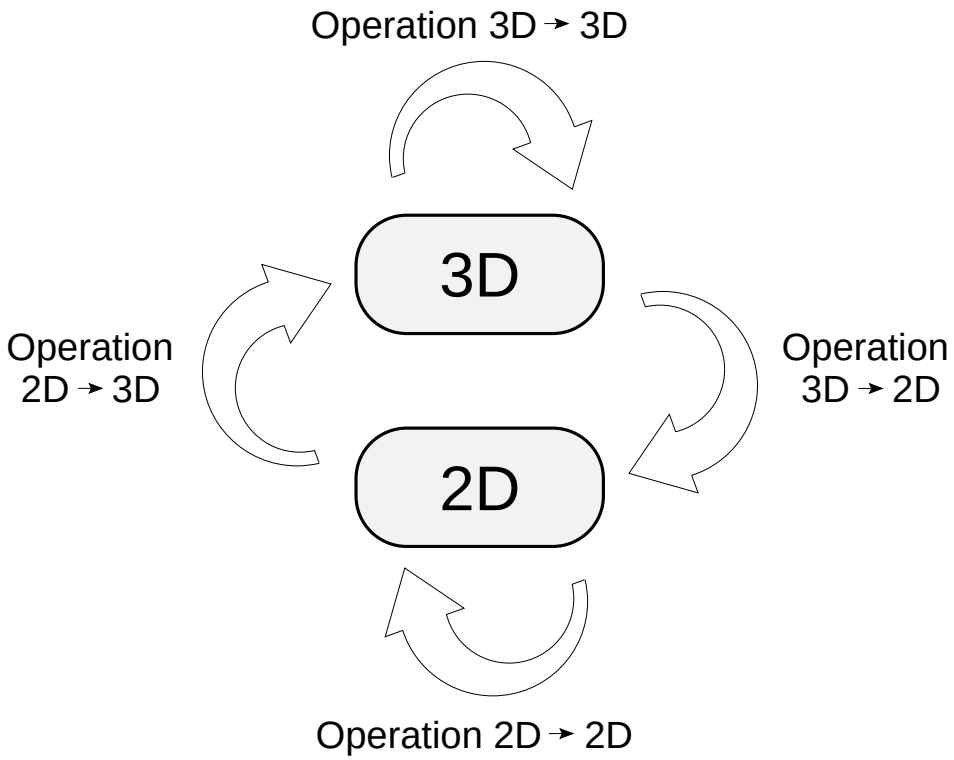


Abbildung 5.1
Die Unterscheidung der Dimension der geometrischen Datenstrukturen führt zum zugrundeliegenden Konzept von Booga.

Operation 2D → 3D
: Rekonstruktion dreidimensionaler Strukturen aus Bildinformation, Algorithmen der Bildanalyse, Generierung dreidimensionaler Modelle aus zweidimensionaler Information.

Das Konzept führt zu einer Zerlegung von Applikationen in eine Menge von Einzelschritten. Die Darstellung eines 3D-Modells mit dem Wireframe-Verfahren würde zum Beispiel in die folgenden Schritte unterteilt werden:

1. Einlesen der Szenenbeschreibung. [*Operation 3D → 3D*]
2. Konvertieren aller geometrischen Objekte der Szene in 3D-Geradensegmente. [*Operation 3D → 3D*]
3. Transformation der 3D- in 2D-Geradensegmente unter Berücksichtigung eines virtuellen Kameramodells. [*Operation 3D → 2D*]

4. Scankonvertierung der erhaltenen Geradensegmente in eine Pixmap. [*Operation 2D → 2D*]

5. Reduktion der Farbinformation der Pixmap zur Anpassung an die Eigenschaften des Ausgabegerätes. [*Operation 2D → 2D*]

6. Präsentation der resultieren Pixmap auf dem Bildschirm. [*Operation 2D → 2D*]

Die Zerlegung der Applikation in Einzelschritte orientiert sich an der Dimension der verarbeiteten Datenstrukturen. Dies entspricht einer traditionellen *Datenflussmodellierung*. Kapitel 9 zeigt an weiteren Beispielapplikationen die Zerlegung in primitive Operationen auf. Dies wird durch ein Vorgehensmodell für die Applikationsentwicklung weiter fundiert.

Aus dem vorgeschlagenen, sehr allgemeinen Konzept leitet sich die Architektur von BOOGA auf natürliche Weise ab. Die folgenden Abschnitte diskutieren nun ihre logischen und physischen Komponenten.

5.2 Logische Architektur

Unter der logischen Architektur eines objektorientierten Systems werden Entscheide zusammengefasst, die sich mit der Organisation von Klassen zu Hierarchien, der Plazierung von Methoden in Klassen oder allgemein dem Einsatz der zur Verfügung stehenden Sprachkonstrukte befassen. In BOOGA setzt sich die logische Architektur aus den drei Hauptabstraktionen `Component`, `World` und `Traversal` zusammen, die jeweils durch eigene Klassen realisiert sind. Abbildung 5.2 zeigt die Zusammenhänge als vereinfachtes Klassendiagramm grafisch auf.

Die Aufteilung in drei Abstraktionen ist eine Verallgemeinerung des im Abschnitt 4.2 auf Seite 34 vorgeschlagenen Prinzips der Trennung von Darstellungs- (`Component`), Objekt- (`World`) und Traversierungshierarchie (`Traversal`). Die einzelnen Abstraktionen sind wie folgt charakterisiert:

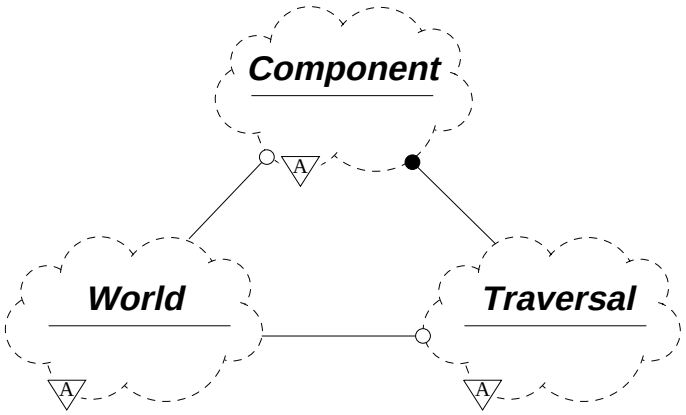


Abbildung 5.2
Die logische Architektur von BOOGA.

Component
: Die Abstraktion `Component` dient als Basis für die Modellierung der vier Übergänge des BOOGA-Konzeptes. Für jeden Übergang existiert eine zugehörige `Component`-Variante.

World
: Eine Welt entspricht einer Menge von 2D- oder 3D-Objekten. Sie stellt die Datenstrukturen zur Verfügung auf denen die Komponentenabstraktionen arbeiten. Elemente der Welt können geometrische Objekte aber auch beliebige andere Objekttypen sein. `World`-Abstraktionen existieren für den 2D- und 3D-Fall.

Traversal
: Die Abstraktion `Traversal` modelliert die Traversierungsstrategie. Diese wird von einer `Component` verwendet, um alle Elemente einer Welt in der durch die Traversierung bestimmten Reihenfolge zu besuchen. Auch bei dieser Abstraktion existieren Basisklassen für den 2D- und 3D-Fall.

Detailliertere Beschreibungen zu den drei Abstraktionen, sind in den Kapiteln 7 und 8 zu finden.

5.3 Physische Architektur

Die physische Architektur befasst sich mit der Organisation von Klassen zu zusammengehörigen Einheiten wie Module oder Bibliotheken. BOOGA folgt der Architektur eines Komponentenframe-

works und ist demzufolge in Bibliotheks-, Framework- und Komponentenschicht gegliedert. Die einzelnen Schichten bestehen wiederum aus mehreren physischen Bibliotheken[1], die eine jeweils abgeschlossene Aufgaben erfüllen. Abbildung 5.3 zeigt diese Schichtenarchitektur in einer grafischen Übersicht.

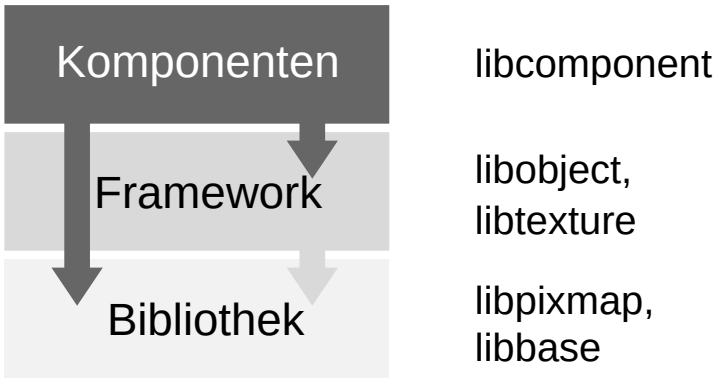


Abbildung 5.3 BOOGA *ist als Komponentenframework in Schichten unterteilt, die physisch als eigenständige Bibliotheken realisiert sind.*

Den einzelnen Schichten mit ihren physischen Bibliotheken sind klar umrissene Aufgabenbereiche zugeordnet:

Bibliotheksschicht

Die unterste Schicht von BOOGA besteht aus den Bibliotheken `libbase` und `libpixmap`, die grundlegende Datenstrukturen und Algorithmen für die übergeordneten Schichten bereitstellen. Unter anderem sind dies:

libbase

Generische Behälterklassen wie dynamische Liste und Symboltabelle; geometrische Datenstrukturen für Vektoren, Quaternionen und Transformationsmatrizen; verschiedene geometrische Algorithmen wie die Triangulierung beliebiger Polygone; Betriebssystem unabhängige Behandlung der Zeitmessung; Scankonvertierung von Ellipsen und Strecken.

[1] Der Begriff *Bibliothek* hat in diesem Zusammenhang eine Doppelbedeutung. Zum einen bezeichnet er die unterste Schicht eines Komponentenframeworks, zum anderen wird er für ein Element der physischen Architektur verwendet. Die jeweilige Bedeutung sollte aus dem Zusammenhang klar ersichtlich sein.

libpixmap
: Generische Strukturen für die Speicherung digitaler Bilder mit variabler Farbinformation und Unterstützung benutzerdefinierter Informationskanäle; verschiedene Operationen auf Pixmaps wie Gamma-Korrektur und Farbreduktion; Unterstützung für die Verarbeitung unterschiedlicher Bildformate.

Kapitel 6 geht im Detail auf die Bibliotheksschicht ein.

Frameworkschicht

Diese Ebene von BOOGA konzentriert sich ausschliesslich auf die Modellierung des Szenengraphen. Die einzelnen Protokolle und Mechanismen wurden so ausgelegt, dass Aufgaben der übergeordneten Komponentenschicht möglichst gut unterstützt werden.

Die Frameworkschicht ist ebenfalls in zwei Bibliotheken unterteilt. Dadurch wird eine saubere Trennung zwischen geometrischer Struktur und Attributen für die Beschreibung der Oberflächenbeschaffenheit einzelner Objekte erreicht. Dies führt zu den beiden Bibliotheken `libobject` und `libtexture`:

libobject
: Framework von geometrischen Objekten zur Modellierung des Szenengraphen. Dieser kann neben geometrischen Objekten auch andere Elemente wie Lichtquellen oder Kameras enthalten. Diverse Protokolle, wie zum Beispiel das Schneiden mit einem Strahl, werden von allen Objekten unterstützt. Etliche geometrische Objekte für den 2D- und 3D-Fall stehen dem Anwender zur Verfügung. Zudem ist das Framework für den Einbau neuer Objekttypen ausgelegt.

libtexture
: Texturen modellieren Oberflächeneigenschaften geometrischer Objekte. Sie erlauben die Variation der Darstellungsqualität von Rendering-Operationen in weiten Grenzen. Das Texturkonzept bleibt dabei nicht auf den 3D-Fall beschränkt, sondern steht gleichermassen im 2D-Fall zur Verfügung.

Kapitel 7 geht im Detail auf die Frameworkschicht ein.

Komponentenschicht

Die oberste Ebene bietet Abstraktionen für die Realisierung von Komponenten[2]. Beim Entwurf wurde auf ein einheitliches und einfaches Dienstinterface geachtet, das über alle Komponententypen hinweg gleich bleibt. Als Kompositionsmechanismus wird direkt die Implementierungssprache C++ verwendet. Alle Komponenten sind in der Bibliothek `libcomponent` organisiert. Unter anderem stehen folgende Elemente zur Verfügung:

libcomponent
: Klassen für die Modellierung der vier Übergänge des BOOGA-Konzeptes. Konkrete primitive Operationen, wie beispielsweise das Einlesen von Szeneninformation aus einer externen Repräsentation, das Durchsuchen des Szenengraphen nach Objekten eines bestimmten Typs, die Berechnung der Konvexen Hülle einer Punktmenge oder die Darstellung eines Szenengraphen mit dem Raytracing-Verfahren.

Kapitel 8 geht im Detail auf die Komponentenschicht ein.

5.4 Implementation

BOOGA ist in C++ [ES90, Str95] implementiert. Das Komponentenframework besteht aus über 200'000 Codezeilen (Kommentare und Beispielapplikation sind in dieser Zahl enthalten) verteilt auf mehr als 700 Klassen mit über 6'500 Methoden. Zur Zeit existieren mehr als 30 Applikationen, die die grundlegenden Konzepte von BOOGA mit Erfolg angewendet haben.

Die Verwendung der Entwicklungsumgebung SNiFF+ erlaubt uns, den Überblick über die Gesamtentwicklung zu behalten. Zudem bietet sie ein ausgefeiltes Versions- und Konfigurationsmangement an, das für die Koordination der vielen Entwickler unabdingbar geworden ist.

[2] Vgl. auch die Komponentendefinition aus Abschnitt 2.2.5 auf der Seite 13. Sie hat als Grundlage für die Entwicklung der BOOGA-Komponenten gedient.

Bibliotheksschicht von BOOGA

6

6.1 Überblick

Die Bibliotheksschicht von BOOGA ist eine Klassenbibliothek, d.h. eine lose Sammlung von Klassen, ausgelegt für die Unterstützung der darauf aufbauenden Schichten. Es existieren Abstraktionen für die Speicherung digitaler Bilder und Algorithmen für die Umwandlung verschiedener Bildformate in die BOOGA-eigene Repräsentation. Auch typische Bildverarbeitungsoperationen, wie die Reduktion der Farbinformation oder das Ausdünnen von Konturen sind unterstützt. Weiter umfasst der breite Funktionsumfang mathematische, insbesondere geometrische Funktionen, unterstützt verschiedene generische Datenbehälter, bietet aber auch Mechanismen für Zeitmessungen, die einheitliche Behandlung von Fehlermeldungen und die Konfiguration von Applikationen an.

Entscheidend für korrekte Resultate bei der Implementierung von geometrischen Algorithmen ist die Behandlung von möglichen Ungenauigkeiten, die in schwerwiegenden Fällen zu Inkonsistenzen führen können. Es gibt viele Vorgehensweisen, wie dieses Problem angepackt werden kann. Einerseits sind robuste Varianten für viele Algorithmen bekannt, die wenig anfällig auf Ungenauigkeiten sind. Andererseits versucht man mit verschiedenen Techniken, die Ungenauigkeiten selbst in den Griff zu bekommen [BW93]. BOOGA verfolgt einen pragmatischen Ansatz. Es wird der "naive" Standpunkt vertreten, dass sich Ungenauigkeiten nicht vermeiden lassen

und daher "erlaubt" sind. Alle arithmetischen Operationen berücksichtigen deshalb einfach eine frei konfigurierbare Ungenauigkeitskonstante (*fudge factor*). Zwei Zahlenwerte werden beispielsweise genau dann als gleich betrachtet, falls ihre Differenz in der durch die Ungenauigkeitskonstanten gebildeten Umgebung enthalten ist. Im weiteren werden Zahlenwerte in numerischen Berechnungen unter Verwendung des BOOGA-spezifischen Datentyps `Real` ausgeführt. Dieser lässt sich den jeweiligen Bedürfnissen anpassen. Ein Entwickler kann durch den Einsatz robuster Algorithmen die Qualität der Resultate weiter verbessern. Hierfür bietet BOOGA allerdings keine Unterstützung mehr an.

Alle Klassen sind in den Bibliotheken `libbase` (Basisdatenstrukturen und Algorithmen) und `libpixmap` (alle Abstraktionen und Operationen, die sich auf digitale Bilder beziehen) organisiert. Die meisten bedürfen keiner weiteren Beschreibung, da zum Beispiel die Aufgabe einer Stringklasse allseits bekannt sein dürfte. Auch die geometrischen Operation auf Vektoren, wie das Bilden eines Vektorproduktes oder die Verknüpfung von Transformationsmatrizen, brauchen keine weiteren Erläuterungen. Im folgenden wird deshalb lediglich auf die Modellierung digitaler Bilder (Pixmaps) eingegangen, die in BOOGA eine spezielle Behandlung erfahren.

6.2 Digitale Bilder

Die verschiedenen Teilbereiche der Computergrafik stellen sehr unterschiedliche Anforderungen an eine Abstraktion für digitale Bilder. So beschränken sich Bildverarbeitungsoperationen häufig auf die Bearbeitung von Schwarzweiss- oder Graustufenbildern. Applikationen der synthetischen Bilderzeugung arbeiten dagegen mit hochgenauen Farbmodellen und setzen eine zusätzliche Unterstützung von Alpha- und Tiefenkanälen voraus. Nun ist es nicht sinnvoll, oder sogar unmöglich, eine Abstraktion für die Darstellung von digitalen Bildern zu finden, die allen Anforderungen ohne Kompromisse gleichzeitig gerecht werden kann. In BOOGA wird diesem Konflikte daher mit einem generischen Konzept begegnet, das eine an die jeweilige Anwendung angepasste Repräsentation im Arbeitsspeicher erlaubt (Abschnitt 6.2.1), neben dem BOOGA-eigenen Format auch andere Bildformate unterstützt und Operationen auf Bildern als eigenständige Klassen im Sinne des Strategy Patterns realisiert (Abschnitt 6.2.2).

6.2.1 Die Pixmap-Hierarchie

Abbildung 6.1 zeigt die für die Unterstützung von digitalen Bildern (Pixmaps) relevanten Klassen. Es werden die Aspekte Organisation der Bildinformation im Arbeitsspeicher (Hierarchie `AbstractPixmap`) und Repräsentation der Farbinformation eines einzelnen Pixels unterschieden (die in der Abbildung grau hinterlegten *Pixeltypen*). Diese Zweiteilung ermöglicht die flexible Konstruktion von Pixmaps mittels Komposition, die so den jeweiligen Bedürfnissen einer Anwendung optimal angepasst werden können. Die Realisierung basiert auf dem C++ Template-Mechanismus, wodurch eine generische Lösung entstanden ist, die sich gleichzeitig durch gute Laufzeit- und Speichereffizienz auszeichnet.

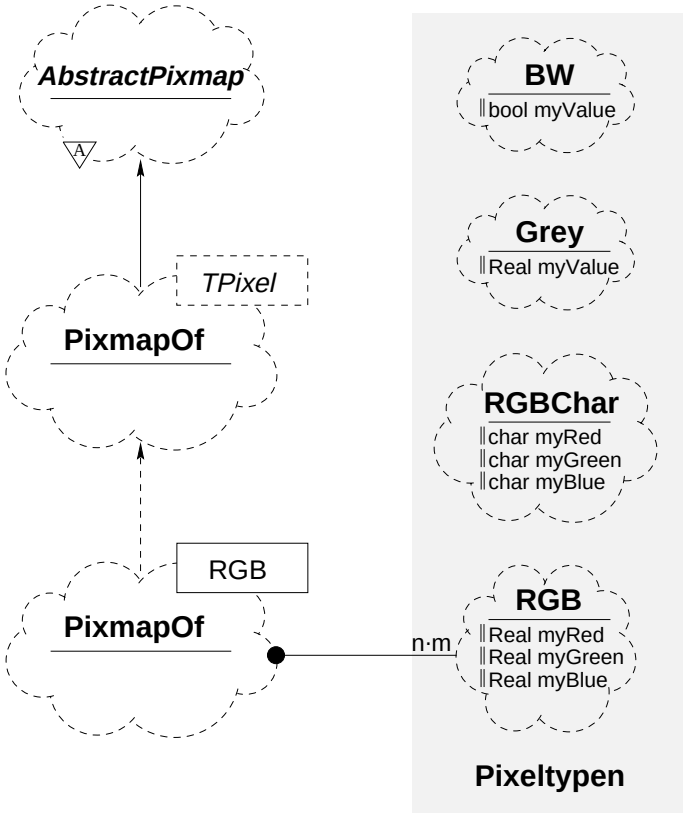


Abbildung 6.1 *Die Klassenhierarchie für die Unterstützung digitaler Bilder. Es wird zwischen Klassen für die Organisation der Bildinformation im Arbeitsspeicher und der Repräsentation der Pixelinformation selbst unterschieden.*

Die Pixmap-Abstraktion von Booga unterstützt Alpha- und Tiefenkanäle. Eine beliebige Anzahl benutzerdefinierter Kanäle lässt sich zudem nach Bedarf anlegen. Weiter braucht beim Algorithmenentwurf auch nicht berücksichtigt zu werden, welche Bildtypen bearbeitbar sind. Liegt zum Beispiel ein Farbbild vor, obwohl

nur Schwarzweissbilder erlaubt sind, kann sich die Pixmap automatisch in der gewünschten Form präsentieren. Dies wird durch einen entsprechenden Aufbau der Zugriffsmethoden in der Klasse `AbstractPixmap` erreicht, wie ausschnittsweise im folgenden Codefragment zu sehen ist:

```
class AbstractPixmap {
  ...
public:
  // Lese- oder Schreibposition in der Pixmap festlegen.
  void setPosition(int x, int y) const;

  // Farbwert eines Pixels an der durch die Methode
  // setPosition() angegebenen Position ermitteln.
  void getColor(Real& c1, Real &c2, Real& c3) const;
  void getColor(Real& val) const;
  void getColor(bool& val) const;

  // Farbwert eines Pixels an der durch die Methode
  // setPosition() angegebenen Position neu setzen.
  void setColor(Real c1, Real c2, Real c3);
  void setColor(Real val);
  void setColor(bool val);
};
```

Die verschiedenen `setColor(...)` und `getColor(...)` Methoden erlauben einen für den jeweiligen Algorithmus angepassten Zugriff auf die Pixmap. Ein Aufruf wird an den verwendeten Pixeltyp delegiert, der eine allenfalls notwendige Konvertierung der Farbinformation vornimmt. Als Beispiel zur Illustration dient ein Ausschnitt des Pixeltyps `Grey`:

```
class Grey {
public:
  void getValue(Real& c1, Real &c2, Real& c3) const;
  void getValue(Real& val) const;
  void getValue(bool& val) const;

  void setValue(Real c1, Real c2, Real c3) {
    // Umrechnung in Grauwert.
     myValue = 0.59*c1 + 0.3*c2 + 0.11*c3;
  }

  void setValue(Real val) {
    myValue = val;
  }
```

```
  void setValue(bool val) {
    if (val == false)
      myValue = 0.0;
    else
      myValue = 1.0;
  }

private:
  Real myValue;
};
```

Die beschriebenen Mechanismen lassen sich nun leicht für verschiedenartige Aufgaben ausnützen. So kann durch die Verwendung zweier Pixmaps mit unterschiedlichen Pixelrepräsentationen eine Umwandlung eines Farbbildes in eine Graustufen-Pixmap realisiert werden:

```
// Erzeugen einer RGB-Pixmap durch Einlesen der Daten
// aus der Datei 'rgb.pixi'.
AbstractPixmap* original = new PixmapOf<RGB>("rgb.pixi");

// Graustufen-Pixmap derselben Groesse erzeugen.
AbstractPixmap* kopie =
      new PixmapOf<Grey>(original->getResoultionX(),
                         original->getResolutionY());

// Zwischenspeicher fuer die Farbinformation eines
// einzelnen Pixels.
Real c1, c2, c3;

// Bildinformation vom Original zur Kopie uebertragen.
for (long p=0; p<original->getSize(); p++) {
  original->setPosition(p);
  original->getColor(c1, c2, c3);

  kopie->setPosition(p);
  kopie->setColor(c1, c2, c3); // Automatische Umrechnung!
}
```

Solche Operationen sollten allerdings als eigenständige Klassen realisiert werden, damit sie leichter in anderen Anwendungen wiederverwendet werden können. BOOGA sieht speziell hierfür einen Mechanismus vor, der im folgenden Abschnitt erläutert wird.

6.2.2 Operationen auf Pixmaps

Es existieren eine grosse Menge von Bildbearbeitungsfunktionen und auch eine enorme Vielfalt von Varianten für deren Implementie-

rung. Um eine Wiederverwendung solcher Operationen sicherzustellen, gibt BOOGA einen Implementierungsrahmen vor. Grundsätzlich wird zwischen zwei Typen unterschieden: den bild*erzeugenden* und bild*verändernden* Operationen. Abbildung 6.2 zeigt die hierfür vorgesehenen Basisklassen.

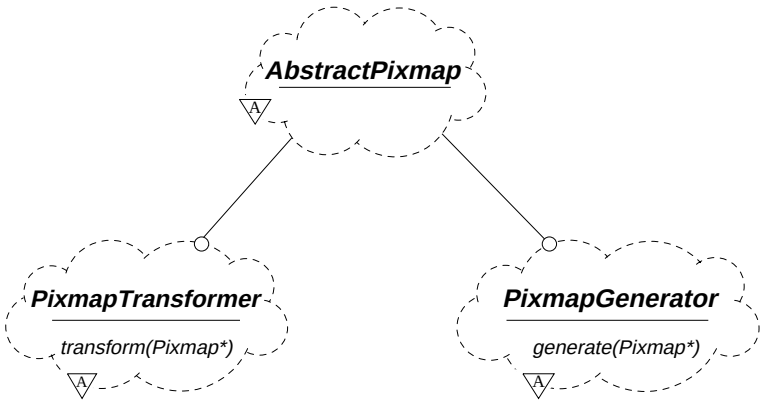


Abbildung 6.2 *Basisklassen für die beiden Typen von Pixmap-Operationen.*

Als einfaches Anwendungsbeispiel dient die Gammakorrektur eines Bildes [Cat79]. Der folgende Ausschnitt zeigt die wesentlichen Elemente der Implementation:

```
class GammaCorrection : public PixmapTransformer {
public:
  GammaCorrection(Real gamma);
  virtual void transform(AbstractPixmap* pm);
private:
  Real myGamma; // Faktor fuer die Korrektur.
};

void GammaCorrection::transform(AbstractPixmap* pm)
{
  Real r, g, b;

  for (long p=0; p<pm->getSize(); p++) {
    // Farbwert an aktueller Position ermitteln.
    pm->setPosition(p);
    pm->getColor(r, g, b);

    // Gammakorrektur durchfuehren.
    r = pow(r, 1./myGamma);
    g = pow(g, 1./myGamma);
    b = pow(b, 1./myGamma);

    // Neuer Farbwert setzen.
    pm->setColor(r, g, b);
  }
}
```

Operationen vom Typ `PixmapGenerator` besitzen einen ähnlichen Aufbau. Als erster Schritt wird eine neue Pixmap erzeugt, die dann unter Berücksichtigung der als Parameter übergebenen Pixmap bearbeitet wird. Das so erzeugte Bild dient dann als Rückgabewert der Operation.

Allgemein ist bei der Implementierung auf möglichst einfache, atomare Pixmap-Operationen zu achten. Komplexere Algorithmen lassen sich dann meist mit Hilfe dieser einfachen Operationen zusammensetzen. Es hat sich gezeigt, dass durch das Befolgen dieses Ratschlags das Wiederverwendungspotential von Bildbearbeitungsfunktionen wesentlich grösser ist.

Frameworkschicht von BOOGA

7

7.1 Überblick

Die Frameworkschicht von BOOGA stellt die Funktionalität für die flexible Modellierung eines *Szenengraphen* zur Verfügung. Zum einen enthält sie Abstraktionen für die Repräsentation geometrischer Objekte, zum anderen ist die Modellierung von *Oberflächeneigenschaften* dieser Objekte ein wichtiger Bestandteil. Alle Abstraktionen und Mechanismen sind in analoger Weise für den 2D- und 3D-Fall vorhanden. Durch den intensiven Einsatz von C++ Templates konnte aber eine Codeverdoppelung für die beiden Fälle vermieden werden.

Die Frameworkschicht kennt keine Mechanismen zur Traversierung des Szenengraphen. Auch Operationen für die Verarbeitung der geometrischen Objekte sind auf dieser Ebene nicht vorgesehen. Lediglich die Grundlagen in Form allgemeiner Dienstprotokolle sind vorhanden, die in der Komponentenschicht Verwendung finden.

Ein Leitgedanke bei der Entwicklung von BOOGA war der Grundsatz:

> *Flexibilität ist nur dann erwünscht, falls sie auch benötigt wird.*

Dies hat zu wenigen aber allgemeinen *Mechanismen*[1] geführt, die in den folgenden Ausführungen präsentiert werden.

7.2 Modellierung des Szenengraphen

Eine BOOGA-Szene ist als azyklischer, gerichteter Graph (DAG) organisiert, strukturiert mit Hilfe von Aggregatobjekten (siehe Abschnitt 4.2.2). Alle Elemente, die für eine mögliche Darstellungsstrategie benötigt werden sind als eigene Abstraktionen im Graphen enthalten. Unter anderem sind dies Objekte für die Repräsentation von Lichtquellen und virtuellen Kameras. Attribute, wie geometrische Transformationen oder Texturen, sind nicht als eigenständige Knoten im Szenengraph enthalten, sondern sind Bestandteile der geometrischen Objekte.

Mit einigen wenigen Ausnahmen existieren die gleichen Abstraktionen und Mechanismen für den 2D- und 3D-Fall. Die Dualität führt aber durch den Einsatz von C++ Templates nicht zu einer Codeverdoppelung. Diese Implementierunsvariante hat allerdings den Nachteil, dass die dadurch resultierenden Klassenhierarchien etwas aufwendiger ausfallen und entsprechend schwerer zu verstehen sind.

Die folgenden Abschnitte führen nun die Abstraktionen für die Modellierung des Szenengraphen ein und zeigen die wichtigsten Mechanismen detailliert auf. Die Ausführungen orientieren sich dabei am 3D-Fall. Dieselben Konzepte gelten aber auch für den 2D-Fall. Bei allfälligen Abweichungen wird im Text speziell darauf hingewiesen.

7.2.1 Basisabstraktionen

Abbildung 7.1 zeigt die Basisklassen der Objekthierarchie in einer Übersicht. Die Klassen `Object2D` und `Object3D` verankern die Abstraktionen für den 2D- und 3D-Fall. Beide werden aus der Template Klasse `BOOGAObject` instanziert, die ihrerseits von `Makeable`

[1] Ein Mechanismus beschreibt die Lösung von Teilaufgaben auf der Abstraktionsebene von Methoden. Die jeweilige Lösung kann durch das Ausführen einer einzelnen oder das komplexe Zusammenspiel mehrerer Methoden erfolgen.

abgeleitet ist. Die strenge Aufteilung in 2D- und 3D-Objekte verunmöglicht eine Mischung im Szenengraph, was ja auch dem Konzept von BOOGA entspricht.

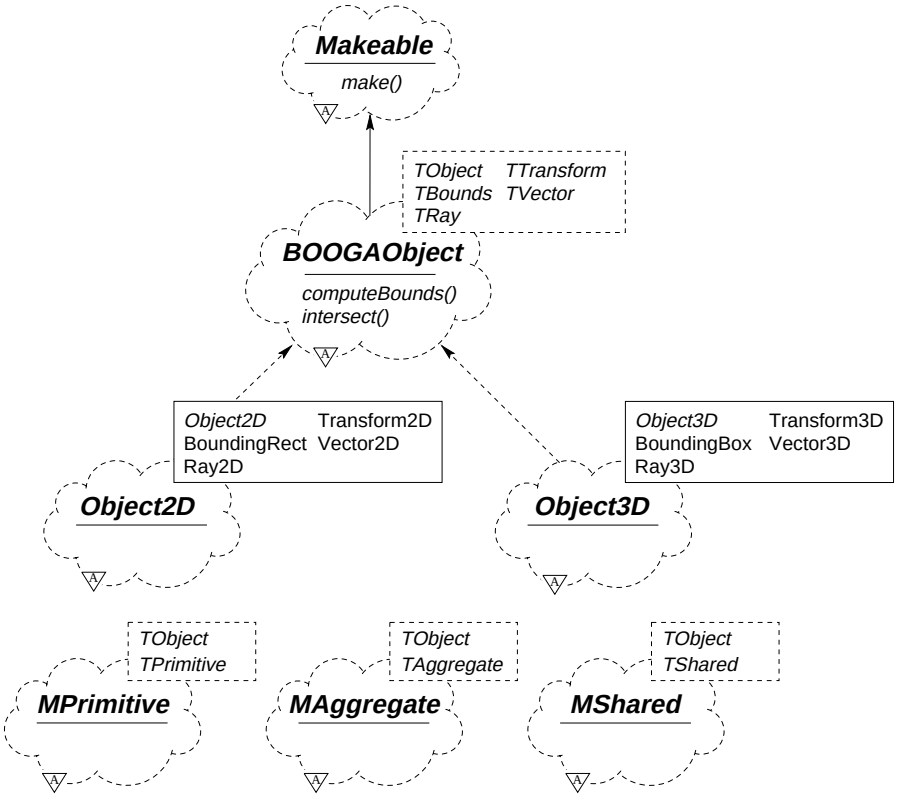


Abbildung 7.1 *Basisklassen der Objekthierarchie.*

Attribute

Die abstrakte Klasse `BOOGAObject` enthält die Deklarationen für *geometrische Transformationen* und *Bounding Volumes*, die für die verschiedensten Aufgaben Verwendung finden:

Transformationen

Für jedes Objekt im Szenengraph kann eine lokale geometrische Transformation spezifiziert werden. Verschachtelte Strukturen (siehe Abschnitt 7.2.3) können auf allen Ebenen Transformationen besitzen, die *nicht* kumuliert werden. Für die Bestimmung der kumulierten Transformation werden Pfadobjekte verwendet (siehe Abschnitt 7.3).

Bounding Volumes

Jedes Objekt besitzt einen Umschliessungskörper (Bounding Volume). Grundsätzlich sind hierfür verschiedene Formen denkbar.

Die einzige Voraussetzung ist die Unterstützung einer In-/Out-Klassifikation und der Schnitt mit einem Strahl. In BOOGA wird von dieser Allgemeinheit zur Zeit nicht Gebrauch gemacht und lediglich achsenparallele Quader bzw. Rechtecke eingesetzt. Die lokalen geometrischen Transformationen sind auf die Umschliessungkörper bereits angewendet.

Mechanismen

`make()`

Die `make()`-Methode erlaubt die dynamische Erzeugung neuer Objekte unabhängig von Konstruktoraufrufen. Dieser Mechanismus ist auch unter dem Idiom[2] *Virtual Constructor* [Cop92] bekannt. In BOOGA wird die `make()`-Methode unter anderem in erweiterbaren Editoren eingesetzt, um abhängig von Benutzereingaben neue Objekte zu erzeugen.

Die Argumente der `make()`-Methode sind polymorph und können sich aus Zeichenketten, Zahlen, Vektoren und Matrizen zusammensetzen. Als Beispiel sei hier die Implementation für die Klasse `Sphere3D` präsentiert. Sie repräsentiert eine Kugel, definiert durch Zentrum und Radius.

```
Makeable* Sphere3D::make(List<Value*>* parameters) const
{
  // Konversion der polymorphen 'Values' in die
  // gewuenschten Typen.
  Real     radius;
  Vector3D center;

  if (parameters->item(0)->toReal(radius) &&
      parameters->item(1)->toVector3D(center)) {
    // Parameter sind korrekt, neue Kugel erzeugen.
    return new Sphere3D(radius, center);
  }
  else {
    // Erzeugung wegen fehlerhaften Parameter abgebrochen.
    return NULL;
  }
}
```

`computeBounds()`

Mit der `computeBounds()`-Methode werden die Bounding Volumes eines Szenengraphen neu berechnet. Diese Operation wird etwa in einem Editor benötigt, der durch das Hinzufügen oder Entfernen

[2] Anstelle von Idiom wird auch häufig der Begriff *Coding Pattern* verwendet. Grundsätzlich wird damit ein Muster bezeichnet, das eine Lösung auf der Abstraktionsstufe einer Programmiersprache beschreibt.

von Objekten den Szenengraph verändert, wodurch die aktuellen Werte der Bounding Volumes ungültig werden.

Die Operation besitzt einen konstanten und einen variablen Anteil, repräsentiert durch zwei Methoden. Der konstante Anteil wird in der `computeBounds()`-Methode abgehandelt, während der variable Teil, der für jedes Objekte individuell implementiert werden muss, in der `doComputeBounds()`-Methode enthalten ist. Diese Implementierungstechnik entspricht dem Design Pattern *Template Method* [GHJV95, Seite 325]. In diesem Zusammenhang wird im obigen Beispiel `computeBounds()` als *Template Method* und `doComputeBounds()` als *Hook Method* bezeichnet. Der folgende C++ Code zeigt den Aufbau des konstanten Anteils der Operation, der in der Klasse `BOOGAObject` abgelegt ist:

```
template <...>
BOOGAObject::computeBounds()
{
  // Aktuelle Werte zuruecksetzen.
  myBounds.reset();

  // Rekursiv alle untergeordneten Objekte besuchen.
  for (alle untergeordnete Elemente o)
    o->computeBounds();

  // Hook-Methode aufrufen. Hier wird die eigentliche
  // Berechnung durchgefuehrt.
  doComputeBounds();

  // Lokale geometrische Transformation beruecksichtigen.
  myBounds.transform(myTransform);
}
```

Die `doComputeBounds()`-Methode für die Klasse `Sphere3D` bestimmt einen die Kugel umschliessenden Würfel:

```
void Sphere3D::doComputeBounds()
{
  // Extrempunkte berechnen und dem Bounding Volume
  // hinzufuegen.
  // Innerhalb von doComputeBounds() werden die lokalen
  // geometrischen Transformationen nicht beruecksichtigt.
  myBounds.expand(myCenter.x() - myRadius,
                  myCenter.y() - myRadius,
                  myCenter.z() - myRadius);
  myBounds.expand(myCenter.x() + myRadius,
                  myCenter.y() + myRadius,
                  myCenter.z() + myRadius);
}
```

`intersect()`

Der dritte Mechanismus erlaubt das Schneiden eines einzelnen Objektes oder ganzen Szenengraphen mit einem Strahl. Diese Operation wird beispielsweise beim Raytracing-Verfahren oder der Erkennung von Kollisionen in Virtual-Reality-Systemen eingesetzt.

Wie bei der Berechnung der Bounding Volumes, besitzt der `intersect`-Mechanismus einen konstanten und variablen Anteil. In diesem Fall durch die beiden Methoden `intersect()` und `doIntersect()` realisiert. `intersect()` ist dafür zuständig, Transformationen und Bounding Volumes zu berücksichtigen:

```
template <...>
bool BOOGAObject::intersect(TRay& ray)
{
  bool hit = false;

  // Wird das Bounding Volume getroffen?
  if (BoundsUtilities::intersect(myBounds, ray)) {

    // Lokale Transformation auf den Strahl anwenden.
    TRay transformedRay(ray);
    transformedRay.transform(myTransform):

    // Hook-Methode aufrufen. Hier wird die eigentliche
    // Schnittberechnung durchgefuehrt.
    hit = doIntersect(transformedRay);
  }

  return hit;
}
```

Die `doIntersect()`-Methode berechnet den eigentlichen Schnittpunkt mit dem Strahl (am Beispiel der Klasse `Sphere3D` ausgeführt):

```
bool Sphere3D::doIntersect(Ray3D& ray)
{
  // Ursprung des Strahls transformieren.
  Real xray = myCenter.x() - ray.getOrigin().x();
  Real yray = myCenter.y() - ray.getOrigin().y();
  Real zray = myCenter.z() - ray.getOrigin().z();

  // Quadratische Gleichung loesen.
  Real b = xray*ray.getDirection().x() +
           yray*ray.getDirection().y() +
           zray*ray.getDirection().z();
  Real discrim = sqrt(b*b - xray*xray
                          - yray*yray
                          - zray*zray
                          + myRadius*myRadius);
```

```
    // Potentieller Schnittpunkt berechnen.
    Real t = b - discrim;
    if (t < ray.getBestHitDistance()) {
      // Die Kugel wurde getroffen.
      ray.setBestHitObject(this);
      ray.setBestHitDistance(t);
      return true;
    }

    // Kein Schnittpunkt.
    return false;
  }
```

Die folgenden Abschnitte zeigen nun die Realisierung konkreter Objekttypen auf. Sie sind alle von `Object2D` bzw. `Object3D` abgeleitet und müssen deshalb konkrete Implementierungen der oben beschriebenen Methoden bereitstellen.

7.2.2 Geometrische Primitive

Primitivelemente sind auf der untersten Ebene des Szenengraphen anzutreffen und repräsentieren konkrete geometrische Objekte. Abbildung 7.2 zeigt die Konstruktion der Klasse `Primitive3D`. Zusätzlich sind einige Beispiele von Primitivelementen abgebildet. Codeverdoppelungen für den 2D- und 3D-Fall konnte wiederum durch den Einsatz von Templates vermieden werden.

Neben den im vorangegangenen Abschnitt beschriebenen Methoden `make()`, `doComputeBounds()` und `doIntersect()` implementieren Primitivobjekte zwei zusätzliche Methoden, die sie von anderen Objekttypen unterscheiden. Dies ist die Berechnung eines Normalenvektors für einen beliebigen Oberflächenpunkt und die Bereitstellung einer alternativen geometrischen Repräsentation, wie die folgenden Ausführungen näher erläutern.

Mechanismen

`normal()`

Zu jedem Punkt seiner Oberfläche muss ein Primitivobjekt den zugehörigen Normalenvektor berechnen können. Diese Funktionalität wird durch die Methode `normal()` aktiviert. Für die Klasse `Sphere3D` sieht die konkrete Implementation wie folgt aus:

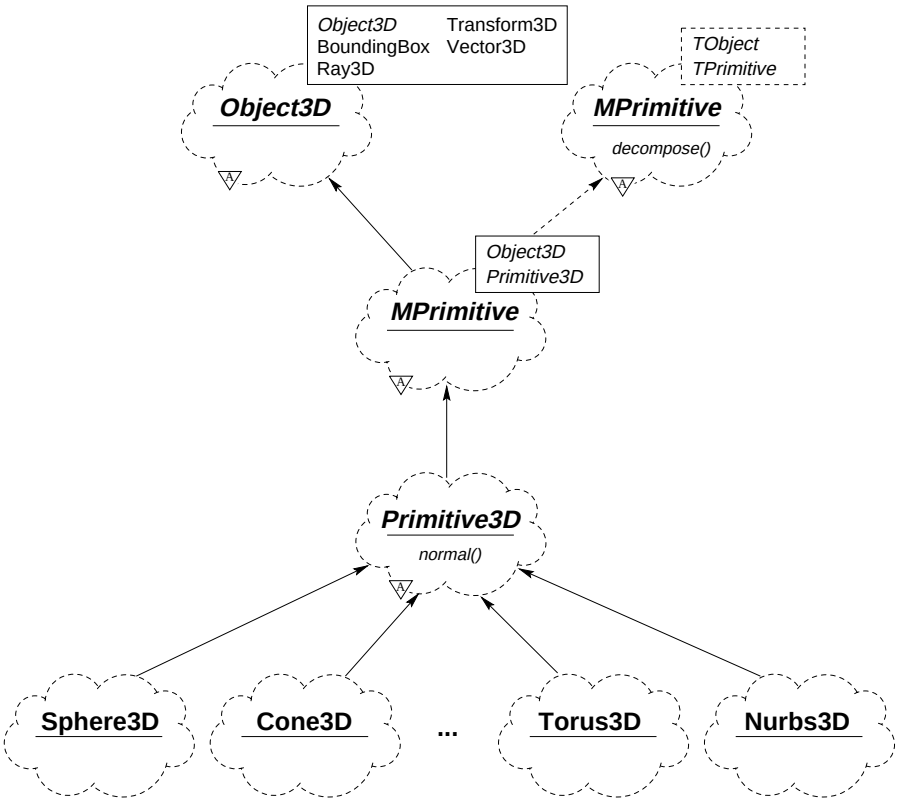


Abbildung 7.2
Die `Primitive3D`-Hierarchie dient als Basis für die Konstruktion geometrischer Objekte.

```
Vector3D Sphere3D::normal(const Vector3D& point)
{
  // Die Normale ist die Verbindung vom Kugelzentrum zum
  // angegebenen Punkt.
  // Eine anschliessende Normalisierungsoperation begrenzt
  // die Laenge des ermittelten Vektors auf 1.
  return (point-myCenter).normalized();
}
```

Normalenberechnungen werden im lokalen Koordinatensystem des jeweiligen Objektes durchgeführt. Lokale oder globale geometrischen Transformationen sind nicht berücksichtigt.

Normalenvektoren können ausschliesslich für 3D-Primitive berechnet werden. Im 2D-Fall macht dies wenig Sinn. Daher wird die `normal()`-Methode in der Klasse `Primitive3D` deklariert und nicht bereits in der Template-Klasse `MPrimitive`, die ja für den 2D- und 3D-Fall Verwendung findet.

`decompose()`

Viele Algorithmen können nur mit ganz bestimmten Objekttypen umgehen. So beschränkt man sich bei der Implementierung des Z-Buffer-Verfahrens oft auf Dreiecke. Andere Objekttypen können in eine Menge von Dreiecken zerlegt werden, ohne allzu grosse Qualitätseinbussen hinnehmen zu müssen. BOOGA stellt mit der `decompose()`-Methode einen entsprechenden Mechanismus zur Verfügung. Jedes Primitivobjekt erzeugt beim Aufruf dieser Methode eine alternative Repräsentation, bestehend aus einer beliebigen Anzahl von Objekten. Dabei muss nicht notwendigerweise eine Triangulation durchgeführt werden, sondern die alternative Repräsentation kann aus beliebigen Objekten unterschiedlichen Typs zusammengesetzt sein. Ein `Text3D`-Objekt, das für die Darstellung von Text im 3D-Raum verwendet wird, liefert beispielsweise eine Menge von Zylindern und Kugeln als alternative Repräsentation. Die Zylinder repräsentieren die Strecken der einzelnen Buchstaben, während Kugeln Verbindungen und Abschlüsse bilden. Die weiter oben erwähnte Z-Buffer-Implementation könnte mit diesen Objekten allerdings immer noch nichts anfangen und würde deshalb die `decompose()`-Methode wiederholt anwenden. Dies wird solange fortgesetzt bis eine homogene Menge von Dreiecken entstanden ist. Kann ein einzelnes Objekt allerdings nicht in Dreiecke konvertiert werden, so wird es bei der Darstellung ignoriert.

7.2.3 Aggregate

Aggregatstrukturen sind, wie in Abschnitt 4.2.2 vorgeschlagen, als eigenständige Elemente in die Objekthierarchie integriert. Sie können dadurch als Knoten in einem Szenengraphen auftreten und sind massgeblich an der Strukturierung der Szene beteiligt. Abbildung 7.3 zeigt den für die Aggregatstrukturen relevanten Teil der Objekthierarchie von BOOGA.

Die Implementation folgt dem Design Pattern *Composite* [GHJV95, Seite 163] und erlaubt eine beliebige Verschachtelung von Aggregatstrukturen.

Neben der Möglichkeit zur Gruppierung von Objekten zu zusammengehörenden Einheiten, ergibt sich durch den Einsatz von Aggregaten auch eine Beschleunigung des `intersect()`-Mechanismus. Dieser Effekt tritt durch die Verschachtelung von Bounding Volumes ein [KK86]. Es existieren aber auch spezielle Aggregattypen, die eine sehr effiziente Implementierung der `intersect()`-Methode

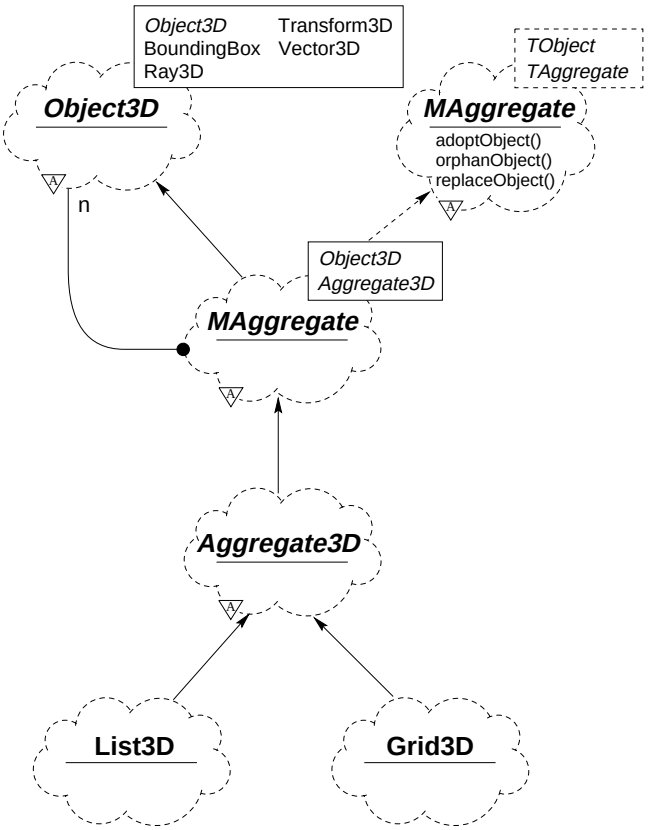


Abbildung 7.3 *Konstruktion der Aggregattypen für den 3D-Fall.*

erlauben [Col90]. Mit `Grid3D` enthält BOOGA bereits ein Aggregat dieses Typs [AW87], das eine reguläre Unterteilung des Raumes in Voxels vornimmt und die Objekte gemäss ihren Positionen den entsprechenden Zellen zuordnet. Bei einer Schnittberechnung werden nur noch die vom Strahl getroffenen Zellen besucht und die allenfalls darin enthaltenen Objekte getestet. Details zur Implementation können in [Ama93] nachgelesen werden.

Mechanismen

`adoptObject()`, `orphanObject()`, `replaceObject()`

In der Template-Klasse `MAggregate` werden Methoden für das Einfügen (`adoptObject()`), das Entfernen (`orphanObject()`) und das Ersetzen (`replaceObject()`) eines Objektes deklariert. Alle Aggregattypen implementieren diese Methoden, die sowohl im 2D- wie im 3D-Fall zur Verfügung stehen.

7.2.4 Mehrfachreferenzen

Mit den bisher beschriebenen Objekttypen lassen sich Szenen mit Baumstrukturen aufbauen. In Abbildung 7.4 sind diejenigen Klassen aufgeführt, die den Aufbau eines Szenen*graphen* ermöglichen.

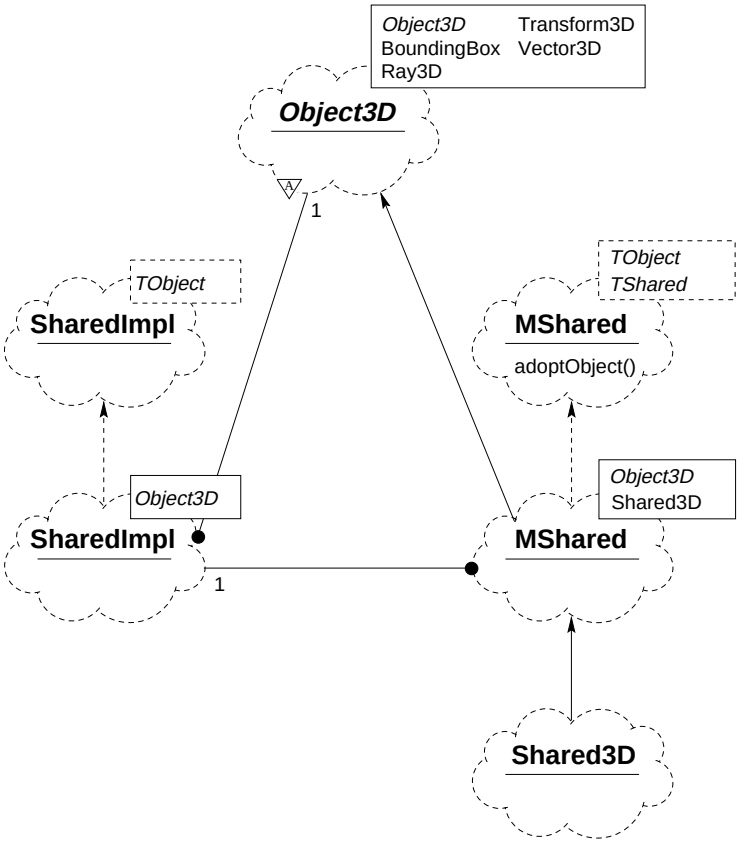


Abbildung 7.4 *Aufbau der Klassen für die Modellierung von Mehrfachreferenzen in einem Szenengraph.*

Der Typ `Shared3D` erlauben den mehrfachen Einsatz von Objekten in einer Szene. Die Implementation bedient sich dabei dem *Letter Envelope* Idiom [Cop92]. Dieses teilt eine Abstraktion in zwei Klassen auf, in eine Hülle (Envelope) mit einem Verweis auf einen Inhalt (Letter). Mehrere Hüllen können nun auf denselben Inhalt verweisen, wodurch der Inhalt logisch mehrfach vorhanden ist.[3] In unserem Anwendungsfall ist der Inhalt ein Objekt, welches mehrfach im Szenengraphen verwendet werden soll. In Abbildung 7.5 ist die einfache Beispielszene aus Abschnitt 4.2.2 unter Verwendung von `Shared3D`-Objekten aufgebaut.

[3] Die Namensgebung ist in diesem Zusammenhang leider etwas unglücklich. Mehrfachreferenzen bilden allerdings nur einen Teilaspekt des Letter Envelope Idioms.

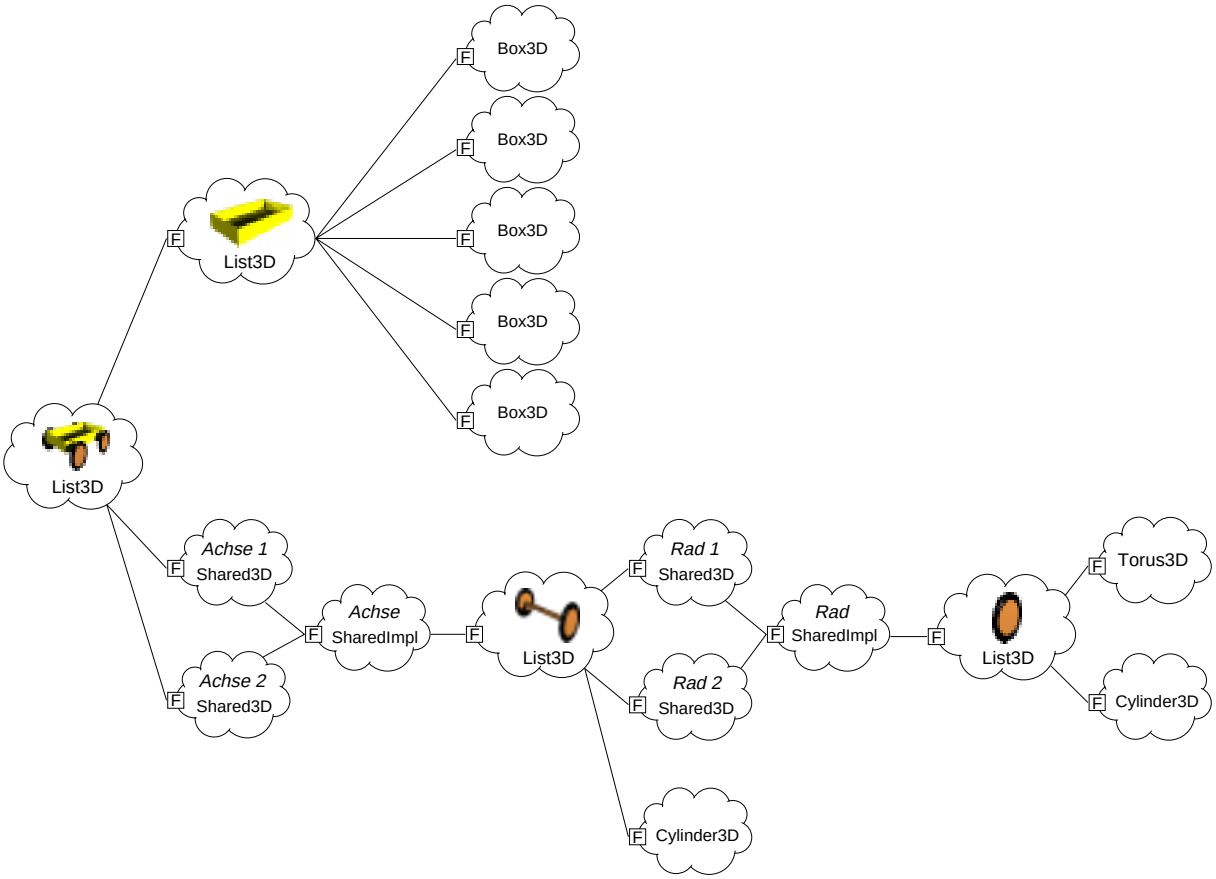


Abbildung 7.5
Ein Szenengraph mit Mehrfachreferenzen. Obwohl das Modell beispielsweise logisch vier Räder besitzt, ist physisch lediglich eine Instanz im Arbeitsspeicher abgelegt.

Der Vorteil von Mehrfachreferenzen liegt vor allem in der erreichbaren Speicherersparnis. Erst mit dieser Technik können beispielsweise ganze Landschaften mit Tausenden von Bäumen im Arbeitsspeicher abgelegt werden. Nachteilig wirkt sich aus, dass eine mehrfach referenzierte Objektinstanz nicht mehr eindeutig ist. Der Identitätsbegriff für Objekte muss deshalb als Pfad von der Wurzel des Szenengraphen bis zum jeweiligen Objekt definiert werden. In BOOGA existiert hierzu eine spezialisiert Pfad-Klasse, die in Abschnitt 7.3 beschrieben wird.

Mechanismen

Lediglich eine zusätzliche Methode ist für die Verwaltung von `Shared3D` Objekten notwendig. Mittels `adoptObject()` kann der Inhalt neu gesetzt werden. Eine allenfalls bestehende Belegung wird gelöscht.

`adoptObject()`

7.2.5 Virtuelle Kamera

Das Konzept der virtuellen Kamera ist grundlegend für die verschiedensten Darstellungsverfahren. In BOOGA ist dafür die Klasse `Camera3D` zuständig. Sie ist direkt von der Basisklasse `Object3D` abgeleitet, wie in der Abbildung 7.6 zu sehen ist.

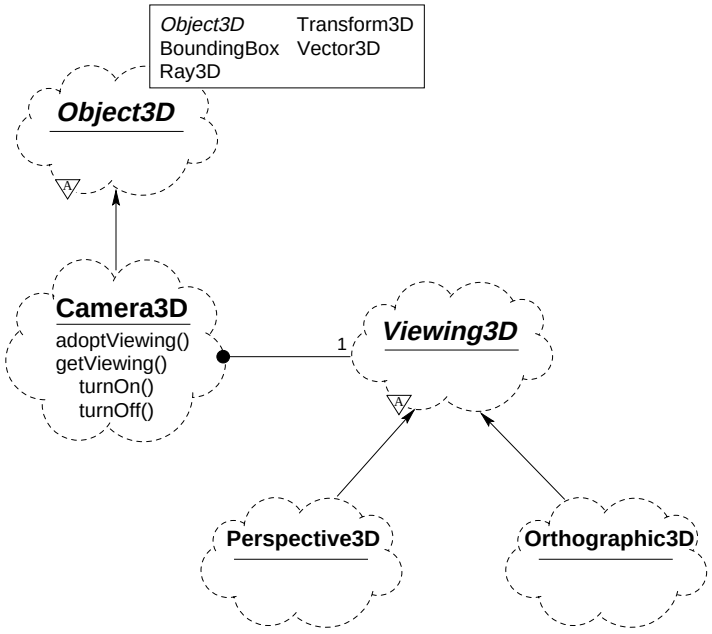


Abbildung 7.6 *Virtuelle Kameras werden als eigenständige Objekte in den Szenengraph integriert.*

Eine beliebige Anzahl von Kameraobjekten kann im Szenengraph enthalten sein. Diese sind vollwertige Objekte, besitzen aber keine geometrische Repräsentation. Daher wird das Bounding Volume immer eine minimale Grösse haben und ein Aufruf der `intersect()`-Methode liefert niemals einen Schnittpunkt.

Die `Camera3D` Abstraktion ist eigentlich nur eine Hülle, ein

Wrapper[4], für eine Projektionsstrategie. Im 3D-Fall existieren deren zwei, eine perspektivische (`Perspective3D`) und eine orthographische (`Orthographic3D`) Variante. Beliebig andere Strategien sind denkbar, wie zum Beispiel eine Panorama- oder Fish-Eye-Projektion. Im 2D-Fall wird dasselbe Konzept verwendet. Die Projektionsstrategien folgen aber einen entsprechend angepassten Modell.

Darstellungsverfahren, die die Kamerainformation für die Bildgenerierung verwenden, müssen den Szenengraph in einem ersten Schritt nach den `Camera3D`-Objekten durchsuchen. Dabei werden möglicherweise mehrere Kameradefinitionen gefunden, aus denen ausgewählt werden kann.

Mechanismen

`getViewing()`, `adoptViewing()`, `turnOn()`, `turnOff()`

Die eigentliche Funktionalität ist innerhalb der von `Viewing3D` abgeleiteten Klassen enthalten. Diese erlauben die Umwandlung von Welt- nach Kamera- oder Bildschirmkoordinaten.

Die Klasse `Camera3D` stellt lediglich Methoden für die Handhabung der Projektionsstrategie zur Verfügung (`getViewing()` und `adoptViewing()`) und erlaubt ein Ein- und Ausschalten der Kamera (`turnOn()` bzw. `turnOff()`).

7.2.6 Lichtquellen

Ähnlich wie Kameras, werden auch Lichtquellen als eigenständige Objekte in einen BOOGA-Szenengraph integriert (siehe Abbildung 7.7).

Grundsätzlich werden zwei Typen unterschieden: *Gerichtete Lichtquellen* sind von der Klasse `DirectedLight` abgeleitet, *ambiente Lichtquellen* sind Instanzen der Klasse `AmbientLight`. Alle Lichtquellen besitzen eine *Leuchtstärke* (Luminance) und eine *Farbe* (Color). Miteinander multipliziert fliessen diese Parameter in die unterschiedlichen Beleuchtungsgleichungen ein (siehe Abschnitt 7.4).

[4] Ein Wrapper stattet eine bestehende Klasse durch Einkapselung mit einem neuen Interface aus. Methodenaufrufe für ein solches Wrapper-Objekt werden nun an das gekapselte Objekt delegiert. Diese Technik erlaubt die Verwendung existierender Abstraktionen in Anwendungen, die zum Beispiel andere Methodennamen voraussetzten.

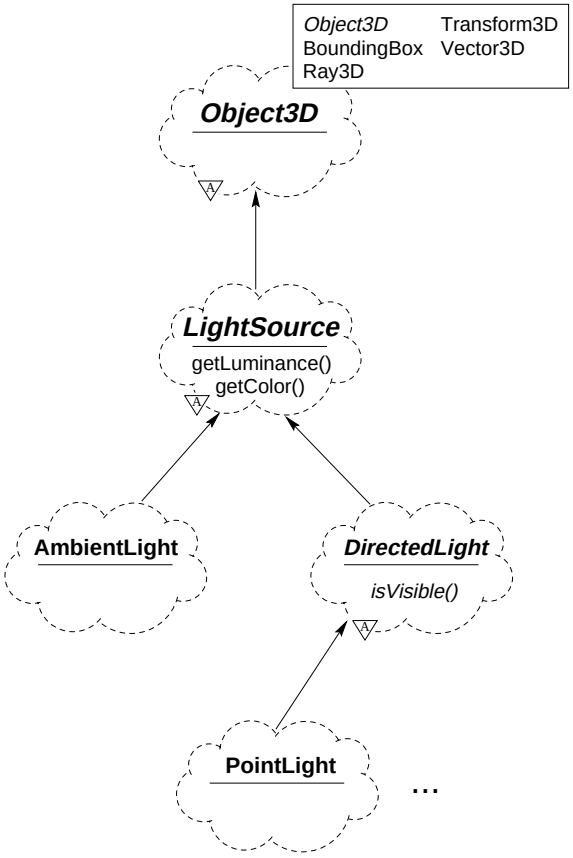


Abbildung 7.7
Die Hierarchie der Lichtquellen.

Während im ambienten Fall die gesamte Szene uniform aufgehellt wird, besitzen gerichtete Lichtquellen wesentlich mehr Möglichkeiten, das Resultat zu beeinflussen. Im Zusammenspiel mit Beleuchtungsgleichungen werden durch sie beispielsweise Glanzlichter oder Schattenwürfe erzeugt. Dazu sind allerdings einige zusätzliche Mechanismen notwendig, wie weiter unten ausgeführt wird.

Im Unterschied zu Kameraobjeken können Lichtquellen geometrische Repräsentationen besitzen. So wird etwa eine Punktlichtquelle (`PointLight`) durch eine Kugel dargestellt, deren Farbe abhängig von den Parametern der Lichtquelle gewählt wird.

Im 2D-Fall gibt es kein zu den Lichtquellen vergleichbares Konzept, da hier entsprechende Beleuchtungsmodelle fehlen. Dies will natürlich nicht heissen, dass eine Integration entsprechender Abstraktion nicht möglich wäre. Das zugrundeliegende Konzept müsste allerdings erst entwickelt werden.

Mechanismen

`isVisible()`, `getDirection()`

In Beleuchtungsgleichungen fliessen diverse Faktoren ein. Unter anderem sind dies die Position und Ausrichtung, die Farbe und Leuchtkraft sowie die Sichtbarkeit einer Lichtquelle. BOOGA stellt für die Gewinnung dieser Informationen diverse Methoden bereit. Während die meisten Informationen durch einfache Parameterabfragen realisiert werden können, stellt die Beantwortung der Sichtbarkeitsfrage wesentlich höhere Anforderungen an den zugrundeliegenden Algorithmus. Die virtuelle Methode `isVisble()` ermöglicht jeder Lichtquelle eine individuelle Implementierung. Die Implementation der Methode für die Punktlichtquelle (`PointLight`) ist wie folgt aufgebaut:

```
Real PointLight::isVisible(Object3D* scene,
                           Vector3D& objectPos)
{
  // Ein Strahl wird in Richtung des anfragenden Objektes
  // ausgeschickt (an der Position 'objectPos'). Liegt der
  // Schnittpunkt nicht bei diesem Objekt, so ist die
  // Lichtquelle nicht sichtbar.
  Ray3D ray(myPosition, objectPos-myPosition);

  if (scene->intersect(ray)) {
    // Strahl hat ein Objekt getroffen. Ist nun der
    // Schnittpunkt mit der uebergebenen Position
    // identisch?
    if (ray->getHitPoint() == objectPos))
      // Ja, die Lichquelle ist sichtbar.
      return 1.0;
  }

  // Lichtquelle ist fuer das Objekt nicht sichtbar.
  return 0.0;
}
```

Im vorangegangenen Beispiel wird die `intersect()`-Methode verwendet, um die Sichtbarkeitsfrage zu beantworten. Der Rückgabewert der Methode `isVisible()` ist eine Gleitkommazahl im Bereich $[0, 1]$. Innerhalb einer Beleuchtungsgleichung wird der Rückgabewert als *Grad der Sichtbarkeit* interpretiert. Dies erlaubt die einfache Integration von flächigen Lichtquellen (beispielsweise einer Neonröhre), die diffuse Schatten werfen können. Der eigentliche Beleuchtungsanteil, der in die Berechnungen einfliesst, errechnet sich nach der einfachen Formel

```
light.getColor()*light.getLuminance()*light.isVisible(...)
```

unter Berücksichtigung der Position und Ausrichtung der Lichtquelle.

Hiermit ist die Besprechung der Abstraktionen und Mechanismen für die Modellierung des Szenengraphen abgeschlossen. Die nächsten Abschnitte führen weitere Elemente der Frameworkschicht ein, die für viele Applikationen wichtig sind.

7.3 Pfadobjekte

BOOGA-Szenen sind als azyklische, gerichtete Graphen organisiert. Die verschiedenen Objekttypen repräsentieren die Knoten. Ein Objekt ist nun erst zusammen mit seiner Lage innerhalb der Szene (des Graphen) eindeutig identifiziert. Pfadobjekte werden in BOOGA daher zur Lagebeschreibung verwendet, die ausgehend von der Wurzel alle Knoten auf dem Weg bis zum betrachteten Objekt enthalten. Die Abbildung 7.8 zeigt ein konkretes Beispiel auf.

Instanzen der Klassen `Path2D` bzw. `Path3D` repräsentieren Pfade in BOOGA. Als Nebeneffekt berechnet und speichert ein Pfadobjekt die kumulierte geometrische Transformation für das durch den Pfad beschriebene Objekt. Zwei Objektreferenzen bezeichnen zudem nur dann dasselbe Objekte, falls auch die zugehörigen Pfade identisch sind.

Mechanismen

Pfadobjekte sind einfache Datenstrukturen, die weitestgehend einem Stack ähneln. Es ist möglich, Objekte dem Pfad hinzuzufügen (`append()`-Methode) und wieder zu entfernen (Methode `removeLast()`). Weiter ist eine Iteration über alle Objekte im Pfad möglich (Methoden `first()`, `next()` und `isDone()`). Kumulierte Transformationsmatrizen werden beim Hinzufügen von Objekten automatisch berechnet.

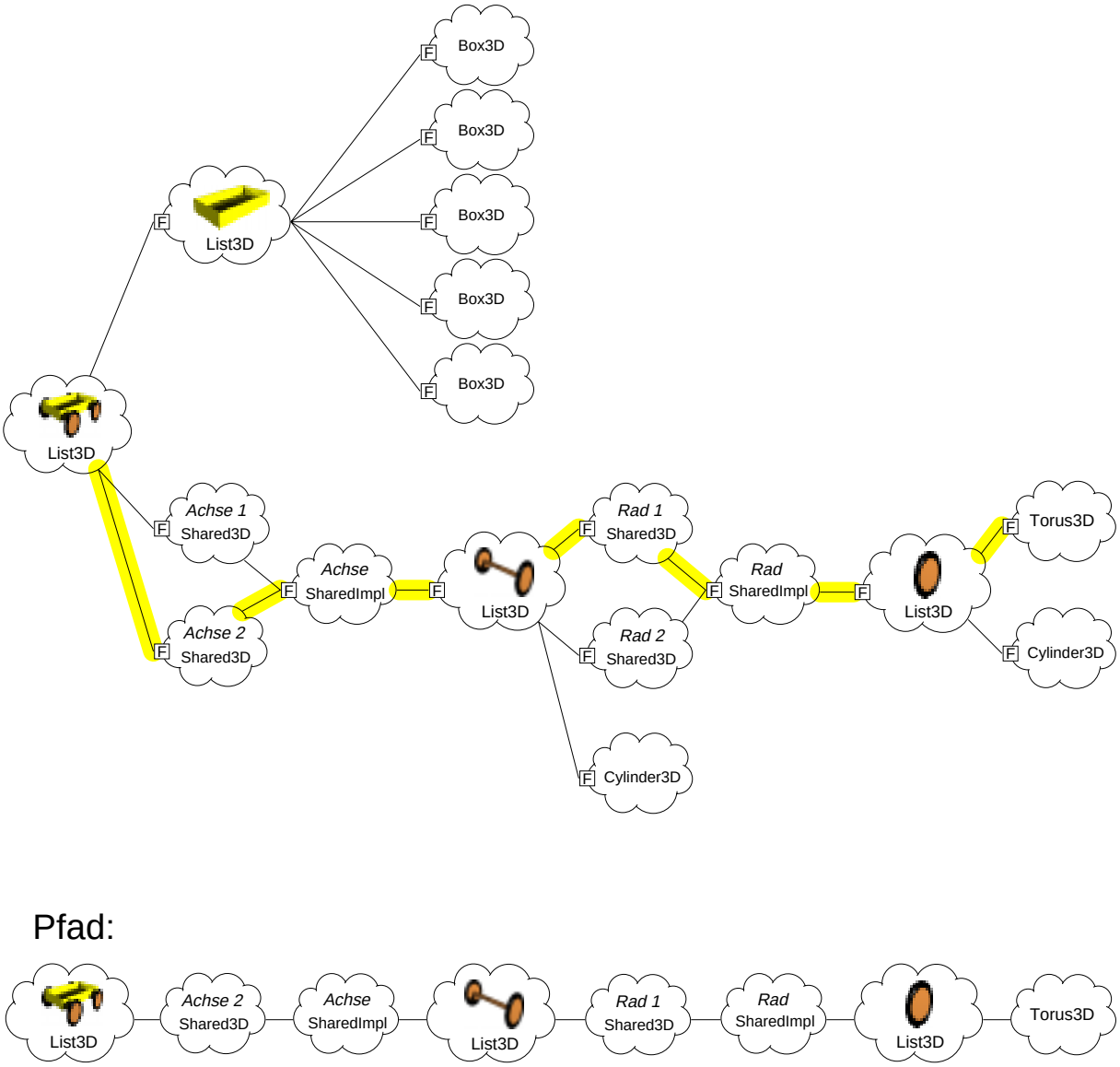


Abbildung 7.8

Alle Objekte eines Szenengraphen werden durch einen Pfad eindeutig identifiziert.

7.4 Texturen

In herkömmlichen Grafiksystemen wird zwischen den Begriffen *Texturing* und *Shading* unterschieden [FvDFH90]. Texturing ist eine Methode, die Oberflächeneigenschaften eines geometrischen Objektes zu beschreiben. Dieser Prozess erlaubt beispielsweise die Variation der geometrischen Struktur oder die Zuordnung von individuellen Farbwerten zu einzelnen Punkten der Oberfläche. Shading ist die Berechnung einer Pixelfarbe aufgrund eines Beleuchtungsmodells und unter Berücksichtigung der Oberflächeneigenschaften. Damit diese Arbeitsteilung erfolgreich sein kann, müssen gemeinsame Attribute vorliegen. Nur so kann eine Textur die Oberflächeneigenschaften manipulieren und dadurch Einfluss auf den Shading-Prozess nehmen. Vor allem in einfachen Grafiksystemen sind diese Bedingungen erfüllt. Sie bieten meist ein einzelnes Shading-Verfahren an, das durch eine kleine, erweiterbare Menge von Texturen in seiner Arbeitsweise beeinflusst werden kann. Das Verfahren ist zudem meist stark mit der Technik zur Berechnung von verdeckten Oberflächen verknüpft.

Komplexere Grafiksysteme verschmelzen das Textur- mit dem Shadingkonzept und ermöglichen so eine wesentlich flexiblere Simulation der Interaktion von Licht und Materie. Auf diese Weise werden Abhängigkeiten zum eigentlichen Darstellungsverfahren (Rendering-Verfahren) minimiert. Mehrere Arbeiten schlagen vor, die Materie-Licht-Interaktion als eigenständige Abstraktion zu beschreiben [KA88, HL90]. RenderMan [Ups90] erlaubt zum Beispiel die detaillierte Formulierung von Oberflächeneigenschaften und Beleuchtungsgleichungen mittels einer speziell dafür konzipierten Sprache. Mit dieser *Shading Language* lassen sich unabhängig vom darunterliegenden Rendering-System sogar die geometrischen Strukturen von Objekten verändern und atmosphärische Effekte und die Eigenschaften von Lichtquellen funktional beschreiben. Das Modell von RenderMan wird auch in neueren Arbeiten aufgegriffen und weiter verfeinert. So präsentierte Shirley [SSB91] ein auf dieser Grundlage aufgebautes Raytracing-Framework, das auch die Modellierung der Radiosity-Technik [GTGB84] erlaubt.

Eine konsequente Weiterführung dieses Konzeptes wurde in Booga realisiert. Eine Textur kapselt alle Attribute und Mechanismen ein, die für die Berechnung der Farbintensität einzelner Pixel notwendig sind. Im Darstellungsprozess werden die geometrischen Objekte

einer Szene lediglich für die Beantwortung der Sichtbarkeitsfrage (*Hidden Surface Detection* oder *HSD*) herangezogen. Danach bestimmen Texturen alle weiteren Effekte, wie Schattenwurf oder Reflexionen. Ein Darstellungsverfahren setzt sich in BOOGA also aus den beiden Elementen *Beantwortung der Sichtbarkeitsfrage* (Hidden Surface Detection, HSD) und *Evaluierung von Texturen* zusammen oder kurz

$$Rendering = HSD + Texturing.$$

Viele traditionelle Darstellungsverfahren existieren in BOOGA daher nicht mehr in ihrer ursprünglichen Form. Zum Beispiel degenerieren die komplexen Mechanismen eines Raytracers zu einem einfachen Aussenden von Strahlen, um die Sichtbarkeitsfrage zu beantworten. In einem zweiten Schritt werden nun für jeden besuchten Oberflächenpunkt die dazugehörigen Texturen evaluiert. Diese entscheidet auch, ob Schatteneffekte, Spiegelungen und Brechungen berücksichtigt werden. Abbildung 7.9 demonstriert die Mächtigkeit dieses Konzeptes. Das Beispiel ordnet zwei unterschiedliche Texturen schachbrettartig auf der Oberfläche eines Quaders an. Die eine berechnet Schatteneffekte, während die andere umgebende Objekte nicht berücksichtigt und nur ein lokales Beleuchtungsmodell implementiert. Auch auf der Kugel werden zwei unterschiedliche Texturen verwendet. Während beide Glanzlichter erzeugen, simuliert eine davon auch Brechungseffekte.

Neben der Implementation verschiedener Beleuchtungsmodelle, wie diejenigen von Phong [Pho75] und Whitted [Whi80], besitzt BOOGA auch eine ausgewachsene Shading Language nach dem Vorbild von RENDERMAN. Die Arbeit von Thomas Teuscher [Teu96] führt in die Details der Realisierung ein und zeigt beeindruckende Anwendungen auf, die die Mächtigkeit des Texturkonzeptes von BOOGA eindrucksvoll belegen.

7.4.1 Modellierung von Texturen

Die Klassenhierarchie der Texturen für den 3D-Fall ist in Abbildung 7.10 dargestellt.[5] Texturen werden als Attribute von Objekten

[5] Im 2D-Fall sind Texturen in gleicher Weise modelliert, weshalb wir uns auf die Beschreibung des 3D-Falls beschränken.

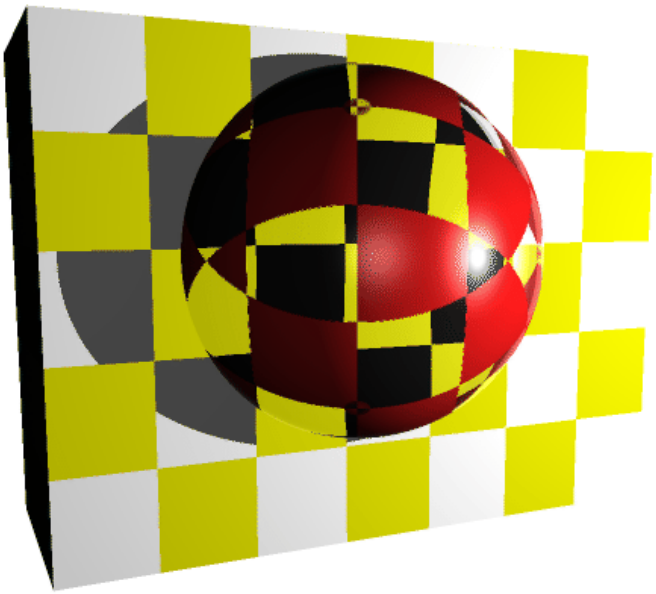


Abbildung 7.9
Das Texturmodell von Booga *erlaubt die flexible Beschreibung der Interaktion von Licht und Materie.*

verwaltet und können auf jeder Ebene im Szenengraphen spezifiziert werden. Im Texturierungsprozess sind daher alle Texturen auf dem Pfad von der Wurzel des Szenengraphen bis zum betrachteten Objekt auszuwerten. Die Einhaltung dieser Reihenfolge ist wichtig, damit auf Blattstufe alle Texturen der übergeordneten Ebenen "überschrieben" werden können.

Die Klasse `Texture3D` dient als Basis für die Texturfunktionen und gibt den Rahmen für deren Implementierung vor (siehe weiter unten). In `TextureUtilities` sind verschiedenste Funktionen enthalten, die für die Implementierung von Texturfunktionen nützlich sind. Beispielsweise stehen effiziente Implementationen der häufig verwendeten Noise- und Chaos-Funktionen von Perlin [Per85] zur Verfügung. Die zentrale Drehscheibe aller Texturfunktionen bildet aber die Klasse `Texture3DContext`. Sie enthält alle Informationen, die während des Texturierungsprozesses benötigt werden. Abschnitt 7.4.3 geht näher auf diese Klasse ein.

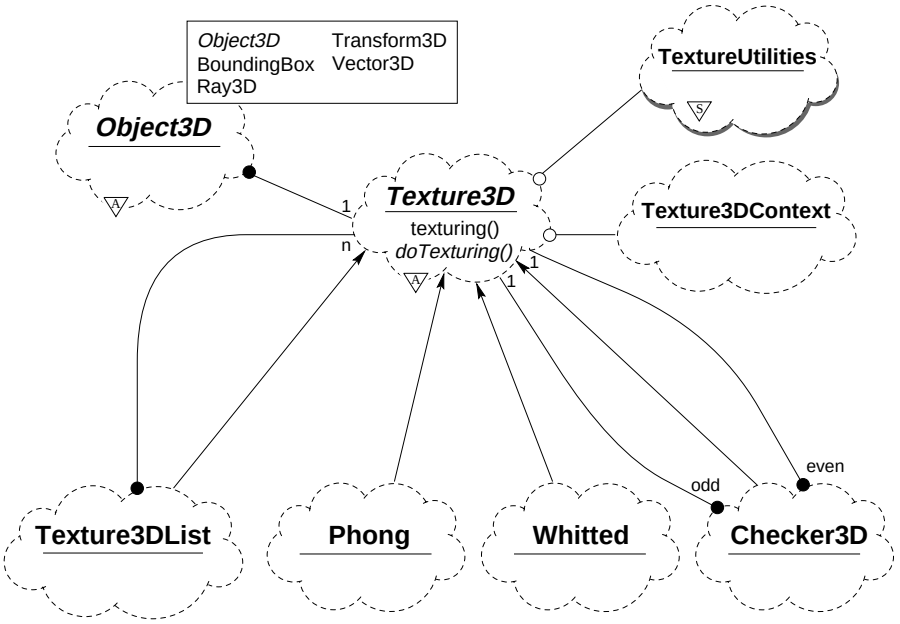


Abbildung 7.10 *Texturen werden als Attribute den Objekten zugeordnet.*

Mechanismen

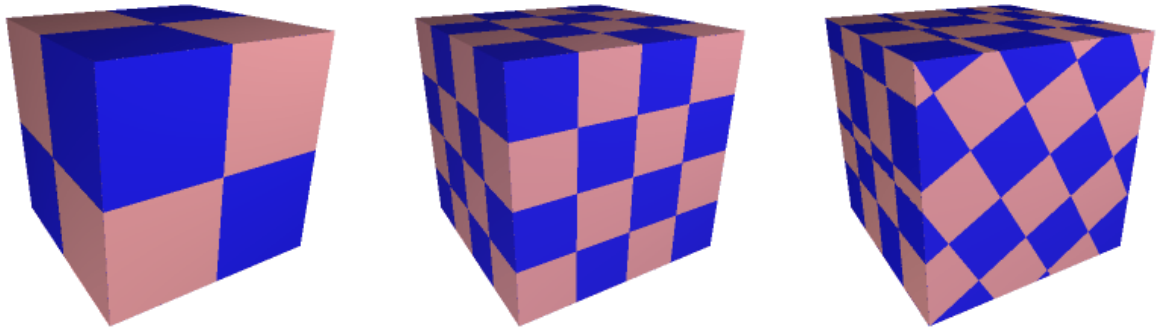
`texturing()`, `doTexturing()`

Die Methode `texturing()` der Klasse `Texture3D` enthält den Algorithmus für die Evaluierung aller Texturfunktionen eines Pfades. Grundsätzlich wird für jedes Texturobjekt die Hook-Methode `doTexturing()` aufgerufen, die die eigentliche Texturbrechnung durchführt:

```
void Texture3D::texturing(Texture3DContext& context,
                          const Path3D& path)
{
  // Ueber alle Objekte auf dem Pfad iterieren. Begonnen
  // wird bei der Wurzel des Szenengraphen, um dann
  // schrittweise zum Blatt zu gelangen.
  for (path.first(); !path.isDone(); path.next()) {
    // Texturobjekt extrahieren.
    Texture3D* texture = o->getTexture();

    // Falls ein Texturobjekt vorhanden ist, wird die
    // Texturfunktion evaluiert.
    if (texture != NULL)
      texture->doTexturing(context);
  }
}
```

In Abschnitt 7.4.4 ist am Beispiel der Whitted-Beleuchtungsgleichung die Implementierung einer `doTexturing()`-Methode diskutiert, weshalb an dieser Stelle nicht weiter darauf eingegangen wird. Über das Kontextobjekt (`Texture3DContext`, siehe Abschnitt



a) b) c)

Abbildung 7.11
Die Auswirkung von geometrischen Transformationen am Beispiel der Schachbrett-Textur.

7.4.3) kann auf die gesamte Szeneninformation zugegriffen werden. Dies erlaubt die Realisierung von ausgefeilten globalen wie lokalen Beleuchtungsmodellen.

7.4.2 Transformationen

Texturfunktionen werden üblicherweise bezüglich eines Einheitskörpers bzw. einer Einheitsfläche entworfen und realisiert. So bezieht sich beispielsweise die Implementation der Schachbrett-Textur von BOOGA auf einen Einheitswürfel, aliniert zu den drei Koordinatenachsen. Abbildung 7.11 a) zeigt die Anwendung dieser Textur auf den erwähnten Einheitswürfel. Mit Hilfe von geometrischen Transformationen kann in gewissen Grenzen eine Anpassung vorgenommen werden, wie die Fälle b) und c) demonstrieren. Im Fall b) wurde eine Skalierung um den Faktor 2 vorgenommen und in c) wurde die Textur nach der Skalierung um 30 Grad gegen den Uhrzeigersinn rotiert.

BOOGA ermöglicht die geometrische Transformation von Objekten *und* die Transformation von Texturen, was zu drei verschiedenen Koordinatensystemen führt:

Objektkoordinatensystem
: Keine, weder Objekt- noch Texturtransformationen werden berücksichtigt.

Weltkoordinatensystem
Wird in Weltkoordinaten gearbeitet, so sind alle dem Objekt zugeordneten geometrischen Transformation mit berücksichtigt. Texturtransformationen besitzen keinerlei Einfluss.

Texturkoordinatensystem
Wird mit Texturkoordinaten gearbeitet, werden alle Transformationen berücksichtigt, die zusammen mit der Textur spezifiziert wurden.

Konvertierungen zwischen den verschiedenen Koordinatensystemen sind jederzeit möglich und werden über das Kontextobjekt angeboten (siehe den nächsten Abschnitt).

7.4.3 Kontextinformation

Die Klasse `Texture3DContext` bez. `Texture2DContext` stellt alle Informationen für den eigentlichen Texturierungsprozess zur Verfügung. Jede Darstellungstechnik ist dafür verantwortlich, ein entsprechendes Kontextobjekt zu erzeugen, mit den aktuellen Werten zu initialisieren und dem Texturierungsprozess als Parameter zu übergeben. Eine Textur führt unter Verwendung dieser Information ihre jeweiligen Berechnungen durch, was zu einer Änderung des Kontextobjektes führt. Nach Beendigung des Texturierungsprozesses fliessen die manipulierten Daten zur nachfolgenden Darstellungsoperation zurück.

Den Texturfunktionen stehen folgende Informationen zur Verfügung, auf die sowohl lesend wie schreibend zugegriffen werden kann:

Szenengraph
Eine Texturfunktion hat den vollen Zugriff auf den Szenengraphen. Dies ist zum Beispiel für die Implementierung von globalen Beleuchtungsmodellen unabdingbar. Nur so kann die Wechselwirkung von Licht mit den Objekten der Szene simuliert werden.

Lichtquellen
Viele Beleuchtungsmodelle beziehen Lichtquellen in die Intensitätsberechnung ein. Die Angaben dafür können direkt aus dem Szenengraphen extrahiert werden. Damit nun aber

nicht jede Texturfunktion diese selbst ermitteln muss, besitzt das Kontextobjekt dafür vorgesehene Datenelemente. In einem Vorverarbeitungsschritt werden diese entsprechend initialisiert. Alle nachfolgenden Texturierungsprozesse beziehen sich nun auf diese Information, wodurch ein wiederholtes und aufwendiges Durchsuchen des Szenengraphen entfällt.

Kameraeinstellungen

Die aktuellen Kameraeinstellungen beeinflussen den Bilderzeugungsprozess wesentlich. In Texturfunktionen besonders wichtig sind der Standort und die Blickrichtung des Betrachters. Diese Parameter kombiniert mit der Lage der Lichtquellen führen beispielsweise zu sichtbaren Reflexionseffekten der Oberflächen und werden deshalb über das Kontextobjekt den Texturfunktionen angeboten.

Position

Texturberechnungen erfolgen für einen bestimmten Punkt auf der Oberfläche des Objektes. Die Texturfunktionen kann dessen Koordinaten über das Kontextobjekt erfragen und in die Berechnungen mit einbeziehen oder sogar verändern. Globale Beleuchtungsmodelle benötigen die aktuelle Position bezüglich des Weltkoordinatensystems. Aber auch die Objekt- und Texturkoordinaten werden angeboten (siehe auch den vorangegangenen Abschnitt).

Das Kontextobjekt ist mit der Position in einem beliebigen der drei Koordinatensysteme zu initialisieren. Konvertierungsmatrizen erlauben dann die automatische Umrechnung zwischen diesen Systemen. So kann zum Beispiel die Position im Texturkoordinatensystem verändert werden, worauf sie in den beiden anderen Koordinatensystemen automatisch nachgeführt wird.

Normale

Analog zu der für die Texturberechnung relevanten Oberflächenposition enthält das Kontextobjekt auch die Angaben zum zugehörigen Normalenvektor. Auch diese Information steht bezüglich Welt-, Objekt- und Texturkoordinatensystem zur Verfügung. Eine Manipulation dieses Vektors, wie zum Beispiel im bekannten Bump-Mapping-Verfahren [BN76] verwendet, wirkt sich auf alle Koordinatensysteme gleichzeitig aus.

Farbe

Das Kontextobjekt dient nicht nur als Informationsbehälter für die Parametrisierung des Texturierungsprozesses, sondern speichert auch dessen Ergebnis: eine Farbintensität. In diesem Sinn repräsentiert das Kontextobjekt den Mittler zwischen Darstellungstechnik auf der einen und Intensitätsberechnungen auf der anderen Seite. Der Informationsaustausch erfolgt also in beide Richtungen gleichermassen.

7.4.4 Implementierung des Whitted-Beleuchtungsmodells

Dieser Abschnitt ist der konkreten Realisierung einer Textur innerhalb von BOOGA gewidmet. Als Beispiel wird das Whitted-Beleuchtungsmodell [Whi80] gewählt, das in vielen Raytracern zum Einsatz kommt. Das Modell berücksichtigt Schatten-, Spiegelungs- und Brechungseffekte und simuliert somit auch globale Licht-Materie-Interaktionen. Durch den flexiblen Zugriff auf die gesamte Szeneninformation über das Kontextobjekt, lässt sich die Beleuchtungsgleichung in BOOGA leicht als Texturfunktion implementieren:

```
void Whitted::doTexturing(Texture3D& context)
{
  // Brechungseffekte werden durch das Aussenden von
  // Strahlen simuliert.
  Color refracting;
  Ray3D* refract = createRefractedRay(context,
                                      myTransparency,
                                      myRefractionIndex);
  bool hasRefractingHit = shadeSecondary(refract,
                                         refracting,
                                         context);

  // Spiegelungseffekte sind ebenfalls durch das
  // Aussenden von Strahlen simuliert.
  Color reflecting;
  Ray3D* reflect = createReflectedRay(context,
                                      myReflectivity);
  bool hasReflectingHit = shadeSecondary(reflect,
                                         reflecting,
                                         context);

  // Nun wird die Leuchtkraft der Lichtquellen in die
  // Berechnungen miteinbezogen.
  Real cosAlpha;      // Winkel zwischen Betrachter und
```

```
                    // Lichtquelle.
  Color illumination; // Kumulierter Farbwert.

  forEachLight(DirectedLight, light) {
    // Winkel zwischen Betrachter und Lichtquelle.
    cosAlpha  = context.getNormalWCS() ^
                light->getDirection(context);

    // Ist die Lichtquelle sichtbar?
    Real visible = light->isVisible(context);
    if (cosAlpha > 0 && visible > 0) {
      // Ja sichtbar -> Lichtanteil kumulieren.
      illumination += light->getColor() *
                      light->getLuminance() *
                      visible *
                      cosAlpha;
    }
  }

  // Nun werden alle Werte miteinander verrechnet.
  Color result;

  // Je groesser der Brechungs- und Spiegelungsanteil,
  // desto kleiner ist die Gewichtung der eigenen
  // diffusen Farbe.
  Real diffuseFactor = 1;

  // Gab es ein Brechungseffekt?
  if (hasRefractingHit) {
    // Brechungsanteil zum Resultat hinzufuegen.
    result += refracting*myTransparency;
    // Anteil der diffusen Farbe verringern.
    diffuseFactor *= (1-myTransparency);
  }

  // Gab es Spiegelungen?
  if (hasReflectingHit) {
    // Spigelungsanteil zum Resultat hinzufuegen.
    result += reflecting*(Real)myReflectivity;
    // Anteil der diffusen Farbe verringern.
    diffuseFactor *= (1-myReflectivity);
  }

  // Anteil der Lichtquellen beruecksichtigen.
  result += myDiffuse*illumination*diffuseFactor;

  // Result wird im Kontextobjekt abgelegt.
  context.setColor(result);
}
```

Eine Anwendung dieser Textur wurde bereits in Abbildung 7.9 gezeigt. Sowohl die Kugel (Brechungseffekte) als auch Teile des Quaders (Schatteneffekte) zeigen die Auswirkungen des Whitted-

Modells. In BOOGA wurden auf diese Weise die verschiedensten Beleuchtungsgleichungen implementiert und Erweiterungen, aber auch Experimente, gefördert.

8 Komponentenschicht von BOOGA

8.1 Überblick

Die Architektur von BOOGA definiert drei zentrale Abstraktionen: `World`, `Traversal` und `Component` (siehe Abbildung 5.2, Seite 65). Sie werden in der Komponentenschicht eingeführt, bauen aber auf den Mechanismen der Bibliotheks- und Frameworkschicht auf. Die `World`-Abstraktion entspricht der Einkapselung einer BOOGA-Szene ergänzt um zusätzliche Attribute. `Traversal` beschreibt eine Traversierungsstrategie für die in der Klasse `World` enthaltenen Szene und verknüpft diese mit einer `Component`, der Komponentenabstraktion von BOOGA. Sie repräsentiert eine Operation, die jeweils einem der vier Übergänge des BOOGA-Konzeptes zugeordnet werden kann. Eine Applikationen entspricht einer Aneinanderreihung von Komponenten, wie in Kapitel 9 weiter ausgeführt wird.

Die nun folgende Diskussion führt die drei Abstraktionen ein und zeigt die wesentlichen Mechanismen auf. Gemäss den jeweils zugeordneten Aufgaben ist die Beschreibung in die Teile *Szenenrepräsentation* (Abschnitt 8.2), *Szenentraversierung* (Abschnitt 8.3) und *Szenenverarbeitung* (Abschnitt 8.4) gegliedert.

8.2 Szenenrepräsentation

BOOGA-Szenen werden innerhalb der Komponentenschicht nicht direkt verwendet, sondern sie bilden den Hauptbestandteil einer *Welt*. Objekte dieses Typs sind Instanzen der Klassen `World2D` im 2D-Fall bez. `World3D` für den 3D-Fall. Abbildung 8.1 zeigt die Konstruktion der Klassen in einer Übersicht.

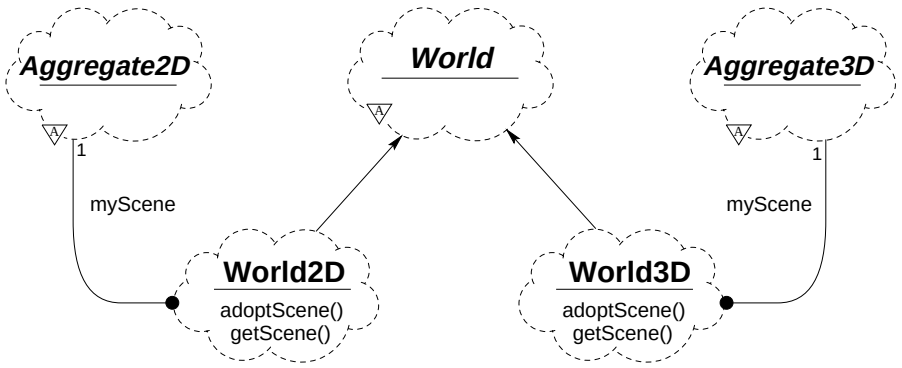


Abbildung 8.1 *Eine* BOOGA*-Szene wird in ein Weltobjekt gekapselt.*

Die Kapselung erlaubt die Speicherung von zusätzlichen Informationen, die nicht in der Szene enthalten sind und auch nicht direkt mit ihr im Zusammenhang stehen müssen. Ein Anwendungsfall ist die Aufbewahrung von applikationsspezifischen Datenelementen, zum Beispiel für die Festlegung der zugrundeliegenden Masseinheiten. Die Repräsentation der `World`-Abstraktion als eigenständige Klassen erlaubt zudem eine beliebige Erweiterung und Anpassung mittels Vererbung und bietet somit die von einem Framework geforderte Flexibilität.

Mechanismen

`getScene()`, `adoptScene()`

Die beiden Methoden `getScene()` und `setScene()` ermöglichen den freien Zugriff auf die in der Welt abgelegte Szene. Diese ist immer ein Aggregatobjekt, wodurch ein beliebiges Hinzufügen und Entfernen von Objekten leicht möglich ist.

In der aktuellen Entwicklungsphase von BOOGA sind keine weiteren Datenelemente in den Weltabstraktionen enthalten. Für jedes zusätzliche Element müssten aber entsprechende Zugriffsmethoden nach dem obigen Muster definiert werden.

8.3 Szenentraversierung

Die Traversierung der in einem Weltobjekt gekapselten Szene ist ein Grundmechanismus für die meisten Operationen eines Grafiksystems. Booga realisiert die Traversierungsstrategie als eigenständige Abstraktion. Dies entspricht dem bereits im Abschnitt 4.2.2 vorgestellten Prinzip und ermöglicht die flexible Anpassung an zukünftige Bedürfnisse. Abbildung 8.2 zeigt den Aufbau der zugehörigen Klassenhierarchie.

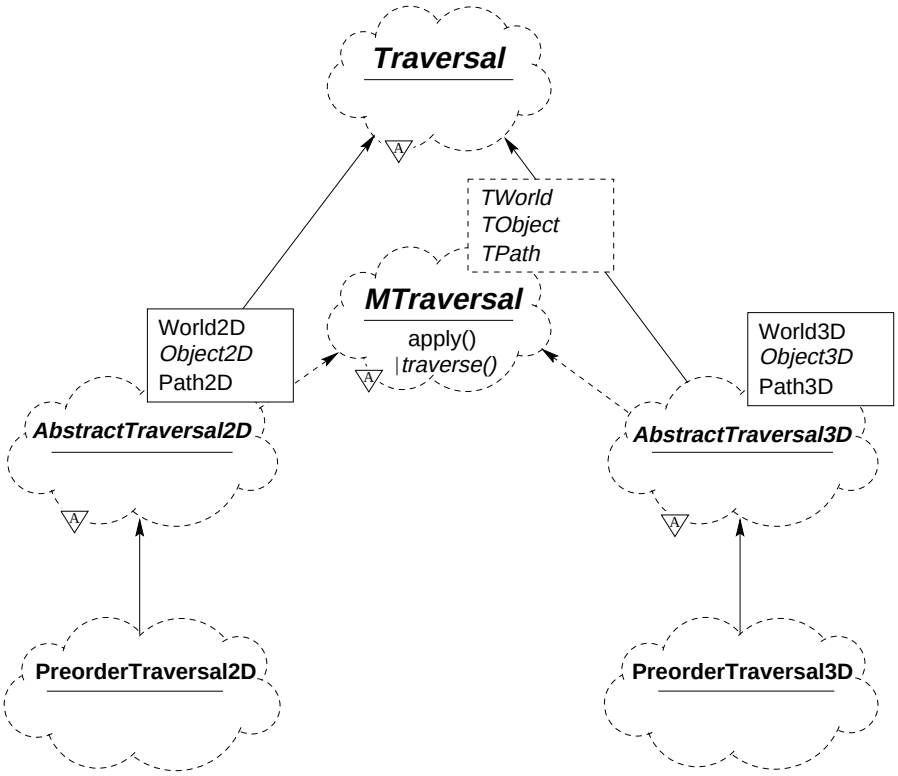


Abbildung 8.2 *Die Traversierungsstrategie ist in* Booga *als eigenständige Abstraktion realisiert.*

Durch den Einsatz der Template-Klasse `MTraversal` kann wie schon bei den Klassen der Frameworkschicht eine Codeverdoppelung bei der Bereitstellungen der Klassen für den 2D- und 3D-Fall vermieden werden. `AbstractTraversal2D` bez. `AbstractTraversal3D` dienen als Basisklassen für alle konkreten Traversierungen einer Szene. Die jeweiligen Implementierungen bauen auf den Basismechanismen der Frameworkschicht auf, wie weiter unten an einem Beispiel zu sehen ist.

Der Traversierung kommt eine Mittlerrolle zwischen Szenengraph auf der einen und Booga-Komponenten auf der anderen Seite

zu. Dabei sind die gegenseitigen Abhängigkeiten gross. Die Traversierung bietet die Objekte aus dem Szenengraphen der Komponenten in einer bestimmten Reihenfolge an, indem sie über den `dispatch()`-Mechanismus (siehe Abschnitt 8.4, Seite 113) mit ihr kommuniziert. Die Komponente kann ihrerseits über vordefinierte Rückgabewerte auf die Arbeitsweise der Traversierung Einfluss nehmen.

Die Aufgabe der Traversierung beschränkt sich aber nicht auf die Bereitstellung von Objekten für die Komponenten. Zusätzliche Operationen können während der Traversierung ausgeführt werden. Standardaufgaben sind beispielsweise die Verwaltung des aktuellen Pfades, die Berechnung von kumulierten Transformationsmatrizen oder das Sammeln von Texturobjekten. Aber auch ein Objektraum-Clipping kann durchaus direkt von der Traversierung durchgeführt werden.

Mechanismen

`apply()`, `traverse()`

Ein Traversierungsobjekte wird über die Methode `apply()` aktiviert. Diese ist in der Template-Klasse `MTraversal` für alle Traversierungstypen implementiert. In einem ersten Schritt initialisiert sie den Traversierungsprozess und startet ihn dann durch Aufruf der rein virtuellen Methode `traverse()`. Diese muss von jeder konkreten Traversierungsstrategie implementiert werden. Zwei Parameter werden für den Traversierungsprozess benötigt. Zum einen ist dies der zu verarbeitende Szenengraph, zum anderen die Komponente. Das Traversierungsobjekt kommuniziert mit der Komponenten über die `dispatch()`-Methode.

`CONTINUE, PRUNE, EXIT, UNKNOWN`

Die Rückgabewerte der `dispatch()`-Methode können die Arbeitsweise der Traversierung in gewissen Grenzen beeinflussen. Vier Konstanten sind hierfür vordefiniert:

`CONTINUE`
: Der Traversierungsprozess soll ohne Änderung weitergeführt werden. Der Rückgabewert zeigt an, dass die Komponente das der `dispatch()`-Methode übergebene Objekt akzeptieren und verarbeiteten konnte.

`PRUNE`
: Die Objekte des aktuellen Teilgraphen sollen durch die Traversierung nicht weiter berücksichtigt werden. Durch diesen

Rückgabewert kann eine Komponente nicht relevante Teile des Szenengraphen von der Bearbeitung ausschliessen. Ein naheliegender Anwendungsfall ist die Realisierung eines Clippings für Rendering-Verfahren.

`EXIT`
: Der Traversierungsprozess wird sofort abgebrochen. Dies können zum Beispiel Komponenten eines grafischen Editors ausnutzen, um zeitaufwendige Operation bei einer Benutzereingabe abzubrechen, was zu ausgezeichneten Antwortzeiten führt.

`UNKNOWN`
: Die Komponente ist nicht in der Lage, den übergebenen Objekttyp zu verarbeiten. Es ist nun Aufgabe der Traversierungsstrategie, auf diese Nachricht mit einer geeigneten Massnahme zu reagieren. So könnte im Falle eines geometrischen Objektes mit Hilfe der `decompose()`-Methode eine alternative Repräsentation generiert werden. Diese wird nun rekursiv traversiert und so anstelle des Originalobjektes der Komponenten angeboten.

Die Implementierung einer konkreten Traversierungsstrategie zeigt das untenstehende Beispiel. Verdeutlicht werden die Mechanismen anhand der Standardtraversierung, die alle Objekte einer Szene in Preorder-Reihenfolge besucht. Die Implementation ist als Wechselspiel zwischen den beiden Methoden `traverse()` und `doTraverse()` realisiert. Die erste verwaltet den aktuellen Pfad und stösst die eigentliche Traversierung an:

```
/*
 * Die traverse()- Methode verwaltet den aktuellen Pfad und
 * ruft fuer die eigentliche Traversierung die doTraverse()-
 * Methode auf.
 * Der Rueckgabewert steuert die Fortsetzung (true) oder den
 * Abbruch (false) der Traversierung.
 */
bool PreorderTraversal3D::traverse(Object3D* obj)
{
  myPath->append(object);
  bool retval = doTraverse(obj)
  myPath->removeLast();
  return retval;
}
```

Die `doTraverse()`-Methode übergibt der zugeordneten Kompo-

nenten das aktuelle Objekt als Parameter beim Aufruf von `dispatch()`. Abhängig vom Rückgabewert werden passende Aktionen ausgelöst:

```
/*
 * Der Rueckgabewert steuert die Fortsetzung (true) oder den
 * Abbruch (false) der Traversierung.
 */
bool PreorderTraversal3D::doTraverse(Object3D* obj)
{
  // Die Komponente aktivieren und Rueckgabewert speichern.
  Traversal::Result state = visitor->dispatch(obj);

  switch (state) {
    case EXIT:
      // Die Traversierung sofort abbrechen.
      return false;
    case PRUNE:
      // Den Teilbaum nicht weiter traversieren.
      return true;
    case UNKNOWN:
      // Ein unbekanntes Objekt kann in eine alternative
      // Repraesentation umgewandelt werden, falls es sich
      // um ein Primitive3D handelt.
      {
        Primitive3D* p = dynamic_cast<Primitive3D*>(obj);
        if (p != NULL) {
          // Alternative Repraesentation erzeugen...
          Object3D* decomposition = p->decompose();
          // ... und traversieren.
          bool retval = traverse(decomposition);
          delete decomposition;
          return retval;
        }

        // Falls kein Primitive3D vorliegt, wie bei CONTINUE
        // weiterfahren...
    case CONTINUE:
      // Die Traversierung soll normal fortgesetzt werden.
      for (long i=0; i<obj->countSubObject(); i++) {
        bool retval = traverse(obj->getSubObject(i);
        if (retval == false)
          return false;  // Traversierung abbrechen.
      }
  }
}
```

Das Beispiel der `doTraverse()`-Methode zeigt die enge Verknüpfung der Traversierungsabstraktionen mit den Komponenten, die im nächsten Abschnitt eingehend vorgestellt werden.

8.4 Szenenverarbeitung

Mit BOOGA werden Applikationen als Aneinanderreihung von Komponenten realisiert. Diese besitzen die Eigenschaften, welche bereits in Abschnitt 2.2.5 des Kapitels 2 postuliert wurden. So präsentiert sich eine BOOGA-Komponente dem Anwender als *Black-Box* und bietet ein *Dienst- und Konfigurationsinterface* an. Bei der Entwicklung einer Komponenten ist darauf zu achten, dass sie eine abgeschlossene, atomare *Teilaufgabe* bearbeitet. Dadurch ist es leicht möglich, dass Komponenten in unterschiedlichen Applikationen frei *kombiniert* werden können, wodurch sie optimal für eine *Wiederverwendung* vorbereitet sind.

BOOGA kann sich auf vier Komponententypen beschränken, die den vier Übergängen des BOOGA-Konzeptes (siehe Abbildung 5.1, Seite 63) entsprechen. Jeder Komponententyp wird durch eine eigene Klasse repräsentiert. Abbildung 8.3 zeigt die Hierarchie der Komponentenklassen in einer Übersicht.

Alle Komponenten erben von der gemeinsamen Basisklasse `Visitor`, die durch die Deklaration der Methode `dispatch()` die Verbindung von Traversierungsstrategien und Komponenten sicherstellt. Als Kompositionsmechanismus für die Aneinanderreihung von Komponenten wird direkt die Implementierunsgsprache C++ verwendet.

Die für ein reibungsloses Zusammenspiel aller Abstraktionen notwendigen Mechanismen werden nun im einzelnen diskutiert und an einem einfachen Beispiel illustriert.

Mechanismen

`execute(),`
`preprocessing(),`
`doExecute(),`
`postprocessing()`

Die Verwendung von Komponenten für eine Applikation muss ausgesprochen einfach sein. Ein Anwender bedient sich des Dienstinterfaces, das aus der einzelnen Methode `execute()` besteht und über alle Komponententypen hinweg gleich aufgebaut ist. In der Template-Basisklasse `Component` ist hierfür die Deklaration

```
TWorldResult* execute(TWorldInput*)
```

vorgesehen. Je nach Komponententyp werden die konkreten Abstraktionen für die Eingabe- und Ausgabewelten verwendet.

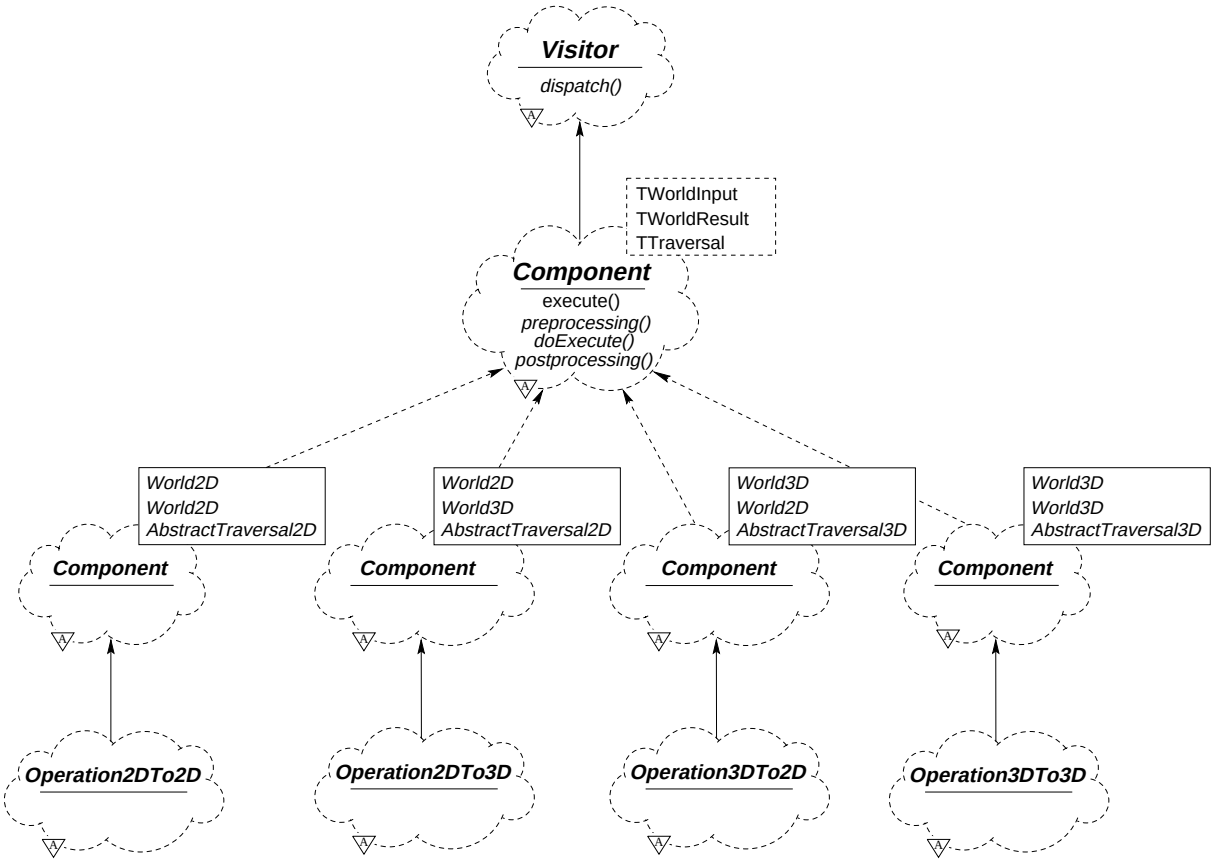


Abbildung 8.3
Jeder Übergang des zugrundeliegenden BOOGA*-Konzeptes wird durch eine eigene Klasse repräsentiert.*

`Operation3DTo2D` besitzt beispielsweise eine `execute()`-Methode mit einem Parameter vom Typ `World3D` und erzeugt ein Weltobjekt vom Typ `World2D`.

Die `execute()`-Methode lässt sich nicht überschreiben und ist somit für alle Komponententypen gleich. Sie enthält aber variable, durch Hook-Methoden realisierte Teile, die eine Anpassung in weiten Grenzen erlaubt. Das folgende Codefragment zeigt den Aufbau von `execute()` schematisch auf:

```
TWorldResult* Component::execute(TWorldInput* world)
{
  // Weltobjekt fuer das Resultat erzeugen.
  result = new TWorldResult;
  ...
```

```
    // Die eigentliche Operation teilt sich in
    // drei Schritte auf.
    preprocessing();
    doExecute();
    postprocessing();

    return result;
  }
```

Die Standardimplementationen der `preprocessing()` und `postprocessing()` Methoden führen keinerlei Aktionen aus. `doExecute()` startet hingegen den Traversierungsprozess unter Verwendung einer spezifizierbaren Strategie.

`dispatch()`

Traversierungsobjekte und Komponenten kommunizieren über die `dispatch()`-Methode miteinander. Diese wird in `Visitor` (die gemeinsame Basisklasse aller Komponenten) deklariert und ist verantwortlich, die passenden Verarbeitungsmethoden aufzurufen. BOOGA verwendet hierzu eine Variation des Visitor Patterns [GHJV95], da das Standardvorgehen einige Nachteile besitzt, wie bereits in Abschnitt 4.2.2 erläutert wurde.

Das Grundproblem liegt aber bei der Implementationssprache C++, die kein *Multidispatch*[1] unterstützt. Grundsätzlich existieren zwei Techniken, um dieses Defizit in gewissen Grenzen auszugleichen. Zum einen kann der Dispatch-Mechanismus mit Hilfe von eigenen Sprungtabellen vollständig selbst implementiert werden. Meyers gibt in [Mey96] eine gute Übersicht verschiedener Implementierungsvarianten. Verwendung findet diese Technik zum Beispiel in OPENINVENTOR (siehe Abschnitt 4.3.3). Der Aufwand für die Realisierung ist allerdings gross. Auch können die Sprungtabellen bei komplexen Vererbungshierarchien ohne zusätzliche Metainformation über die Relationen zwischen den Klassen nicht korrekt aufgebaut werden.

Der zweite Ansatz verwendet das Visitor Pattern und ist nur für ein Doubledispatch (Methodenselektion aufgrund zweier polymorpher Argumente) einsetzbar. Für unseren Anwendungsfall ist dies allerdings keine Einschränkung. Wesentlich gewichtiger ist aber der Nachteil, dass eine unabhängige Erweiterung der beiden am Prozess beteiligten Klassenhierarchien (Objekt- und Komponentenhierarchie) nicht möglich ist. Mehrere Varianten des Visitor Patterns

[1] Mit Multidispatch wird der Prozess der Methodenselektion aufgrund mehrerer polymorpher Argumente bezeichnet.

wurden vorgeschlagen, die die Defizite des Originals beheben. Für unseren Fall eignet sich die *Extrinsic Visitor* Technik [Nor96] ausgezeichnet. Hier wird durch direktes Ermitteln der Typeninformation ein Dispatch ausgeführt.

Das folgende Codefragment zeigt die Implementation der `dispatch()`-Methode für die `Wireframe`-Komponente (siehe auch Abschnitt 9.3.1, Seite 125). Sie erlaubt die Erzeugung einer 2D-Drahtgitterdarstellung einer 3D-Szene und ist daher von Typ `Operation3DTo2D`. Die Komponente kann nur Dreiecke (`Triangle3D`) verarbeiten und schickt alle anderen Objekte mit dem Vermerk `UNKNOWN` an den Traversierungsprozess zurück. Dieser wird gegebenenfalls eine alternative Repräsentation erzeugen und dann diese der `Wireframe`-Komponenten erneut anbieten.

```
Traversal::Result Wireframe::dispatch(Makeable* obj)
{
  // Falls ein Dreieck dispatch() uebergeben wurde,
  // wird die zugehoerige visit()-Methode aufgerufen.
  if (dynamic_cast<Triangle3D*>(obj) != NULL)
    // Verarbeitungsroutine aufrufen.
    return visit((Triangle3D*) obj);

  // ... Gegebenenfalls Tests fuer weitere Objekttypen ...

  // Falls wir an diese Stelle gelangen, lag kein
  // verarbeitbares Objekt vor. UNKNOWN an den
  // Traversierungsprozess schicken, damit eine
  // alternative Repraesentation erzeugt wird.
  return Traversal::UNKNWON;
}
```

Erstaunlicherweise ist die Ausführungsgeschwindigkeit dieser Technik äusserst gut, obschon sie von der Anzahl verarbeiteter Objekttypen linear abhängt. Verglichen mit dem Standard Visitor Pattern, das in einer früheren Phase von BOOGA eingesetzt wurde, konnte bei einer kleinen Zahl von unterschiedenen Objektentypen sogar eine Verbesserung der Laufzeit gemessen werden.

`visit(...)`

Für jeden unterstützten Objekttyp bietet eine Komponente eine zugehörige `visit()`-Methode an. Die Aufgabe der `dispatch()`-Methode ist es die korrekte Verarbeitungsroutine zu identifizieren und aufzurufen. Die `Wireframe`-Komponenten akzeptiert beispielsweise nur den Objekttyp `Triangle3D` und deklariert daher lediglich

```
Traversal::Result visit(Triangle3D* obj);
```

Natürlich wird eine beliebige Anzahl von `visit()`-Methoden unterstützt. Die meisten der in BOOGA enthaltenen Komponenten beschränken sich allerdings auf einige wenige.

`visit()`-Methoden beinhalten die eigentliche Funktionalität einer Komponenten. Diese ist also erst durch die Summe ihrer `visit()`-Methoden charakterisiert. Sie können durch entsprechende Rückgabewerte zusätzlich den Traversierungsprozess beeinflussen, wie bereits im vorangegangen Abschnitt erläutert wurde. Die vereinfachte Implementation für die `Wireframe`-Komponente besitzt folgenden Aufbau:

```
Traversal::Result Wireframe::visit(Triangle3D* obj)
{
  // Parameter fuer die Viewing-Transformation ermitteln.
  const Viewing3D* viewing = camera->getViewing();

  // Die aktuelle globale Modellierungs-Tranformation wird
  // vom Traversierungsobjekt verwaltet.
  const TransMatrix3D* t =
              traversal->getPath()->getTransform();

  // Die Weltkoordinaten der Eckpunkte des Dreiecks muessen
  // nach Bildschirmkoordinaten umgewandelt werden.
  Vector2D v1 = viewing->transformWorld2Screen(
                   t->transformAsPoint(obj->getVertex(0)));
  Vector2D v2 = viewing->transformWorld2Screen(
                   t->transformAsPoint(obj->getVertex(1)));
  Vector2D v3 = viewing->transformWorld2Screen(
                   t->transformAsPoint(obj->getVertex(2)));

  // Die Berechnung von Farbinformation fuer die zu
  // erzeugenden Geradensegmente wurden in diesem
  // vereinfachten Beispiel weggelassen.
  ....

  // 2D-Geradensegmente generieren und in die Resultatwelt
  // aufnehmen.
  result->append(new Line2D(v1, v2, v3));
  result->append(new Line2D(v2, v3, v1));
  result->append(new Line2D(v3, v1, v2));

  // Die Operation war erfolgreich.
  return Traversal::CONTINUE;
}
```

8.5 Zusammenspiel

Die beschriebenen Abstraktionen und Mechanismen der Komponentenschicht sind eng miteinander verknüpft. Erst ihr Zusammenspiel erlaubt die Lösung komplexer Aufgaben. Das Interaktionsdiagramm in Abbildung 8.4 zeigt es anhand einer einfachen Beispielsszene auf. Verwendung findet wiederum die `Wireframe`-Komponente.

Das Szenario ist in fünf Schritte gegliedert:

1. Die Szene, bestehend aus einem Aggregat (`List3D`) und zwei geometrischen Objekten (`Triangle3D`, `Sphere3D`), kann nicht direkt durch die `Wireframe`-Komponente verarbeitet werden. Der Rückgabewert `UNKNOWN` weist deshalb die Traversierung an, mit den im Aggregat enthaltenen Objekten weiterzufahren.

2. Als erstes wird das Dreieck der `dispatch()`-Methode übergeben. Diese reicht das Objekt an `visit(Triangle3D*)` weiter, wo drei Geradensegmente erzeugt und der Resultatwelt hinzugefügt werden.

3. Der Rückgabewert `CONTINUE` weist die Traversierung an, das Kugelobjekt für die Verarbeitung durch die Komponente bereitzustellen. Dieser Objekttyp kann allerdings nicht direkt verarbeitet werden, worauf die Traversierung mit Hilfe der `decompose()`-Methode eine alternative Repräsentation erzeugt.

4. Das resultierende Aggregat enthält eine Menge von Dreiecken. Allerdings ist die `Wireframe`-Komponente immer noch nicht in der Lage, diese direkt zu verarbeiten.

5. Deshalb wird die Traversierung die im Aggregat enthaltenen Dreiecke mit `getSubObject()` aus dem Aggregat extrahieren und einzeln der `dispatch()`-Methode übergeben. Nun kann die Komponente entsprechende 2D-Geradensegmente erzeugen und in die Resultatwelt integrieren.

Das gezeigte Beispiel löst nur eine Teilaufgabe, indem die Komponente aus einer 3D-Eingabeszene eine Menge von 2D-Geradensegmenten erzeugt. Eine vollständige Applikation muss

aber zusätzlich die 3D-Szene zum Beispiel durch Einlesen einer externen Repräsentation aufbauen. Nachdem die `Wireframe`-Komponente ihre Arbeit verrichtet hat, sollte das Resultat beispielsweise in einem Window präsentiert werden. Diese Aufgaben können durch eine Aneinanderreihung von Komponenten realisiert werden, die jeweils eine Teilaufgabe bearbeiten. Erst die Kombination führt zu BOOGA-Applikationen. Das nächste Kapitel zeigt einige konkrete Beispiele.

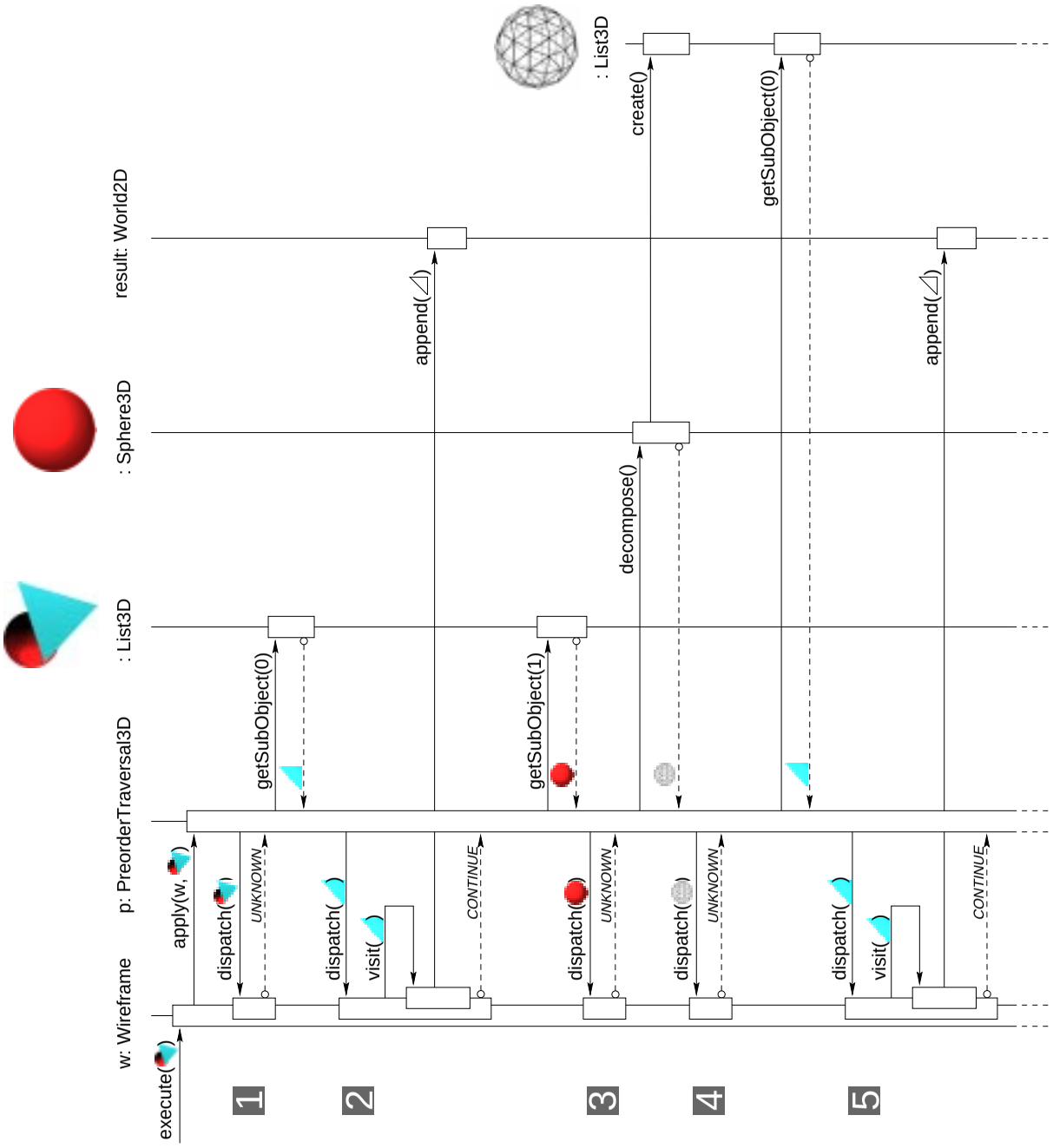


Abbildung 8.4

Das Zusammenspiel der Hauptabstraktionen von BOOGA.

9 Applikationsentwicklung mit BOOGA

Das vorliegende Kapitel widmet sich ganz der Applikationsentwicklung mit BOOGA. Diese ist wesentlich vom Aufbau und den Mechanismen des Frameworks geprägt. Selbst der Begriff der Applikation muss speziell für diesen Kontext definiert werden.

Das Kapitel ist in drei Teile gegliedert. Begonnen wird mit der Definition des Applikationsbegriffs. Die eigentliche Applikationsentwicklung wird dann durch ein Vorgehensmodell unterstützt, welches sich in der praktischen Arbeit mit BOOGA als günstig erwiesen hat. Abschliessend werden einige typische Beispielapplikationen vorgestellt, die die Natur von BOOGA-Applikationen besonders gut aufzeigen.

9.1 Der Applikationsbegriff

Eine BOOGA-Applikation wird durch Aneinanderfügen von existierenden Komponenten gebildet. Im Kontext von BOOGA ist daher die Applikation wie folgt definiert:

> *Eine Applikation ist eine Aneinanderreihung von Komponenten mit dem Ziel, eine übergeordnete Aufgabe zu erfüllen. Die Instanzierung und Aktivierung sind in ein Hauptprogramm eingebettet.*

Im einfachsten Fall besteht eine Applikation aus einer einzelnen Komponente, deren Instanzierung und Aktivierung wie folgt vorgenommen werden:

```
int main()
{
  Komponente k;             // Instanzieren ...
  k.execute(new World());   // ... und aktivieren.
  return 1;
}
```

Komplexere Applikationen setzen sich hingegen aus Dutzenden von Komponenten zusammen, um die an sie gestellte Anforderungen erfüllen zu können. Natürlich ist es nicht möglich, für alle denkbaren Anwendungsszenarien entsprechende Komponenten vorrätig zu haben. Deshalb gibt es grundsätzlich zwei Aufgaben bei der Applikationsentwicklung:

1. Komponentenentwicklung

Eine neue Komponente wird unter Verwendung der Abstraktionen und Mechanismen der Frameworkschicht entwickelt. Es ist im allgemeinen nicht notwendig, diese Schicht selbst anzupassen oder zu erweitern. Die vorgegebenen Strukturen werden also als Black-Box-Artefakte betrachtet, die der Anwender den Bedürfnissen entsprechend konfigurieren muss. Gute Kenntnisse des zugrundeliegenden Designs von BOOGA sind allerdings unabdingbar.

Bei der Entwicklung sollte zudem darauf geachtet werden, dass jede Komponente eine möglichst einfache, atomare Teilaufgabe löst. Dadurch steigt die Chance, dass eine Komponente in einem völlig anderen Zusammenhang wiederverwendet werden kann. Im übrigen ist es auch ein gutes Vorgehen, bei der Entwicklung neuer Komponenten auf bestehende zurückzugreifen. Zum Beispiel kommt die Komponente `CollectorFor` immer zum Einsatz, wenn Objekte eines bestimmten Typs in einem Szenengraphen gesucht werden müssen. Die `Wireframe`-Komponente ermittelt beispielsweise in einem Vorverarbeitungsschritt mit Hilfe von `CollectorFor` die zu berücksichtigenden Lichtquellen. Im Idealfall kann auf diese Weise eine neue Komponente durch Komposition bestehender erstellt werden.

2. Komposition von Applikationen
Applikationen werden durch das Zusammenfügen von Komponenten erstellt. Dazu sind Kenntnisse der Abstraktionen und Mechanismen der Komponentenschicht, aber auch ein guter Überblick über die bereits vorhandenen Komponenten notwendig. Ein Applikationsentwickler könnte zusätzlich durch geeignete Komponentenbrowser in seiner Arbeit unterstützt werden. Auch ein Werkzeug, das die visuelle Komposition von Applikationen erlaubt, wäre äussert hilfreich und effizienzsteigernd.

Die beiden Grundaufgaben ignorieren allerdings die Tatsache, dass ein Framework weiterentwickelt werden muss damit es am Leben bleibt.[1] Deshalb muss im Falle von BOOGA auch berücksichtigt werden, dass eine Weiterentwicklung der Abstraktionen und Mechanismen aller Schichten möglich ist. Diese Aufgaben können allerdings erst nach längerer Einarbeitungszeit in BOOGA selbständig bearbeitet werden.

Der resultierende Applikationscode ist meist von trivialer Natur. Trotzdem würde es die Applikationsentwicklung weiter vereinfachen, falls diese mit visuellen Programmiertechniken erfolgen kann. Dazu ist eine geeignete grafische Notation notwendig, wie wir sie auch für BOOGA vorgeschlagen haben (siehe Anhang A). Obwohl zur Zeit noch kein Werkzeug für die grafische Komposition existiert, kommt die Notation für die Illustration von Applikationen zum Einsatz, wie im Abschnitt 9.3 an einigen Beispielen gezeigt wird.

9.2 Vorgehensmodell für die Applikationsentwicklung

Die Framework-basierte Entwicklung von Applikationen prägt den Entwicklungsprozess wesentlich. Damit schlüsselfertige Abstraktionen wiederverwendet werden können, werden die Freiheiten bei der Entwicklung bewusst eingeschränkt. Eine Applikation adaptiert die

[1] *"A program that is used in a real-world environment necessarily must change or becomes less and less useful in that environment." [Som89, zitiert Lehman und Belady]*

vorgegebene generische Lösung und berücksichtigt die Möglichkeiten des Frameworks. Dadurch entsteht im Normalfall aus softwaretechnischer Sicht nicht die "optimale" Architektur für ein gegebenes Problem. Die Entwicklungszeit kann aber drastisch reduziert werden, und einmal geleistete Entwicklungen amortisieren sich viel rascher.

Damit ein Framework effizient eingesetzt werden kann, ist ein darauf abgestimmtes Vorgehensmodell für die Applikationsentwicklung notwendig. Dieses kann sowohl implizit als auch explizit als eine Menge von Regeln oder wohl definierten Arbeitsschritten vorliegen. Das Vorgehensmodell manifestiert in gewisser Weise die bei der Entwicklung des Frameworks prägenden Ideen. Diese sollen natürlich auch in allen entstehenden Applikationen spürbar bleiben.

Während der Entwicklung von BOOGA und bei seinem Einsatz in diversen Projekten hat sich ein Vorgehensmodell für die Applikationsentwicklung bestehend aus fünf Schritten als günstig erwiesen:

1. Zerlegung

In einem ersten Schritt wird die zu erstellende Applikation in eine Menge von Verarbeitungsschritten ("Komponenten") zerlegt. Die Zerlegung basiert auf der Analyse der Anforderungen, was zu einer Identifikation von Funktionspunkten[2] führt. Nun wird jeder Funktionspunkt einem der vier Übergänge des BOOGA-Konzeptes zugeordnet. Dabei darf nicht notwendigerweise ein Funktionspunkt mit einer Komponenten gleichgesetzt werden. Auf diese Weise entsteht eine initiale Architektur der Applikation.

Resultat: Initiale Architektur.

2. Überprüfung

Das Resultat des ersten Schrittes ist eine grobkörnige Zerlegung der Applikation in "Komponenten" ohne die Berücksichtigung bereits bestehender BOOGA-Komponenten. Ein Ziel einer jeden Entwicklung ist es, möglichst viele Abstraktionen direkt wiederverwenden zu können. Zudem sollten neu entwickelte Teile nach Möglichkeit auch in anderen Applikationen eingesetzt werden können.

[2]Funktionspunkte entsprechen in vielen Fällen herkömmlichen Funktionen. Sie können allerdings auch Aufgaben eines Systems bezeichnen, zu deren Lösung mehrere Funktionen nötig sind.

Diese Forderungen führen zu einer Überprüfung und gegebenenfalls Verfeinerung der initialen Zerlegung. Diese berücksichtigt existierende und identifiziert neu zu entwickelnde Komponenten. Dabei sind vor allem zwei Punkte zu beachten:

1. Neue Komponenten sollten primitive (atomare) Operationen behandeln. Dadurch wird es wahrscheinlicher, dass sie auch in anderen Applikationen eingesetzt werden können.
2. Die Zerlegung sollte so vorgenommen werden, dass möglichst viele der existierenden Komponenten verwendet werden können. Dadurch kann der Anteil der Neu- oder Weiterentwicklung minimiert werden.

Die Zerlegung der Applikation in BOOGA-Komponenten ist die wohl schwierigste Aufgabe bei der Applikationsentwicklung. Dieser Schritt ist entscheidend für den Aufwand, der in den nachfolgenden Phasen getrieben werden muss. Erfahrene Entwickler werden immer versuchen, möglichst grosse der Teile der Applikationen aus bereits existierenden Komponenten zusammenzubauen.

Resultat: Verfeinerte Architektur, existierende Komponenten sind berücksichtigt.

3. Implementierung

Der dritte Schritt umfasst die Implementierung der neuen Komponenten, möglichst unter Verwendung bestehender Komponenten, Abstraktionen und Mechanismen.

Es hat sich als günstig erwiesen, bereits in einer frühen Entwicklungsphase Simulatoren für die neuen Komponenten bereitzustellen. Diese weisen die gleichen Schnittstellen wie die richtigen Komponenten auf, bieten aber nur eine unvollständige oder überhaupt kein Funktionalität. Dadurch kann sehr früh zum vierten Schritt übergegangen werden, um das Verhalten der gesamten Applikation zu überprüfen. Dieses Vorgehen ist besonders bei der Applikationsentwicklungen in einem kleinen Team von Vorteil, da mit Hilfe der Simulatoren die Applikation zusammengebaut werden kann und so Schnittstellentests durchgeführt werden können. Einzelne Teammitglieder werden dann Simulatorkomponenten durch echte er-

setzen, sobald die entsprechenden Entwicklungen abgeschlossen sind. Die so vervollständigte Applikation steht sofort allen Teammitgliedern zur Verfügung.

Resultat: Neue Komponenten, Objekttypen oder Traversierungsstrategien.

4. Komposition

Im vierten Schritt wird die Applikation aus den nun zur Verfügung stehenden Komponenten zusammengebaut. Zur Zeit ist dies nur mit Hilfe von C++ Codierung möglich. Eine spezialisierte Kompositionssprache oder auch eine visuelles Kompositionswerkzeug könnte diesen Schritt vereinfachen.

Resultat: Lauffähige Applikation.

5. Integration

Der abschliessende Schritt sichert die getätigten Investitionen, indem neue Komponenten, Objekttypen oder Traversierungsstrategien ins Basissystem übernommen werden. Dadurch stehen sie für zukünftige Entwicklungen zur Verfügung. Es ist aber darauf zu achten, dass keine applikationsspezifischen Elemente übernommen werden, da sie verwaltet und erhalten werden müssen, aber kaum in neuen Anwendungen zum Einsatz kommen.

Der Integrationsschritt kann aber auch die Weiterentwicklung von bestehenden Mechanismen beinhalten, falls sich diese als zu inflexibel oder zu wenig generisch erwiesen haben.

Resultat: Erweitertes Basissystem.

9.3 Beispielapplikationen

Die nun folgenden Abschnitte diskutieren vier Beispielapplikationen. Sie sind so gewählt, dass bei jeder Applikation einer der vier Übergänge des BOOGA-Konzeptes im Zentrum steht. Der erste Applikationstyp konzentriert sich auf den Übergang 3D → 3D, dann werden 2D → 2D, 2D → 3D und 3D → 3D im Vordergrund stehen.

9.3.1 Rendering (3D → 2D)

Ein wichtiges Anwendungsgebiet von BOOGA ist die Implementierung von Rendering-Verfahren. Diese setzen sich üblicherweise aus den drei Elementen *Einlesen* einer Szenenbeschreibung, *Rendering-Schritt* und *Präsentation* des Resultates zusammen. Mit BOOGA bietet es sich natürlich an, diese Schritte direkt als Komponenten zu realisieren.

Als Ausgangsvariante betrachten wir die Darstellung einer Szene mit der Wireframe-Methode. Sie wird benutzt, falls mit möglichst geringem Zeitaufwand das Abbild einer Szene generiert werden soll. Jedes Objekt wird durch eine Menge von Geradensegmenten repräsentiert, die durch eine nachfolgende Rasterkonvertierung in eine Pixmap übertragen werden. Abbildung 9.1 zeigt die Zerlegung der Wireframe-Applikation dargestellt mit der im Anhang A beschriebenen grafischen Notation. Diese bietet für jeden Komponententyp eine grafische Repräsentation.

Abbildung 9.1
Die Darstellung einer 3D-Szene mit Hilfe des Wireframe-Modells. Erzeugt werden eine Pixmap und die Ausgabe für einen PostScipt-Drucker.

Die folgenden Komponenten kommen in der Wireframe-Applikation zum Einsatz:

`Parser3D`
: Die `Parser3D`-Komponente verarbeitet eine beliebige 3D-Szenenbeschreibung[3] und baut die zugehörige Objektstruktur im Arbeitsspeicher auf.

`Wireframe`
: Die `Wireframe`-Komponente kapselt die Darstellungsstrategie ein und ist entsprechend den Ausführungen aus Abschnitt 8.4 implementiert. Als Resultat wird eine 2D-Szene erzeugt, die aus einer Menge von Geradensegmenten besteht. Der Texturierungsprozess liefert Farbwerte für alle Start- und Endpunkte, die für die nachfolgenden Verarbeitungsschritte zur Verfügung stehen.

`PSWriter`
: Die Elemente einer 2D-Szene werden durch die `PSWriter`-Komponente in PostScript-Kommandos umgewandelt und in einer Datei gespeichert. Diese wird in einem dafür vorgesehenen Drucker verarbeitet, wodurch ein Abbild der Szene entsteht.

`Rasterize`
: Die `Rasterize`-Komponente führt eine Rasterkonvertierung aller Vektorobjekte der 2D-Szene durch. Das Resultat dieser Operation ist eine einzelne Pixmap.

`Display`
: Die 2D-Szene wird durch die `Display`-Komponente nach Pixmap-Objekten durchsucht. Wird eine Pixmap gefunden, so erzeugt die Komponente ein Window und stellt sie darin dar.

Der folgende C++ Code ist für die Komposition der Wireframe-Applikation notwendig[4]:

[3] Siehe Anhang B für die Beschreibung der Szenenbeschreibungssprache von Booga.

[4] Die Überprüfung von Fehlerzuständen und die Berücksichtigung von Kommanodzeilenargumenten wurden weggelassen, um den Code möglichst einfach zu halten.

```
int main()
{
  // Szenenbeschreibung mit Hilfe der Parser-Komponenten
  // einlesen.
  Parser3D parser;
  World3D* world = new World3D;
  parser.execute(world);

  // Die Wireframe-Darstellung der Szene generieren.
  Wireframe wireframe;
  World2D* result1 = wireframe.execute(world);

  // Das Resultat in eine PostScript-Datei ausgeben.
  PSWriter writer;
  writer.execute(result1);

  // Eine Rasterkonvertierung durchfuehren...
  Rasterizer rasterize;
  World2D* result2 = rasterize.execute(result1);

  // ... und das Resultat direkt in einem Fenster
  // darstellen.
  Display display;
  display.execute(result2);

  // Aufraeumen.
  delete result1;
  delete result2;
  delete world;

  return 1;
}
```

Die Wireframe-Applikation kann nun leicht für ein alternatives Rendering-Verfahren adaptiert werden. Für die Implementierung eines Raytracers müsste zum Beispiel lediglich die `Wireframe`-Komponente durch die `Raytracer`-Komponente ersetzt werden.

Eine dritte Variante ist in Abbildung 9.2 zu sehen. Hier arbeiten zwei unterschiedliche Rendering-Verfahren an der Darstellung derselben Szene. Beispielsweise könnten auf diese Weise das Raytracing- mit dem Z-Buffer-Verfahren kombiniert werden. Für diese Applikation kommen folgende Komponenten zum Einsatz:

`Parser3D`
: Das Einlesen einer beliebigen 3D-Szenenbeschreibung und den Aufbau der zugehörigen Objektstruktur übernimmt wiederum die `Parser3D`-Komponente.

`Split`
: Die `Split`-Komponente teilt die 3D-Szene in zwei disjunkte

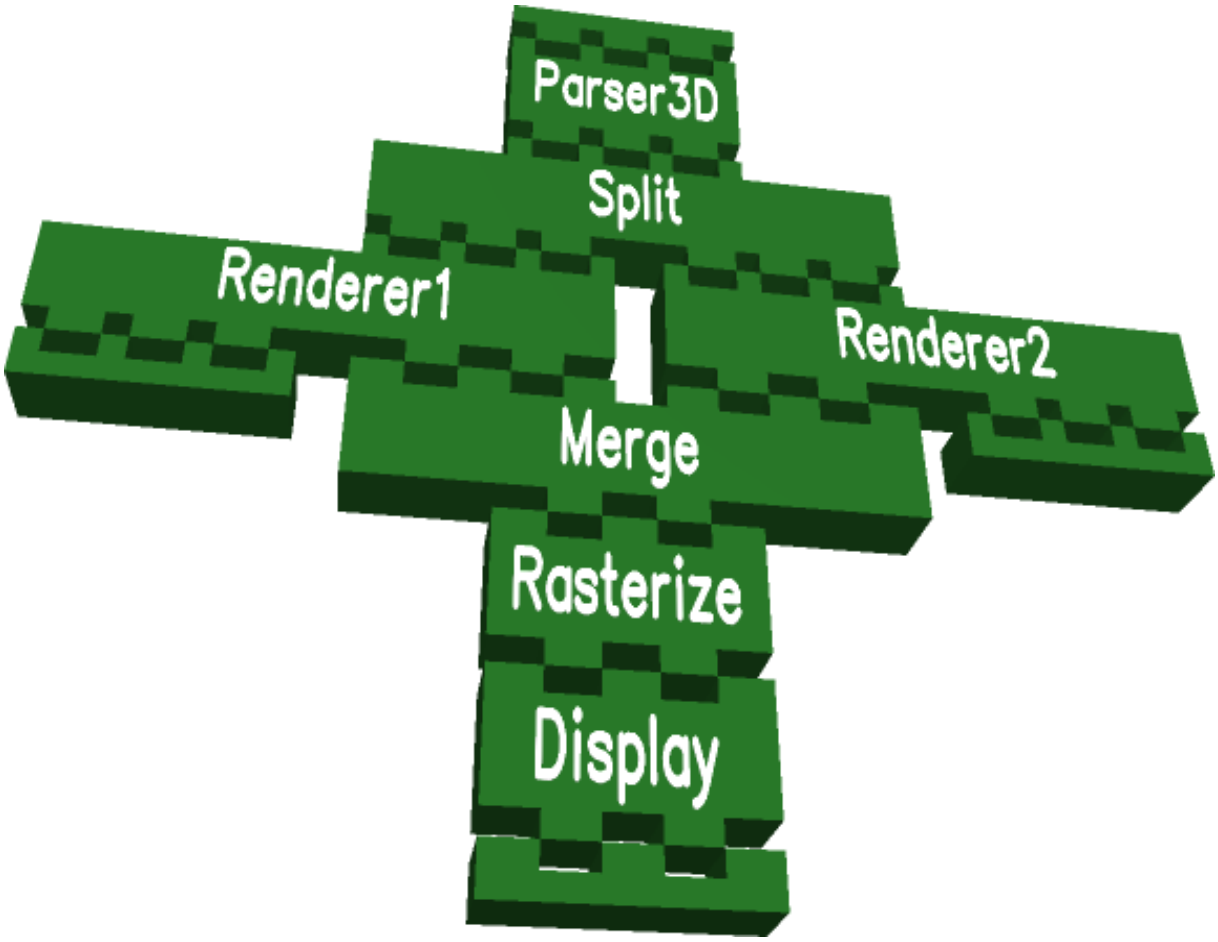


Abbildung 9.2
Mehrere Renderer-Komponenten können für die Darstellung einer einzelnen Szene zusammenwirken.

Teilszenen auf. Als Kriterium für die Unterteilung dient beispielsweise die Distanz zum Betrachter. Dadurch könnte die eine Szene ausschliesslich Objekte im Vordergrund beinhaltet, während sich die zweite Szene aus Hintergrundobjekten zusammensetzt.

Renderer1, Renderer2
Zwei Rendering-Komponenten erzeugen Abbilder der beiden Szenen. Die Komponenten werden aufgrund des Unterteilungskriteriums der `Split`-Komponenten ausgewählt. So müssen in unserem Beispiel Objekte im Vordergrund möglichst exakt dargestellt werden, was den Einsatz der

`Raytracer`-Komponenten nahelegt. Die Ansprüche an die Darstellungsqualität der Hintergrundobjekte sind wesentlich weniger hoch. Der Einsatz einer weniger rechenintensiven und ungenaueren Methode wie das Z-Buffering (`ZBuffer`-Komponente) ist deshalb angemessen.

`Merge`
: Die beiden Renderer erzeugen zwei unterschiedliche 2D-Szenen, die durch die `Merge`-Komponente zusammengeführt werden.

`Rasterize`
: Die `Rasterize`-Komponente erfüllt in dieser Applikation eine wichtige Aufgabe, muss sie doch die Resultate der beiden Renderer in einer Pixmap vereinen. Dies geschieht aufgrund der Alpha- und Tiefenkanäle, die die von den Renderern erzeugten Pixmaps mitführen.

`Display`
: Für die abschliessende Präsentation des Resultates wird die `Display`-Komponente eingesetzt.

9.3.2 Konvexe Hülle (2D → 2D)

Die Berechnung der *Konvexen Hülle* einer endlichen Punktmenge in $\mathcal{R}^2$ ist eine typische Aufgabe der Algorithmischen Geometrie [PS85]. Die Lösung der Aufgabe gliedert sich in drei Schritte:

1. Einlesen der Punktmenge.
2. Berechnen der Konvexen Hülle, zum Beispiel mit dem Graham-Scan-Algorithmus [Gra72].
3. Darstellen der Konvexen Hülle.

In Abbildung 9.3 ist der mögliche Aufbau einer solchen Applikation unter Verwendung von BOOGA-Komponenten gezeigt. Insgesamt werden vier Komponenten eingesetzt:

`Parser2D`
: Die zu verarbeitende Punktmenge wird mit der `Parser2D`-Komponenten aus einer Datei eingelesen und in einem 2D-Szenengraphen als eine Menge von `Point`-Objekten abgelegt.

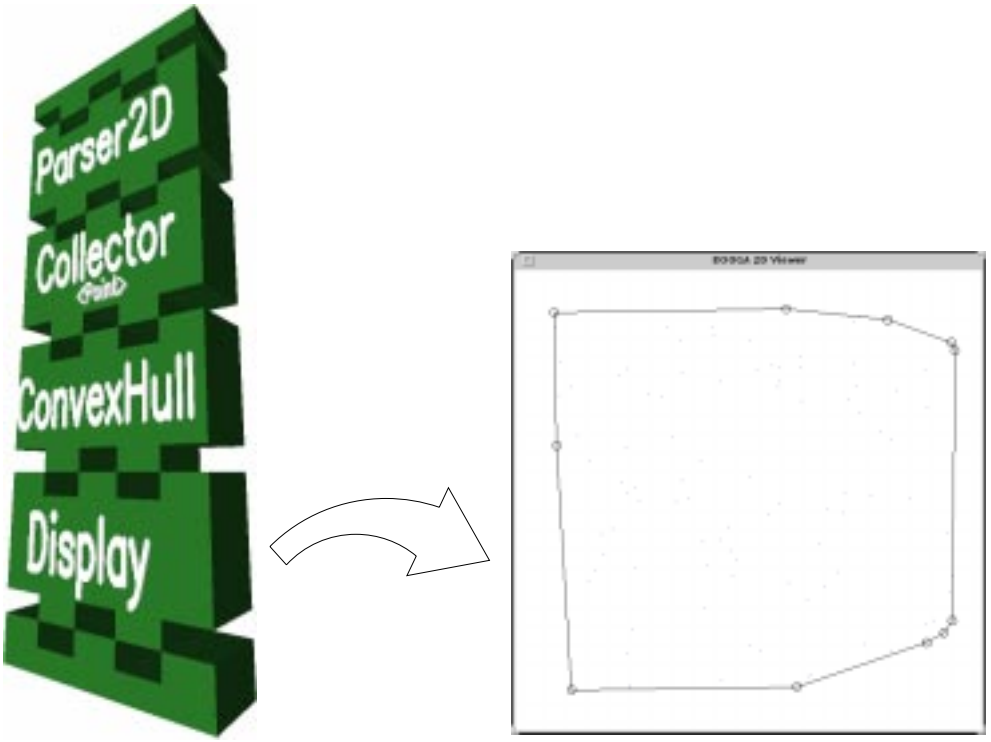


Abbildung 9.3
Die Berechnung der Konvexen Hülle einer endlichen Punktmenge.

`Collector`
: Da die Szene nicht ausschliesslich `Point`-Objekte enthalten muss, ermöglicht die `Collector`-Komponente das "Ausblenden" anderer Objekte. Sie erlaubt das Sammeln eines bestimmten Objekttyps. Die `Collector`-Komponente wird sehr häufig für die Realisierung anderer Komponenten verwendet. Zum Beispiel suchen alle Rendering-Verfahren in einem Vorverarbeitungsschritt die Szene nach Lichtquellen und Kameraobjekten ab.

`ConvexHull`
: In der `ConvexHull`-Komponenten ist der Algorithmus für die Berechnung der Konvexen Hülle implementiert. Damit das Resultat visualisiert werden kann, fügt die Komponente Strecken- und Kreisobjekte der Szene hinzu, die die Konvexe Hülle markieren.

`Display`
: Die abschliessende Darstellung erfolgt wiederum mit der `Display`-Komponenten. Alternativ könnte auch eine PostScript-Datei oder eine Pixmap erzeugt werden.

Dieselbe Applikation könnte natürlich leicht für den 3D-Fall modifiziert werden. 3D-Varianten der Komponenten `Parser2D`, `Collector` und `Display` existieren bereits. Lediglich ein Algorithmus für die Berechnung der Konvexen Hülle im 3D-Fall müsste als Komponente zusätzlich realisiert werden. Der Aufbau der Applikation bleibt sich dabei gleich.

9.3.3 Skizzen als Hilfsmittel für die 3D-Modellierung (2D → 3D)

Anwendungen, die anhand von 2D-Datenstrukturen 3D-Szenen erzeugen oder rekonstruieren, lassen sich leicht in die Architektur von Booga integrieren. Dieser Applikationstyp lässt sich hauptsächlich dem Bereich der Bildanalyse zuordnen. Das folgende Beispiel gehört sicherlich zu dieser Kategorie, verfolgt allerdings einen speziellen Ansatz. Die Idee für die Applikation basiert auf der Beobachtung, dass der Mensch anhand einer Entwurfsskizze, zum Beispiel für die Planung eines Einfamilienhauses und entsprechendem Kontextwissen, exakte Pläne erstellen kann. Es ist möglich, diese "mechanische" Übertragung dem Computer zu übertragen. Mehrere Arbeiten haben sich bereits mit dem Thema befasst [AHKS94, EBE95]. Sie sind allerdings immer noch sehr nahe bei den traditionellen Konstruktionsmethoden.

Unser Ansatz verwendet einen einfachen Editor für die Eingabe von Skizzen (siehe Abbildung 9.4). Der Anwender wird mit Hilfe einer Maus oder eines Zeichentabletts seine Skizzen erstellen. Jedem gezeichneten Objekt muss ein Typ zugeordnet werden. Der Beispieleditor bietet hierzu eine Menge von Buttons an, die jeweils einem bestimmten Objekttyp wie Berg, Haus oder Baum entsprechen. Vor dem Zeichen eines Objektes muss einer dieser Buttons gedrückt werden, wodurch dem System der gewünschte Objekttyp für alle nachfolgenden Zeichenoperationen mitgeteilt wird. Das Resultat der Eingabe ist eine Menge von Objekten, die sich aus einfachen Geradensegmenten zusammensetzen und mit einem Typ assoziiert

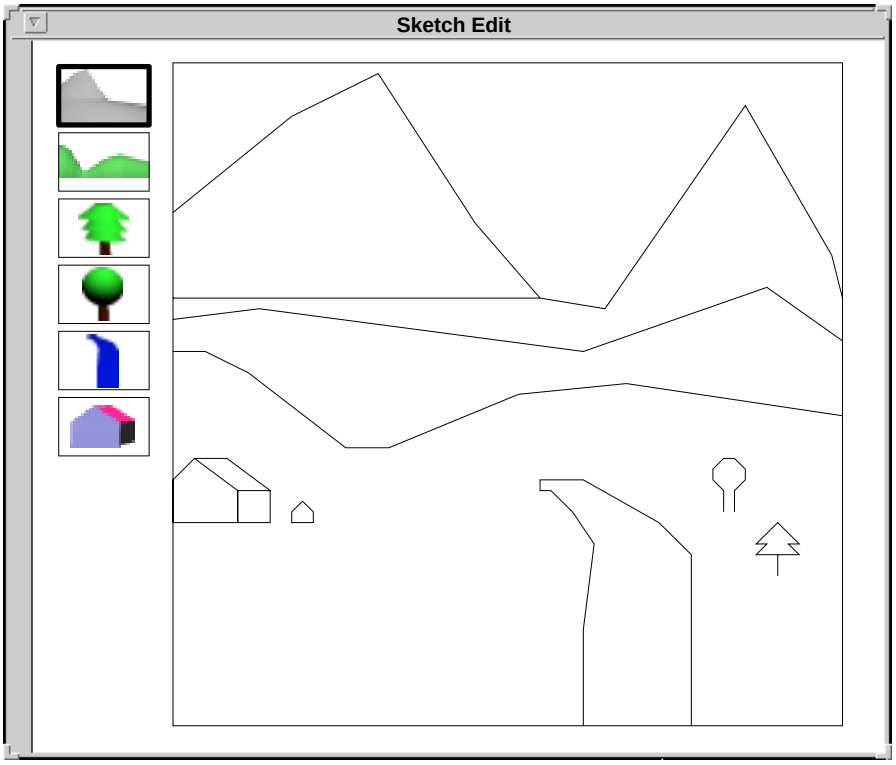


Abbildung 9.4 *Der einfache Editor zur Erzeugung von Skizzen. Jedem gezeichneten Objekt wird ein Objekttyp zugeordnet, ausgewählt durch einen der Buttons auf der linken Seite.*

sind. Anhand dieser Information wird nun eine 3D-Szene erzeugt, wie sie in Abbildung 9.5 zu sehen ist.

Eine mögliche Realisierung des diskutierten Szenarios mit Hilfe von BOOGA ist in Abbildung 9.6 visualisiert. Die Applikation befasst sich allerdings lediglich mit der Interpretation der Eingabeskizze sowie der Erzeugung der 3D-Szene und deren Darstellung. Der Skizzeneditor selbst ist nicht enthalten.

Die einzelnen Komponenten habe folgende Aufgaben:

`Parser2D`
: Die `Parser2D`-Komponente verarbeitet die vom Skizzeneditor stammende 2D-Szene. Die darin enthaltenen Objekte sind mit Typeninformation versehen.

`SketchInterpreter`
: Die eigentliche "Intelligenz" ist in der `SketchInterpreter`-Komponenten abgelegt. Hier ist das Wissen über die verschiedenen Objekttypen enthalten. Unter anderem sind dies Angaben zu Grössenverhältnissen und Methoden für die Generierung verschiedener Objekte. Aber auch Referenzen auf

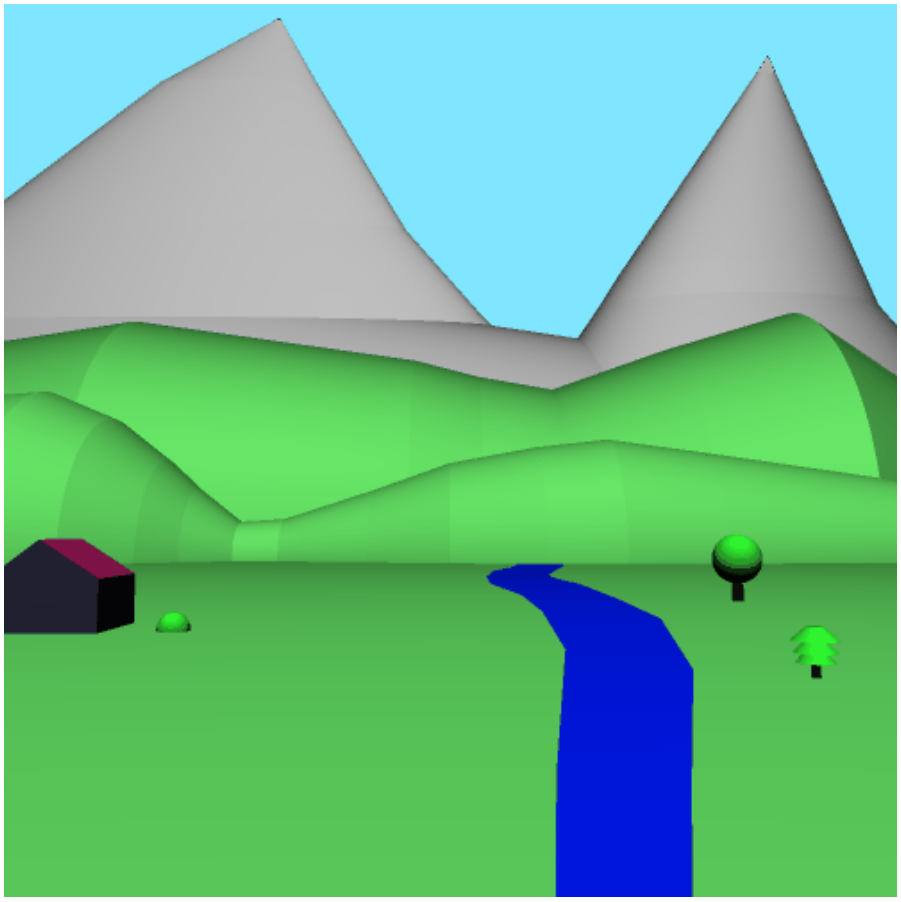


Abbildung 9.5
Anhand der Skizze in Abbildung 9.4 wird eine 3D-Szene erzeugt, die interaktiv begehbar ist.

vorbereitete geometrische Objekte aus einer Datenbank sind möglich. Diese Objekte können ohne Veränderung direkt in die 3D-Szene übernommen werden.

Die Komponente verarbeitet alle im 2D-Szenengraphen enthaltenen Objektskizzen, indem aus den Geradensegmenten objekttyp-spezifische Parameter extrahiert werden. Diese werden für die Generierung geeigneter 3D-Modelle oder zur Suche in der Objektdatenbank verwendet. Ist der gesamte 2D-Szenengraph verarbeitet, resultiert eine der Skizze entsprechende 3D-Szene.

`Raytracer`
: Die Beispielapplikation verwendet die `Raytracer`-Komponente um die Szene darzustellen. Diese nutzt die `intersect()`-Methode, um für jedes Pixel das jeweils sichtbare Objekt zu bestimmen. Mit Hilfe von `texturing()` wird

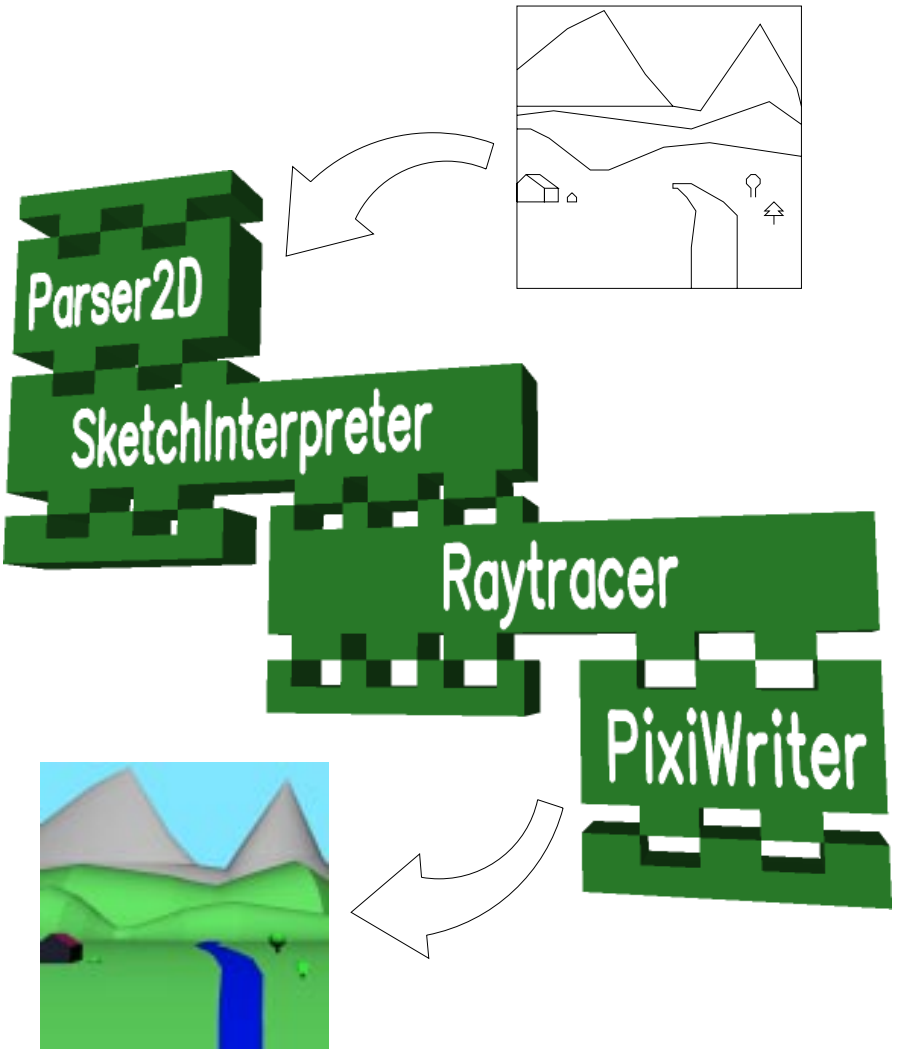


Abbildung 9.6
Die Sketch Applikation erzeugt aus Handskizzen 3D-Szenen, die mit Hilfe des Raytracing-Verfahrens dargestellt werden.

dann der zugehörige Farbwert berechnet und in einer Pixmap abgelegt.

`PixiWriter`
: Die `PixiWriter`-Komponente durchsucht einen 2D-Szenengraph nach Pixmap-Objekten und konvertiert sie in eine externe Repräsentation. In der Beispielapplikation findet sie lediglich die von der `Raytracer`-Komponenten erzeugte Pixmap. Sie wird in einer Datei zur Weiterverarbeitung gespeichert.

Als Variante könnte die Applikation die `Raytracer`-Komponente

durch eine Darstellungsmethode ersetzen, die eine interaktive Begehbarkeit der erzeugten 3D-Szene erlaubt.

Ein erster Prototyp wurde von Roland Balmer [Bal96] realisiert und lieferte bereits vielversprechende Resultate. Die Abbildungen sind auch teilweise der Arbeit von Balmer entnommen.

9.3.4 Lindenmayer-Systeme (3D → 3D)

Lindenmayer-System oder auch kurz L-Systeme wurden von Lindenmayer eingeführt [Lin68], um das Wachstum von lebenden Organismen zu analysieren. Sie sind speziell dazu geeignet, Pflanzen in ihrer Entwicklung formal zu beschreiben. Bei L-Systemen handelt es sich um generative Grammatiken, die ausgehend von einem Startsymbol und unter Anwendungen von Produktionen einen beliebig langen Symbolstring erzeugen. Dieser enthält Befehle, die eine Turtle im 3D-Raum manövrieren. Die Bewegungen der Turtle werden in grafische Primitive umgesetzt, wodurch ein geometrisches Modell entsteht. Eine Übersicht der wichtigsten Typen von L-Systemen und Hinweise für die Implementierung von L-System-Interpretern finden sich in [Str93b, SB94].

Die Integration von L-Systemen in BOOGA kann auf sehr unterschiedliche Weise erfolgen. Zum Beispiel könnte ein L-System als eigenständiger Objekttyp realisiert werden. Bevor nun auf die geometrische Struktur zugegriffen werden kann, muss das Objekt das L-System auswerten, um das zugehörige geometrische Modell zu erzeugen. Ein zweite Variante verwendet ebenfalls einen eigenen Objekttyp für die Repräsentation von L-Systemen. Die Interpretation wird nun allerdings in eine Komponente ausgelagert und ist nicht mehr Aufgabe des Objektes selbst.

Im folgenden wird der zweite Ansatz eingehender diskutiert. Abbildung 9.7 zeigt die Integration des angesprochenen Objekttyps in die BOOGA-Objekthierarchie. Die Klasse enthält keine Intelligenz für die Interpretation des L-Systems und kann somit die zugehörige geometrische Struktur nicht selbständig erzeugen. Auf diese Weise ist die Implementation der Klasse `LSystem` äusserst einfach. Das Vorgehen hat aber den Nachteil, dass es die Aufgabe der Applikation ist, die L-Systeme zu interpretieren und die geometrische Struktur zu erzeugen. Wird dieser Schritt vergessen, werden im Szenengraph enthaltenen L-Systeme nicht berücksichtigt. Eine ent-

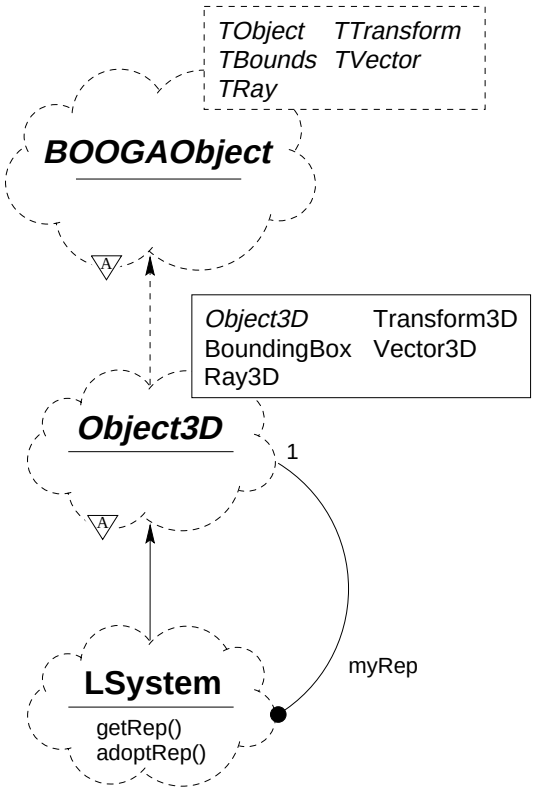


Abbildung 9.7
Ein L-System wird durch eine eigene Klasse repräsentiert.

sprechend angepasste Wireframe-Applikation ist in Abbildung 9.8 zu sehen.

Die in der Applikation verwendete `LSystem`-Komponente ist verantwortlich für die Verarbeitung von L-Systemen. Die Aufgabe wird in zwei Schritten ausgeführt:

1. Der Szenengraph wird nach Objekten vom Typ `LSystem` abgesucht. Dafür kann zum Beispiel die `Collector`-Komponente eingesetzt werden.

2. Alle gefunden L-Systeme werden interpretiert und das zugehörige geometrische Modell erzeugt. Die Instanzen der Klasse `LSystem` besitzen ein Datenelement (`myRep`, siehe Abbildung 9.7), das dieses Modell speichern kann und für nachfolgende Operationen zur Verfügung stellt. Die Methode `adoptRep()` und `getRep()` werden hierzu verwendet.

Die Integration von L-Systemen ist ein typisches Beispiel für die Applikationsentwicklung mit BOOGA. Viele Anwendungen benötigen sowohl neue Objekttypen wie spezifische Komponenten. Meist

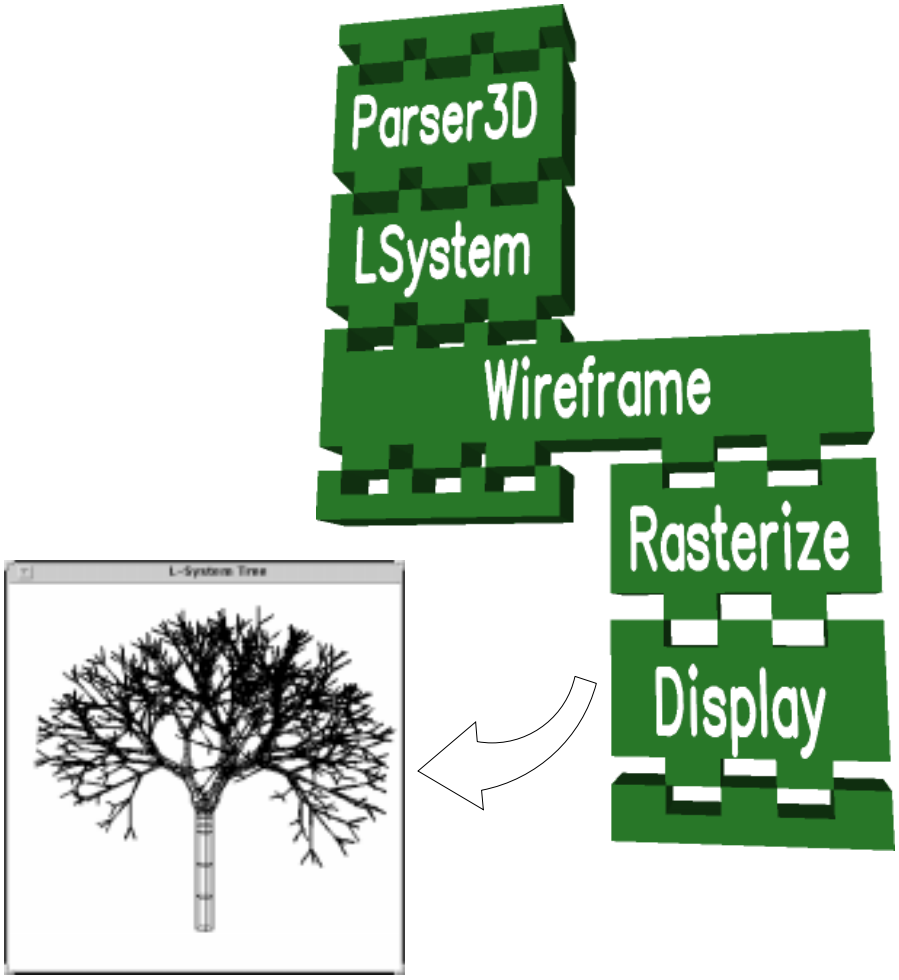


Abbildung 9.8
Die Wireframe-Applikation ergänzt um eine Komponente zur Verarbeitung von L-Systemen.

besteht dabei die Möglichkeit, wie im Beispiel der L-Systeme, die Verarbeitungslogik innerhalb der Objekte oder als eigenständige Komponenten zu realisieren. Welche Variante gewählt wird ist stark vom jeweiligen Anwendungsfall abhängig.

Mit diesen Ausführungen ist unsere Präsentation von BOOGA abgeschlossen. Zusätzliche Beispiele und die Besprechung weiterer Abstraktionen und Mechanismen, sind vor allem in der Arbeit von Stephan Amann [Ama97] zu finden. Aber auch in [Amm96], [Bä95], [Hab96], [Mat96], [Lie96], [Sag96], [vSW96] und [Teu96] werden interessante Anwendungsbereiche und Erweiterungen von BOOGA präsentiert.

10 Schlussfolgerungen und Ausblick

Dieses Kapitel fasst die wichtigsten Schlussfolgerungen und Erfahrungen aus der Arbeit an BOOGA zusammen. Mögliche zukünftige Stossrichtungen werden im letzten Abschnitt vorgeschlagen und Verbesserungsvorschläge präsentiert.

10.1 Erfüllung der Kriterien

BOOGA ist sicherlich ein erfolgreiches Frameworkprojekt. Die von uns in Kapitel 1 aufgestellten Anforderungen konnten zur vollen Zufriedenheit erfüllt werden, wie die folgende Auflistung zeigt:

- *Wiederverwendbarkeit und Erweiterbarkeit*
  Die Wiederverwendbarkeit der zugrundeliegenden Designstrukturen, Abstraktionen und Komponenten konnte in vielen Anwendungen gezeigt werden. Das Konzept des Komponentenframeworks hat sich bewährt. Die erhofften Merkmale — gute Wiederverwendbarkeit, flexible Erweiterbarkeit und die leichte Erlernbarkeit durch einen stufenweisen Zugang zur Funktionalität des Systems — konnten durch mehrere Anwender bestätigt werden.

- *Unterstützung verschiedener Teilgebiete der Computergrafik*
  Die guten Eigenschaften von BOOGA lassen sich nicht ausschliesslich mit der gewählten Architektur erklären. Einen beträchtlichen Anteil besitzt auch die der Architektur zugrundeliegende Klassifizierung des Computergrafikbereichs, die von der Unterscheidung von 2D- und 3D-Welten ausgeht (siehe auch Abschnitt 5.1 ab Seite 61). Die Eignung des daraus abgeleiteten Konzeptes wird wohl auch durch die Tatsache bewiesen, dass es in den drei Jahren der Entwicklung von BOOGA nie an neue Anforderungen angepasst werden musste. Dieses Konzept erlaubte auch die umfassende Unterstützung der gewünschten Teilgebiete der Computergrafik, was BOOGA im Vergleich zu anderen Grafiksystemen einzigartig macht.

- *Minimierung des Einarbeitungsaufwandes*
  Mit Hilfe von BOOGA konnten bereits einige Projekte mit Erfolg abgeschlossen werden. Besonders Beat Liechti [Lie96] und Peter Sagara [Sag96] müssen in diesem Zusammenhang erwähnt werden, da sie ihre Arbeiten innerhalb zweier Wochen beenden mussten. In diesem Zeitraum ist sowohl die Einarbeitungsphase wie auch die eigentliche Bearbeitung der Problemstellung enthalten. Diese Beispiele, aber auch die Erfahrung mit anderen Projekten, belegen den geringen Zeitaufwand, der für den Einstieg in BOOGA benötigt wird. Dies führen wir wiederum vor allem auf das Konzept des Komponentenframeworks zurück.

10.2 Beurteilung des Vorgehens

Während der Entwicklung von BOOGA konnten wir viele Erfahrungen sammeln, sowohl positive wie negative. Die eigentliche Realisierung lässt sich in zwei Phasen gliedern. Zu Beginn standen konzeptionelle Überlegungen und die Implementierung der initialen Version der BOOGA-Architektur im Vordergrund. Nach kaum einjähriger "Vorlaufzeit" stiessen die ersten Anwender zum Projekt. Dies erlaubte uns eine frühe Überprüfung der Konzepte und offenbarte viele Mängel und Einschränkungen. Die sehr frühe Miteinbeziehung von Anwendern in den Entwicklungsprozess, ist im nachhinein als sehr positiv zu bewerten. Wir sind überzeugt, dass sich ein Framework sehr rasch in realen Anwendungsentwicklungen bewähren

muss. Einige weitere positive Erfahrungen sind in der folgenden Liste enthalten:

- *Grafikvorkenntnisse*
  Eine Frameworkentwicklung kann nur dann erfolgreich sein, falls ausgezeichnete Vorkenntnisse des Problembereichs vorhanden sind. In unserem Fall war diese Bedingung zweifellos erfüllt. Durch mehrere Vorlesungen wurden die verschiedenen Teilgebiete der Computergrafik abgedeckt. Das so erhaltene Wissen konnte zudem durch unseren Projekt- und Diplomarbeiten vertieft werden. Ohne diese Vorkenntnisse wäre es sicherlich nicht möglich gewesen, ein solch umfassendes Grafiksystem wie BOOGA planen und realisieren zu können.

- *Entwicklungsumgebung*
  Die Arbeit an und mit BOOGA wurde durch eine leistungsfähige Entwicklungsumgebung vereinfacht. Diese wurde mit Hilfe von SNiFF+ realisiert und unterstützt eine vollständige Versions- und Konfigurationskontrolle aller Entwicklungsergebnisse.

  Bereits zu Beginn der BOOGA-Projektes wurde zwischen zwei Typen von Entwicklern unterschieden: *Core-* und *Application-Developer*. Ein Core-Developer hat den vollen Zugriff auf die BOOGA-Sourcen und kann beliebige Änderungen vornehmen. Application-Developer hingegen arbeiten mit einer vom Team der Core-Developer freigegebenen Version des Frameworks, auf die sie lediglich lesenden Zugriff erhalten. Erweiterungen an BOOGA können Application-Developer also nicht direkt vornehmen, sondern nur über ein Mitglied aus dem Core-Team veranlassen. Auf diese Weise kann eine Weiterentwicklung der Basisabstraktionen und Mechanismen von BOOGA erfolgen, ohne dass die Applikationsentwicklung davon betroffen ist. In regelmässigen Zeitabstanden werden neue Versionen des Frameworks vom Core-Team freigegeben, was gegebenenfalls eine Anpassung bestehender Applikationen bedingt.

  Rückblickend ist es auch dem Aufbau der Entwicklungsumgebung zu verdanken, dass Projekte mit kurzen Laufzeiten erfolgreich abgeschlossen werden konnten. Ein Projektbearbeiter musste durch die Entkopplung von Framework- und Applikationsentwicklung nicht zwingend die laufenden Ände-

rungen berücksichtigen, sondern konnte sich besser auf das eigentliche Problem konzentrieren.

- *Beispielapplikationen als Dokumentationshilfen*
  Für die Dokumentation der grundlegenden Mechanismen und Abstraktionen wurden einfache Beispielapplikationen entworfen. Diese demonstrieren jeweils ausgewählte Konzepte von BOOGA. Der Einstieg in das Framework erfolgt dann über das Studium und die Erweiterung dieser Applikationen. Diese Vorgehensweise wurde auch schon bei ET++ [WGM88] mit Erfolg eingesetzt. Es ist aber zu beachten, dass diese Art der Dokumentation nicht alle Bedürfnisse abzudecken vermag. Ein herkömmliche Beschreibung der Architektur und der zentralen Mechanismen ist weiterhin unabdingbar.

- *C++*
  Die Programmiersprache C++ hat sich als gute Wahl für die Implementierung von BOOGA erwiesen. Die oft bemängelte schlechte Portierbarkeit von C++ Code auf andere Plattformen mit unterschiedlichen Compilern konnte nicht bestätigt werden. Ein Nachteil bleiben aber die langen Turn-Around-Zeiten, die aber durch das gute Laufzeitverhalten des erzeugten Code relativiert werden.

Wo Licht ist, ist natürlich auch Schatten. So haben wir bei der Entwicklung von BOOGA die Dokumentation, ein zentrales Thema aller Entwicklungsprojekte, vernachlässigt. Gute Code-Dokumentation und die oben erwähnten einfachen Beispielanwendungen vermögen viele Fragen eines Anwenders zu beantworten, sind aber keinenfalls ausreichend. Vielmehr ist ein umfassendes *Dokumentationskonzept* gefordert. Dabei ist von besonderer Wichtigkeit, dass die Dokumentation mit der Entwicklung Schritt halten kann. Die Struktur der Dokumentation muss zudem an die Software-Architektur angepasst sein und diese sogar widerspiegeln. Einige Gedanken und Vorschläge für ein Dokumentationskonzept für BOOGA sind in der Arbeit von Amann [Ama97] enthalten. Ohne hier auf Einzelheiten eingehen zu wollen, kann aber ein Schluss gezogen werden:

> *Konzepte für die Software-Architektur und die Dokumentation haben die gleiche Wichtigkeit und sollten auch gleichzeitig erarbeitet werden.*

Diese Forderung darf nicht überraschen, ist doch eine gute und angepasste Dokumentation eine Voraussetzung für den langfristigen Erfolg eines Frameworks. Bei einem Neubeginn würden wir deshalb diesem Punkt unbedingt das nötige Gewicht beimessen!

Der nächste Abschnitt gibt nun einen Ausblick auf mögliche Erweiterungen von BOOGA.

10.3 Ausblick

Eine Frameworkentwicklung ist niemals abgeschlossen. Auch BOOGA muss sich weiterentwickeln, um für zukünftige Anforderungen gerüstet zu sein. Das Framework hat bereits einige Entwicklungszyklen hinter sich gebracht. Die Objekthierarchie der Frameworkschicht ist ausgereift und das Komponentenprinzip hat sich in vielen Anwendungen bewährt. Einige Abstraktionen besitzen allerdings zur Zeit lediglich Prototyp-Charakter, wie zum Beispiel die Behandlung von Lichtquellen. Wichtige Verfahren, unter anderem Radiosity [GTGB84], werden zudem überhaupt noch nicht unterstützt. Es besteht also nach wie vor der Bedarf für Erweiterungen.

Neben der Erweiterung der Basismechanismen und -abstraktionen sollte aber auch die Entwicklung von interessanten Applikationen vorangetrieben werden. Anwendungen aus den Gebieten Animation, Algorithmische Geometrie und Bildanalyse sind zur Zeit noch untervertreten. Sie könnten die zugrundeliegenden Konzepte weiter überprüfen und die Anwendbarkeit von BOOGA an neuen Problemstellungen demonstrieren.

Es ist zu hoffen, das BOOGA am Institut für Informatik und angewandte Mathematik weiterhin im Einsatz bleibt. Vor allem die Verwendung in der Grundausbildung wäre erstrebenswert. Denn falls BOOGA nicht mehr verwendet, gepflegt und weiterentwickelt wird und sich auch keine Alternativen anbieten, könnte die in der Einleitung gemachte Bemerkung

> *"In der Gruppe für Computergeometrie und Grafik am Institut für Informatik und angewandte Mathematik der Universität Bern stieg der Bedarf für ein Grundsystem, das die vielfältigen Forschungsarbeiten im Bereich der Computergrafik und Bildverarbeitung unterstützen kann. Bereits erstellte Software konnte in vielen Fällen nicht*

> *als Grundlage für Weiterentwicklungen dienen, da unter anderem keine gemeinsamen Klassenbibliotheken verwendet wurden oder der Aufwand für eine Codeintegration viel zu hoch gewesen wäre. Die meisten Realisierungen kamen deshalb Neuentwicklungen gleich."*

bald wieder Gültigkeit haben.

Literaturverzeichnis

[AE93] P. Ackermann and D. Eichelberg. Interactive 3D Graphics in ET++. In *Technology of Object-oriented Languages and Systems (TOOLS)*, 1993.

[AHKS94] M. Akeo, H. Hashimoto, T. Kobayashi, and T. Shibusawa. Computer Graphics System for Reproducing Three-dimensional Shape from Idea Sketch. In *Computer Graphics Forum*, pages 477–488. Eurographics '94, 1994.

[Ama93] St. Amann. RadShade*: Implementation eines globalen Beleuchtungsmodells basierend auf Radiosity und Raytracing.* Master's thesis, IAM, Universität Bern, 1993.

[Ama97] St. Amann. *Komponentenorienterte Entwicklung von Grafikapplikationen mit* Booga. PhD thesis, Universität Bern, 1997.

[Amm96] L. Ammon. *Magische Bilder entzaubert – SIRDS-Algorithmen im Vergleich.* Informatikprojekt IAM-FCG-9602, Universität Bern, 1996.

[Arc92] Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1.* Addison-Wesley, 1992.

[ASB96] St. Amann, Ch. Streit, and H. Bieri. Booga: *A Component-Oriented Framework for Computer Graphics.* IAM, Universität Bern, 1996.

[AW87] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In G. Marechal, editor, *Eurographics '87*, pages 3–10, August 1987.

[Bä95] R. Bächler. *Entwurf und Implementierung einer NURBS-Library.* Master's thesis, IAM, Universität Bern, 1995.

[Bal96] R. Balmer. *Sketching – Generierung von Szeneninformation aus einer Skizze.* Informatikprojekt IAM-FCG-9604, Universität Bern, 1996.

[BB95] E. Beier and U. Bozetti. *A Generic Graphics Kernel and a Customized Derivative.* Submitted to 6th Euro-Graphics Workshop on Rendering, 1995.

[Bei94] E. Beier. *Objektorientierte 3D-Grafik: Grundlagen. Konzepte and praktische Realisierungen.* International Thomson Publishing GmbH, 1994.

[Bei96] E. Beier. A Generic Approach to Computer Graphics. In *Visualization and Mathematics.* Springer-Verlag, 1996.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley, 1996.

[BN76] J. F. Blinn and M. E. Newell. Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19(10):542–546, October 1976.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 2nd edition, 1994.

[Boo96a] G. Booch. *Object Solutions: Managing the Object-Oriented Project.* Addison-Wesley, 1996.

[Boo96b] G. Booch. The Booch Method: Patterns and Protocols. *ROAD (Report on Object Analysis & Design)*, 5, 1996.

[BPP96] G. Bell, A. Parisi, and M. Pesce. *The Virtual Reality Modeling Language: Version 1.0C Specification.* `http://vag.vrml.org/vrml10c.html`, 1996.

[BW89] E. H. Blake and P. Wisskirchen. Object-oriented graphics. In W. Purgathofer and J. Schonhut, editors, *Advances in Computer Graphics V*, pages 109–154. Springer-Verlag, 1989.

[BW93] A. Bowyer and J. Woodwark. *Introduction to Computing with Geometry*. Information Geometers, 1993.

[Cat79] E. Catmull. A tutorial on compensation tables. In *Computer Graphics (SIGGRAPH '79 Proceedings)*, pages 1–7, August 1979.

[CG94] L. Christov and M. Gorzelanczyk. *3D–Graphik mit HOOPS*. Springer-Verlag, 1994.

[CGG+88] G. Champine, J. Gettys, G. Grinstein, B. Herzog, and R. Scheifler. X Window System. In R. J. Beach, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, pages 349–348. ACM Press, August 1988.

[Col90] A. Collison. *Effizienzsteigerung von Ray Tracer*. Informatikprojekt IAM-PR-90355, Universität Bern, 1990.

[Cop92] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[DDtHR94] D. A. Duce, D. J. Duke, P. J.W. ten Hagen, and G. J. Reynolds. PREMO - An Initial Approach to a Formal Definition. In *Computer Graphics Forum*, pages 393–406. Eurographics, 1994.

[EBE95] L. Eggli, B. Brüderlin, and G. Elber. Sketching as a Solid Modeling Tool. In C. Hoffmann and J. Rossignac, editors, *Proceedings of the 3rd Symposium on Solid Modeling and Applications (SSMA'95)*, pages 313–322. ACM Press, May 1995.

[Egb92] P. K. Egbert. *An Object-Oriented Approach to Graphical Application Support*. PhD thesis, Department of Computer Science, University of Illinois, June 1992.

[Egb95] P. K. Egbert. Utilizing Renderer Efficiencies in an Object-Oriented Graphics System. In R. C. Veltkamp and E. H. Blake, editors, *Programming Paradigms in Graphics: Proceedings of the Eurographics Workshop*. Springer-Verlag, 1995.

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Rerenence Manual*. Addison–Wesley, 1990.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 2nd edition, 1990.

[Gam91] E. Gamma. *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools.* PhD thesis, Institut für Informatik, Universität Zürich, 1991.

[Gam96] E. Gamma. Design Patterns, Frameworks, Components – Elements of modern Application Architectures. In *Components User's Conference '96 (CUC)*, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gra72] R. L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Inf. Process. Lett.*, 1:132–133, 1972.

[GTGB84] C. M. Goral, K. K. Torrance, D. P. Greenberg, and B. Battaile. Modelling the Interaction of Light Between Diffuse Surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 213–222, July 1984.

[GW96] E. Gamma and A. Weinand. *Mythos der objektorientierten Programmierung.* Seminar IFA Zürich, 1996.

[Hab96] P. Habegger. *Ein grafischer Strukturbrowser.* Master's thesis, IAM, Universität Bern, 1996.

[Hec89] P. S. Heckbert. Writing a ray tracer. In A. S. Glassner, editor, *An introduction to ray tracing*, pages 263–293. Academic Press, 1989.

[HL90] P. Hanrahan and J. Lawson. A Language for Shading and Lighting Calculations. *Computer Graphics*, 24(4):289–298, August 1990.

[Hun78] G. M. Hunter. *Efficient computation and data structures for graphics.* PhD thesis, Princeton University, 1978.

[ISO85] ISO. Specification for a set of functions for computer graphics programming, the Graphical Kernel System (GKS) [ISO 7942:1985], 1985.

[ISO89] ISO. Information processing systems – Computer Graphics – Programmers Hierarchical Interactive Graphics System (PHIGS) - Part 1: 1989 Functional description [ISO/IEC 9592–1:1989], 1989.

[ISO92] ISO. A Reference Model for Computer Graphics [ISO/IEC 11072:1992], 1992.

[ISO9x] ISO. Information processing systems – Computer graphics and image processing – Presentation Environments for Multimedia Objects (PREMO) [ISO/IEC 14478–1], 199x.

[JF88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming (JOOP)*, June/July 1988.

[Joh96] R. E. Johnson. *Frameworks*. Seminar Zühlke Informatik, Zürich, Mai 1996.

[KA88] D. Kirk and J. Arvo. The Ray Tracing Kernel. In *Proceedings of Ausgraph '88*, pages 75–82, 1988.

[Kil94] M. J. Kilgard. OpenGL and X: An OpenGL Toolkit. *The X Journal*, November/December 1994.

[KK86] T. L. Kay and J. T. Kajiya. Ray Tracing Complex Scenes. In D. C. Evans and R. J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 269–278, August 1986.

[KR94] K. Konstantinides and J. R. Rasure. The KHOROS software development environment for image and signal processing. *IEEE Transactions on Image Processing*, 3(3):243–52, 1994.

[Kru92] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[KW93] L. Koved and W. L. Wooten. GROOP: An object-oriented toolkit for animated 3D graphics. In *OOPSLA'93*, pages 309–325, 1993.

[LBdMP95] C. Laffra, E. H. Blake, V. de Mey, and X. Pintado. *Object-Oriented Programming for Graphics.* Springer-Verlag, 1995.

[Lie96] B. M. Liechti. *Virtual Reality Modeling Language Facility für Berne's Object-Orinented Graphics Architecture.* Diplomarbeit, Ingenieurschule Bern HTL, 1996.

[Lin68] A. Lindenmayer. Mathematical Models for Cellular Interactions in Development, I & II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[Mat96] T. Matthey. *Computer-Animation für* BOOGA. Informatikprojekt IAM-FCG-9601, Universität Bern, 1996.

[McI69] M. D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering.* NATO Science Committee, 1969.

[Mei86] A. Meier. *Methoden der grafischen und geometrischen Datenverarbeitung.* Teubner, 1986.

[Met95] I. Metz. *Bintree Lab: Ein Framework von Datenstrukturen und Algorithmen für Bintrees.* PhD thesis, Universität Bern, 1995.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[Mey96] S. Meyers. *More Effective C++.* Addison-Wesley, 1996.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[ND81] K. Nygaard and O.-J. Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages.* ACM Press, 1981.

[NDW93] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1.* Addison-Wesley, 1993.

[Nor96] M. E. Nordberg. Variations on the Visitor Pattern. In *PLoP'96*, 1996.

[NR72] F. Nake and A. Rosenfeld, editors. *Graphic Language. Proceedings of the IFIP Working Conference on Graphic Laguages*. North-Holland, 1972.

[OMG95] OMG. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, 1995.

[Ope94] Open Inventor Architecture Group. *Open Inventor C++ Reference Manual: The Official Reference Document for Open Systems*. Addison-Wesley, 1994.

[Pav82] T. Pavlidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, 1982.

[Per85] K. Perlin. An Image Synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 287–296, July 1985.

[Pho75] B. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[Pre95] W. Pree. *Design Patterns for Object-Oriented Development*. Addison-Wesley, 1995.

[Pro95] ITTI Gravigs Project. *Standards for Computer Graphics*. University of Manchester `http://info.mcc.ac.uk/CGU/ITTI/Stds/-standards_announce.html`, 1995.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[RR78] D. R. Reddy and S. M. Rubin. *Representation of Three-Dimensional Objects*. Technical Report CMU-CS-78–113, Department of Computer Science, Carnegie-Mellon University, April 1978.

[Sag96] P. S. Sagara. BOOGA*'s new House*. Diplomarbeit, Ingenieurschule Bern HTL, 1996.

[SB94] Ch. Streit and H. Bieri. Interactive Construction of L-Systems in 2- and 3-Space. In *SIBGRAPI '94 Proceedings*, 1994.

[SC92] P. S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. In E. E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 341–349, July 1992.

[SGFR90] R. W. Scheifler, J. Gettys, J. Flowers, and D. Rosenthal. *X Window System: The Complete Reference to Xlib, X Protocol.* Digital Press, 2nd edition, 1990.

[Som89] J. Sommerville. *Software Engineering*, quoted Lehman and Belady. Addison-Wesley, 3rd edition, 1989.

[SS95] P. Slusallek and H. Seidel. VISION – An Architecture for Global Illumination Calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77–96, March 1995.

[SSB91] P. Shirley, K. Sung, and W. Brown. A Ray Tracing Framework for Global Illumination Systems. In *Proceedings of Graphics Interface '91*, pages 117–128, June 1991.

[Str93a] P. S. Strauss. IRIS Inventor, A 3D Graphics Toolkit. In A. Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 192–200. ACM Press, 1993.

[Str93b] Ch. Streit. *Modellierung mit Lindenmayer-Systemen in der Computergrafik.* Master's thesis, IAM, Universität Bern, 1993.

[Str95] B. Stroustrup. *The C++ Programming Language Second Edition.* Addison-Wesley, 1995.

[Tal94] Taligent. *Leveraging Object-Oriented Frameworks.* `http://www.taligent.com/leveraging-oofw.html`, 1994.

[Teu96] T. Teuscher. *Shading Languages.* Master's thesis, IAM, Universität Bern, 1996.

[Ups90] S. Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley, 1990.

[vSW96] T. von Siebenthal and T. Wenger. *Anfertigen von Klassendiagrammen aus Sourcecode.* IAM-FCG-9603, Universität Bern, 1996.

[Wei95] A. Weinand. *If Components are the Solution, what is the Problem?* CHOOSE SIG BEER, 1995.

[Wei96] A. Weinand. Components: Another end to the software crisis? In *Components User Conference (CUC)*, 1996.

[Wer94a] J. Wernecke. *The Inventor Toolmaker.* Addison-Wesley, 1994.

[Wer94b] J. Wernecke. *The Iventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor.* Addison-Wesley, 1994.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ – An object-oriented Application Framework in C++. In *OOPSLA'88 Conference Proceedings*, 1988.

[Whi80] T. Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, June 1980.

[Wis95] P. Wisskirchen. The PREMO Framework: Object-Oriented Issues. In C. Laffra, E. H. Blake, V. de Mey, and X. Pintado, editors, *Object-Oriented Programming for Graphics*, pages 204–213. Springer-Verlag, 1995.

[WK90] P. Wisskirchen and K. Kansy. The New Graphics Standard – Object-Oriented! In E. H. Blake and P. Wisskirchen, editors, *Eurographics Workshop on Object-Oriented Graphics*, pages 199–216. Springer-Verlag, 1990.

[YAK95] M. Young, D. Argiro, and S. Kubica. Cantata: Visual Programming Environment for the KHOROS System. *Computer Graphics*, 29(2):22–33, May 1995.

Grafische Notation für BOOGA-Applikationen

A

Dieses Kapitel beschreibt die grafische Notation, die für die Visualisierung von BOOGA-Applikationen verwendet wird. Beispiele für deren Anwendung, sind in Kapitel 9 zu finden. Die Notation wurde mit dem Ziel entwickelt, den Aufbau von BOOGA-Applikationen auf einfache Art und Weise zu illustrieren. Sie ist besonders für die Darstellung von Applikationen mit einem deterministischen Verhalten geeignet. Anwendungen, deren Ablauf durch den Benutzer in weiten Grenzen beeinflusst werden können, eignen sich hierfür wesentlich weniger gut, da keine Symbole für die Darstellung von Verzweigungen und Iterationen existieren.

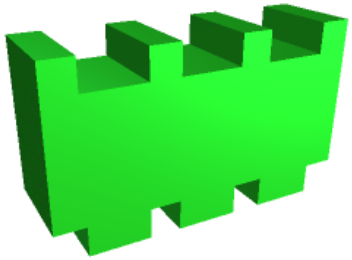
Jeder Komponententyp wird durch ein eigenes Symbol repräsentiert. Die Dimension der Eingabewelt wird durch die Anzahl von Vertiefungen dargestellt und diejenige des Resultates mit der entsprechenden Zahl von Erhebungen. Zum Beispiel besitzt der Komponententyp `Operation3DTo2D` auf der oberen Seite drei Vertiefungen und auf der unteren zwei Erhebungen (siehe weiter unten). Dadurch werden Objekte vom Typ `World3D` und `World2D` als Eingabe- bzw. Ausgabedatenstrukturen repräsentiert. Folgende Symbole stehen zur Verfügung:

`Operation3DTo3D`

Die `Operation3DTo3D`-Komponente verarbeitet 3D-Welten. Das Resultat ist die transformierte Eingabe und/oder ein neu erzeugtes

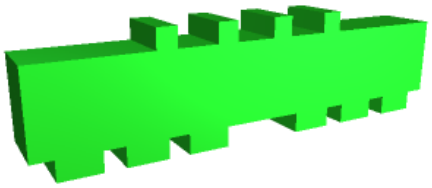
Objekt vom Typ `World3D`. Beispiele für solche Komponententypen sind Parser oder Editoren.

Eingabewelt : 3D
Resultatwelt(en) : 3D



Die Eingabewelt durchläuft die Komponente und wird von dieser verarbeitet. Im Verarbeitungsschritt hat die Komponente lesenden und schreibenden Zugriff auf die Elemente der Eingabewelt.

Eingabewelt : 3D
Resultatwelt(en) : 2 × 3D

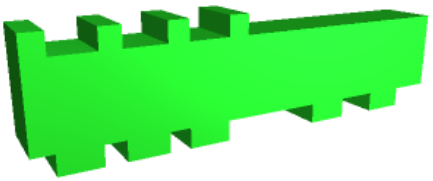


Eine neue 3D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die Eingabewelt.

Operation3DTo2D

Die `Operation3DTo2D`-Komponente erzeugt eine 2D-Welt basierend auf einer 3D-Eingabe. Beispiele für diesen Komponententyp sind die verschiedenen Rendering-Verfahren.

Eingabewelt : 3D
Resultatwelt(en) : 3D+2D



Eine 2D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die 3D-Eingabewelt.

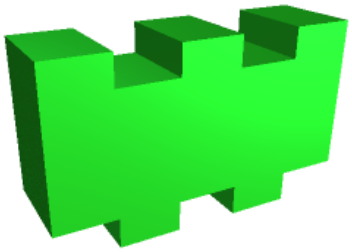
Operation2DTo2D

Die `Operation2DTo2D`-Komponente verarbeitet 2D-Welten. Das Resultat ist die transformierte Eingabe und/oder ein neu erzeugtes

Objekt vom Typ `World2D`. Beispiele für solche Komponententypen sind Bildbearbeitungsfunktionen oder Editoroperationen.

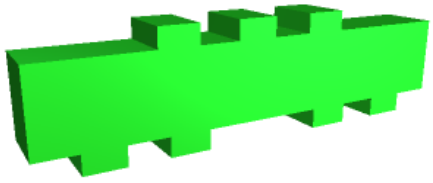
Eingabewelt : 2D
Resultatwelt(en) : 2D

Die Eingabewelt durchläuft die Komponente und wird von dieser verarbeitet. Im Verarbeitungsschritt hat die Komponente lesenden und schreibenden Zugriff auf die Elemente der Eingabewelt.



Eingabewelt : 2D
Resultatwelt(en) : 2 × 2D

Eine neue 2D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die Eingabewelt.

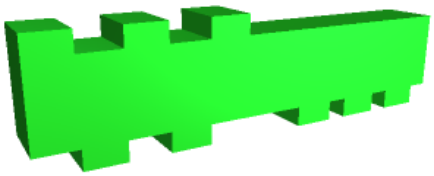


`Operation2DTo3D`

Die `Operation2DTo3D`-Komponente erzeugt eine 3D-Welt basierend auf einer 2D-Eingabe. Beispiele für diesen Komponententyp sind die verschiedenen Verfahren der Bildanalyse, die aus 2D-Datenstrukturen 3D-Modell rekonstruieren.

Eingabewelt : 2D
Resultatwelt(en) : 2D+3D

Eine 3D-Welt wird erzeugt. Die Komponente hat aber auch lesenden und schreibenden Zugriff auf die 2D-Eingabewelt.



Creation

Die beiden folgenden Symbole symbolisieren die Erzeugung leerer 2D- bez. 3D-Welten (Objekte vom Typ `World2D` und `World3D`).

Eingabewelt : —
Resultatwelt(en) : 3D

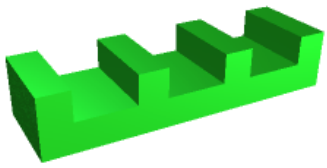
. . . . . .

Eingabewelt : —
Resultatwelt(en) : 2D

Deletion

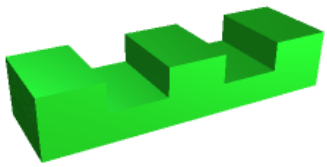
Die beiden letzten Symbole werden für die Entfernung von zuvor erzeugten Weltobjekten verwendet. Diese entstehen durch eine *Creation*-Operation oder bei der Aktivierung einer Komponenten.

Eingabewelt : 3D
Resultatwelt(en) : —



. . . . . .

Eingabewelt : 2D
Resultatwelt(en) : —



BOOGA Scene Description Language

B

BOOGA besitzt eine eigene Szenenbeschreibungssprache (BSDL, BOOGA Scene Description Language), die sowohl für den 2D- wie für den 3D-Fall zum Einsatz kommt. Die Sprache zeichnet sich durch folgende Eigenschaften aus:

- *Wenige Schlüsselwörter*

  Im Unterschied zu vergleichbaren Sprachen (vgl. zum Beispiel RAYSHADE) kennt BSDL lediglich die vier Schlüsselwörter `define`, `const`, `using` und `namespace`. Dadurch hat die Sprache eine sehr einfache Struktur (siehe auch den nächsten Abschnitt).

- *Dynamische Konfiguration*

  Der in BOOGA enthaltene Parser für BSDL lässt sich dynamisch, d.h. zur Laufzeit konfigurieren (Komponenten `Parser2D` und `Parser3D`). Erst danach ist er beispielsweise in der Lage, die Definition einer Kugel zu verarbeiten. Das zugehörige Schlüsselwort (`sphere` in unserem Beispiel) ist vor dem Konfigurationsprozess nicht im Sprachumfang von BSDL enthalten. Auch mathematische Funktionen werden auf diese Weise dem Parser bekanntgegeben.

  Das Konzept erlaubt die beliebige Erweiterung des Sprachumfangs, was eine Grundbedingung für einen Parser in einem Framework darstellt.

- *Mathematische Ausdrücke*

  BSDL kann einfache arithmetische Ausdrücke verarbeiten, die sich aus den vier Grundoperationen zusammensetzen. Der Sprachumfang lässt sich allerdings dynamisch um beliebige mathematische Funktionen erweiteren. Auf diese Weise werden zum Beispiel alle trigonometrischen Funktionen zur Verfügung gestellt.

- *Polymorphe Variablen*

  In Berechnungen oder als Parameter für Objektdefinitionen können einfache Zahlenwerte, Vektoren (2D und 3D), Matrizen (2D und 3D) und Zeichenketten verwendet werden. Die verschiedenen Datentypen können frei miteinander kombiniert werden. Gegebenenfalls werden automatische Konversionen vom System durchgeführt.

B.1 Sprachdefinition

BSDL ist sehr einfach aufgebaut und lässt sich mit Hilfe der erweiterten Backus Nauer Form (EBNF-Notation) wie folgt beschreiben:

Eine Welt besteht aus einer beliebigen Anzahl von Definitionen und Objekten.

```
World ::= { <Definition> | <Object> }+
```

Eine Definition erlaubt die Erzeugung von Ojekten, Namensräumen und Konstanten, die für den Szenenaufbau verwendet werden können. Anstelle von ***IDENTIFIER*** *können zur Laufzeit beliebige neue Schlüsselwörter konfiguriert werden.*

```
Definition ::= define IDENTIFIER <Specifier>
             |   define IDENTIFIER namespace ';'
             |   const  IDENTIFIER <Value>   ';'
```

Die nachstehende Regel spezifiziert das Aussehen von Objektdefinitionen. Sie werden durch ein Schlüsselwort eingeleitet und können mit einer beliebigen Anzahl von Parametern attributiert sein. Zudem ist eine beliebige Verschachtelung der Definitionen möglich.

Mit Hilfe der ***using*** *Direktive kann ein benutzerdefinierter Namensraum aktiviert werden (siehe das Beispiel im nächsten Abschnitt).*

```
Specifier     ::= IDENTIFIER <ValueList> <OptSpecifiers>
              |   using IDENTIFIER ';'

OptSpecifiers ::= ';'
              |   '{' { <Specifier> }+ '}' [ ';' ]

ValueList     ::= [ <Value> ]
              |   '(' <Values> ')'
```

Parameter für Objektdefinitionen oder Argumente von mathematischen Ausdrücken können sich aus Zahlwerten, Vektoren, Matrizen und Zeichenketten zusammensetzen. Im Unterschied zum ***IDENTIFIER*** *ist die Zeichenkette eines* ***STRING*** *durch Anführungsstriche begrenzt.*

```
Values ::= { <Value> ',' } <Value>

Value  ::= NUMBER
       |   VECTOR2D | VECTOR3D
       |   MATRIX2D | MATRIX3D
       |   STRING   | IDENTIFIER
       |   <Expression>
```

Mathematische Ausdrücke setzen sich aus den vier Grundoperationen zusammen. Eine beliebige Klammerung ist möglich. Zusätzlich werden Funktionen unterstützt, die dynamisch in den Sprachumfang integriert werden können.

```
Expression  ::= '(' <Value> ')'
            |   <Value> '+' <Value>
            |   <Value> '-' <Value>
            |   <Value> '*' <Value>
            |   <Value> '/' <Value>
            |   '-' <Value>
            |   IDENTIFIER '(' <Values> ')'
```

B.2 Beispiel

Die folgende einfache Beispielszene besteht aus zwei Kugeln. Die Szene verwendet zur Illustration alle Schlüsselwörter von BSDL.

Abbildung B.1 zeigt das Resultat, nachdem die Szenenbeschreibung mit Hilfe der Raytracer-Applikation verarbeitet wurde.

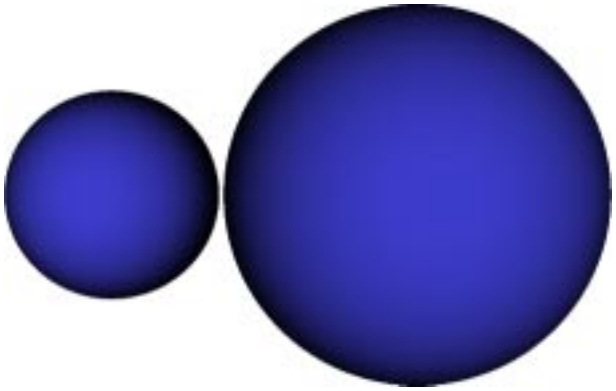


Abbildung B.1
Die einfache Beispielszene mit Hilfe eines Raytracers visualisiert.

```
using 3D; // Wir bewegen uns im 3D Namensraum.

camera {  // Eine perspektivische Kamera.
  perspective {
    eye    [1, 1.5, 11];  // Position,
    lookat [1, 1.5, 0 ];  // Fokussierungsort und
    up     [0, 1,   0 ];  // Orientierung der Kamera.
  }
}

// Ein eigener Namespace definieren.
define MyNS namespace;

// Texturdefinition.
define MyNS::kugelTextur phong {
  ambient [.1, .1, .3];
  diffuse [.3, .3, 1.];
}

// Der Kugelradius wird als Konstante definiert.
const RADIUS 1.4;

// Definition eine Punktlichtquelle mit Intensitaet 1
// und weisser Farbe.
pointLight (1, [1, 1, 1]) {
  position [0, 0, 100];
}
```

```
// 1. Kugel
sphere (RADIUS, [-1.5, 1.5, 0]) {
  MyNS::kugelTextur; // Definierte Textur verwenden.
}

// 2. Kugel mit Transformation.
sphere (RADIUS*1.8, [-1.5, 1.5, 0]) {
  translate [4, 0, 0];
  MyNS::kugelTextur;
}
```

Curriculum Vitae

Personalien:	Christoph Streit geboren am 2. März 1966 Riedweg 47 3705 Faulensee
Heimatort:	Jaberg, BE
Zivilstand:	Ledig
Bildungsweg:	1973 – 1982 Obligatorische Schulzeit 1982 – 1986 Gymnasium Interlaken 1987 – 1993 Informatikstudium, Universität Bern 1994 – 1997 Doktorstudium, Universität Bern
Lizentiatsarbeit:	*Modellierung mit Lindenmayer-Systemen in der Computergrafik* bei Prof. Dr. H. Bieri