Software-Architektur computergraphischer Systeme

Dr. Jürgen Döllner

Software-Architektur computergraphischer Systeme

Habilitationsschrift

vorgelegt von

Dr. rer. nat.
Jürgen Döllner

am Fachbereich
Mathematik und Informatik
der Westfälischen Wilhelms-Universität Münster

Dezember 2000

ZUSAMMENFASSUNG

Die vorliegende Arbeit diskutiert den Entwurf und die Implementierung von Software-Architekturen computergraphischer Systeme und stellt einen Ansatz zur Integration heterogener computergraphischer Software-Architekturen vor.

Der Begriff *computergraphische Systeme* bezeichnet im Allgemeinen Software und Hardware, die die computerunterstützte Modellierung realer oder imaginärer Objekte und die Bilderzeugung auf der Grundlage dieser Modelle übernehmen. Im Zentrum der vorliegenden Arbeit steht die diesen Systemen zugrundeliegende Software-Architektur. Die *Software-Architektur* eines Systems beschreibt die Struktur der das System implementierenden Software durch ein Software-Modell sowie das statische und dynamische Zusammenspiel der Modellkomponenten.

Die Arbeit analysiert zunächst die Software-Architekturen ausgewählter computergraphischer Systeme und evaluiert sie hinsichtlich ihrer Verständlichkeit, Wiederverwendbarkeit und Erweiterbarkeit. Betrachtet werden hauptsächlich solche Systeme, die allgemein als *Renderingsysteme* bezeichnet werden: Sie besitzen überwiegend eine auf die Bilderzeugung ausgerichtete Funktionalität. Als wesentliche Aspekte der Software-Architektur von Renderingsystemen identifizieren wir die Modellierung von Renderingkomponenten und Renderingverfahren.

Ein verallgemeinernder Entwurf eines *generischen Renderingsystems*, das unterschiedliche Renderingsysteme inkorporiert und durch eine methodisch einheitliche Schnittstelle abstrahiert, schließt sich an. Der Entwurf wird ergänzt durch ein Konzept für einen *generischen Szenengraphen*, der die Beschreibung graphisch-geometrischer Sachverhalte und ihre Umsetzung mit unterschiedlichen Renderingverfahren erlaubt.

Die Leistungsfähigkeit dieser Software-Architektur wird an mehreren Fallstudien aufgezeigt. Sie untersuchen die Integration von Multi-Pass-Rendering, von nichtphotorealistischem Rendering und von Terrain-Rendering in Verbindung mit dem generischen Renderingsystem und generischen Szenengraphen.

Schließlich zeigt die Arbeit, welche Konsequenzen und welche Möglichkeiten aus dem verallgemeinernden Entwurf des generischen Renderingsystems und des generischen Szenengraphen sich für die Software-Architektur zukünftiger computergraphischer Systeme ergeben.

VORWORT

Computergraphische Systeme sind als grundlegendes Werkzeug zur Mensch-Maschine-Kommunikation seit ihren Anfangstagen in den 50er Jahren zum Bestandteil von nahezu allen wissenschaftlichen und technischen Anwendungsprogrammen geworden. Die Entwicklungen im Hardware-Bereich bestimmten wesentlich die Fortschritte dieser Systeme: Mit der Einführung des Rasterdisplays und auf Computergraphik spezialisierter Prozessoren wuchsen die Möglichkeiten Computergraphik zur visuellen Kommunikation einzusetzen. Zugleich setzte die Entwicklung von Software-Systemen ein, die versuchten, die Leistung der Hardware in abstrakter und für den Software-Entwickler in verständlicher Form zugänglich werden zu lassen. Allerdings sind diese Bibliotheken durch die Heterogenität ihrer Konzepte und ihre Komplexität gekennzeichnet – es ist schwierig mit einem computergraphischen System zu entwickeln und noch viel schwieriger, mehrere in einer einzigen Anwendung zu integrieren. Insofern besteht ein Bedarf an einem übergreifenden Konzept und in der Methodik einheitlichen Framework für computergraphische Systeme. Dieser Bedarf wird durch die Entwicklung unterstrichen, dass computergraphische Systeme als allgegenwärtige Bestandteile in den meisten computerunterstützten Geräten – vom mobilen Telefon über das Navigationssystem bis hin zur Spielkonsole – Eingang finden. Die Frage bleibt, wie durch geeignete Software das Potential der Computergraphik und der sie implementierenden Systeme genutzt werden kann.

Diese Arbeit verfolgt das Ziel, das Software-Engineering im Bereich der Computergraphik weiterzuentwickeln. Dies soll durch die Analyse einer Reihe computergraphischer Systeme, aber auch durch einen Entwurf eines neuen, generischen Renderingsystems, das vorhandene Renderingsysteme inkorporiert, kapselt und abstrahiert, geschehen. Es ist nicht das Ziel dieser Arbeit, eine weitere Implementierung eines bestimmten computergraphischen Systems darzulegen, sondern die Gemeinsamkeiten und Unterschiede in der Software-Architektur computergraphischer Systeme aufzudecken, die Stärken und Schwächen ihrer Software-Architektur zu identifizieren und schließlich darüber nachzudenken, wie daraus für den Entwurf zukünftiger computergraphischer Systeme gelernt werden kann.

Diese Arbeit stellt die Software-Architektur von Renderingsystemen in den Mittelpunkt ihrer Betrachtung. Renderingsysteme, deren Hauptaufgabe es ist, Bilder computergraphischer Modelle zu synthetisieren, repräsentieren einen Kernbestandteil eines jeden computergraphischen Systems und besitzen deswegen einen großen Einfluss auf die Software-Gesamtarchitektur des Systems. Das Renderingsystem OpenGL wird zum Beispiel in einer nahezu unüberschaubaren Fülle interaktiver computergraphischer Systeme und Anwendungen eingesetzt und prägt deren Software-Architektur nicht

unerheblich. Modellierungssysteme, Computeranimationssysteme oder interaktive graphische Systeme, die auf Renderingsysteme aufbauen, werden in dieser Arbeit nicht im Detail betrachtet, um den Rahmen der Arbeit nicht zu sprengen. Die in dieser Arbeit entwickelte Sofware-Architektur eines verallgemeinernden, generischen Renderingsystems mag jedoch für die Software-Architektur zukünftiger computergraphischer Systeme von großem Interesse sein, denn das verwendete Renderingsystem legt letztlich wesentlich das Leistungsspektrum solcher Systeme fest.

Die Arbeit versteht sich auch als Zusammenfassung der Erfahrungen des Autors in der Anwendung, im Entwurf und in der Implementierung verschiedener computergraphischer Systeme. Die Erfahrungen waren nicht immer erfreulich: Das Verständnis der Software-Architektur dieser Systeme, ihre Ansteuerung und die hinter ihnen stehenden algorithmischen Aspekte sind in ihrer Gesamtheit von einer manchmal erdrückenden Komplexität. Auch deswegen ist diese Arbeit geschrieben: Einen Rahmen zu schaffen, der die Einordnung und das Verstehen der einzelnen Systeme erleichtert, eine Brücke schlägt zwischen den einzelnen Systemen und ihrer integrativen Nutzung den Weg bereiten will.

In der Darstellung habe ich mich bemüht, kompakt die wesentlichen Aspekte computergraphischer Systeme darzustellen. Grundlagen der Computergraphik sind darin kaum eingeführt; sie werden vom Leser erwartet und finden sich z.B. in dem Standardwerk „Computer Graphics: Principles and Practice“ von J. Foley et al. [33]. Ähnlich verfahre ich mit den Grundlagen des Software-Engineering, aus denen ich den objektorientierten Systementwurf, die objektorientierte Modellierung und neuere Entwicklungen, z.B. den komponentenorientierten Systementwurf, stillschweigend verwende, ohne die Begriffe im Detail zu erläutern. Dem Leser seien hierzu die einführenden Bücher „Object Oriented Modeling and Design“ von J. Rumbaugh [84], „Objektorientiertes Konstruktionshandbuch“ von H. Züllighoven [117] und „Component Software“ von C. Szyperski [102] empfohlen.

Was der Leser von dieser Arbeit erwarten kann: Der Computergraphik-Experte wird sich und seine Erfahrungen im Umgang mit computergraphischen Systemen und ihrer Software-Architektur in einem größeren Zusammenhang wiederfinden; der Entwickler computergraphischer Systeme wird darin Muster und Fragmente für eigene Software-Architekturen entdecken können; und Studenten können einen Abriss der gegenwärtigen Software-Architekturen computergraphischer Systeme vorfinden. Schließlich gibt die Arbeit eine kurze, konzeptionelle Einführung in die Software-Architektur des Virtuellen Renderingsystems VRS, das als Software-Produkt zur Herstellung eigener und Integration existierender Renderingsysteme verwendet werden kann.

INHALTSVERZEICHNIS

1 EINLEITUNG

Diese Arbeit hat die Software-Architektur computergraphischer Systeme zum Gegenstand. Die *Software-Architektur* eines Systems beschreibt die Struktur der das System implementierenden Software mit Hilfe eines Modells. Das Modell besteht aus Modellkomponenten, die zueinander in Beziehung stehen, wobei die Beziehungen sowohl statische als auch dynamische Aspekte des Zusammenspiels der Modellkomponenten bezeichnen können. Im Allgemeinen wird dazu im Bereich des Software-Engineerings ein objektorientiertes Modell der Software-Architektur erstellt. Modellkomponenten, die die statischen Aspekte des Systems beschreiben [84], umfassen Klassen, Vererbungsbeziehungen zwischen Klassen, Assoziationen zwischen Klassen sowie Klassenfamilien. Modellkomponenten zur Beschreibung dynamischer Aspekte umfassen Sequenz- und Kollaborationsdiagramme, die die Interaktion zwischen Objekten im Kontext eines bestimmten Szenarios aufzeigen [83]. Zur Spezifikation von Software-Architekturen kann die *Unified Modeling Language* (UML) [83] eingesetzt werden, die insbesondere eine graphische Notation für statische und dynamische Aspekte der Software-Architektur bereitstellt.

Der Begriff *computergraphische Systeme* bezeichnet Software und Hardware, die die computerunterstützte Modellierung realer oder imaginärer Objekte und die Bilderzeugung auf der Grundlage dieser Modelle übernehmen. Computergraphische Systeme können als Software-Bibliotheken, die zur Entwicklung computergraphischer Anwendungen eingesetzt werden können, oder als Anwendungsprogramme, die im Batch-Betrieb oder interaktiv mit dem Benutzer arbeiten, vorliegen. Beispiele computergraphischer Systeme sind OpenInventor, einer objektorientierten Software-Bibliothek für 3D-Computergraphik, und Alias|Wavefront von Pixar, eine interaktive, computergraphische Entwicklungsumgebung zur Produktion von Computeranimationen. Zu den konkreten Aufgaben computergraphischer Systeme zählen die Unterstützung geometrischer Modellierung, die hierarchische Modellierung von Szenen und das Rendering, wobei dort insbesondere die Texturierung, die Beleuchtung und die Schattierung von Szenen und ihren Objekten von besonderer Bedeutung sind.

Die Motivation für die Beschäftigung mit dem Thema der Software-Architektur computergraphischer Systeme hat mehrere Gründe. 1) Aufgrund der Komplexität der Software-Architektur heutiger computergraphischer Systeme sind diese Systeme nicht mehr ökonomisch erlernbar und vermittelbar. 2) Die Komplexität heutiger Renderingverfahren hat ein Maß erreicht, das es nicht erlaubt, Verfahren in eigene computergraphische Systeme oder Anwendungsprogramme zu integrieren, ohne Monate oder Jahre an der Integration zu arbeiten. 3) Schließlich besitzt die computergraphische Entwicklung eine solche Dynamik, dass starre, monolithische Software-Architekturen versagen, wollen sie einzelnen oder sogar mehreren neuen Entwicklungen folgen.

Die Arbeit beschreibt die Software-Architektur einer Reihe computergraphischer Systeme und darüber hinaus einen Ansatz, wie die Gemeinsamkeiten und Unterschiede computergraphi-

scher Systeme in einem konsistenten, konzeptionellen Rahmenwerk systematisch und methodisch ausgedrückt werden können. Mit dem Rahmenwerk werden zwei Ziele verfolgt: Zum einen repräsentiert es ein theoretisches Gerüst, mit dessen Hilfe bestehende und neue computergraphische Systeme zugeordnet, klassifiziert und verglichen werden können. Zum anderen dient es als Grundlage einer konkreten Implementierung eines *generischen Renderingsystems*, das in der Lage ist, bestehende Renderingsysteme aufzunehmen und diese mit Hilfe einer einheitlichen Systematik und Begrifflichkeit anzusteuern. *Generisch* wird hier wie im Angelsächsischen im Sinne von „relating to, or characteristic of a whole group or class“ (Webster's New Encyclopedic Dictionary) gebraucht; diese Verwendung des Begriffs findet sich z.B. bei objektorientierten Programmiersprachen (z.B. *generische Klasse* als Verallgemeinerung einer Familie von Klassen, die durch eine parametrisierte Klasse dargestellt wird [98]) und im Software-Engineering (z.B. *Genericity* als komplementärer Aspekt zu Kind-of- und Part-of-Beziehungen zwischen Klassen [64] und *generische Software-Architektur* im Sinne einer allgemeinen, auf eine Gruppe von Anwendungen übertragbare Software-Architektur, z.B. eine generische Software-Architektur für Data Warehouses).

Einen Schwerpunkt dieser Arbeit bilden Renderingsysteme. Unter *Rendering* werden gemeinhin alle computerunterstützten Verfahren verstanden, die computergraphische Modelle, z.B. dreidimensionale geometrische Modelle, in Form digitaler Bilder zu synthetisieren vermögen. Ein *Renderingsystem* bezeichnet ein System zur Erzeugung von Bildern auf der Grundlage computergraphischer Modelle. Renderingsysteme sind sowohl spezielle computergraphische Systeme, als auch Kernbestandteil jedes computergraphischen Systems, denn alle diese Systeme benötigen Bildsynthesefunktionalität. Ein Renderingsystem kann vollständig in Software oder durch den kombinierten Einsatz von Software und Hardware realisiert sein. Das Renderingsystem OpenGL stellt es z.B. OpenGL-Implementierungen frei, welche Abschnitte seiner Rendering-Pipeline in der Hardware und welche durch Software realisiert werden.

Der Begriff *Rendering* bezeichnet in der Computergraphik-Literatur sowohl den Prozess der Bildsynthese, als auch Bildsyntheseverfahren. Der Begriff kann und muss jedoch viel weiter gefasst werden. Rendering, abgeleitet vom lateinischen *reddere* bedeutet „umsetzen“, „übersetzen“ oder „übertragen“. In Bezug auf ein computergraphisches Modell kann von daher mit Rendering die Übertragung des Modells in ein Zielmedium oder mehrere Zielmedien bezeichnet werden. Eine virtuelle Umgebung, die sowohl graphisch-geometrische Objekte, aber auch Klänge und Geräuschquellen enthält, kann z.B. durch graphisches Rendering und Sound-Rendering in ein audiovisuelles Medium, z.B. Film, übertragen werden. In dieser Arbeit wird ein generisches Renderingsystem eingeführt und das Konzept des klassischen Szenengraphen erweitert und verallgemeinert zu einem generischen Szenengraphen, der einen graphisch-geometrischen Sachverhalt kodiert. Ein solcher Sachverhalt beschreibt ein Szenario mit Hilfe graphischer und geometrischer Ausdrucksmittel und legt fest, wie der Sachverhalt in ein z.B. visuelles oder audiovisuelles Zielmedium übertragen werden kann; die Übertragung in das Zielmedium setzt dabei ein generisches Renderingsystem voraus, das mit Hilfe unterschiedlicher, in ihm enthaltener Renderingsysteme die Übertragung bewerkstelligt.

1.1 Computergraphische Systeme

Computergraphische Systeme sind in heutigen Anwendungsprogrammen allgegenwärtig. Konzeptionell unterscheiden wir in der Software-Architektur eines computergraphischen Anwendungsprogramms zwischen der eigentlichen Anwendung und ihren Anwendungsmodellen (z.B. Netzüberwachungssoftware und Netzverbindungen), dem computergraphischen System und den computergraphischen Modellen, die im Allgemeinen von den Anwendungsmodellen abge-

leitet werden (z.B. graphische Repräsentation von Netzverbindung und Netzknoten), und schließlich der Computergraphik-Hardware und den dort verfügbaren Primitiven. Abbildung 1 illustriert den Aufbau eines typischen computergraphischen Anwendungsprogramms und zeigt die Mittlerfunktion des computergraphischen Systems zwischen Anwendungsmodellen und Computergraphik-Hardware.

1.1.1 Computergraphische Klassenbibliotheken und Frameworks

Computergraphische Klassenbibliotheken und Frameworks bilden eine wichtige Gruppe computergraphischer Systeme. Seit Anfang der 90er Jahre begann die Entwicklung solcher computergraphischer Systeme, die insbesondere Fragestellungen aus dem Gebiet des Software-Engineerings und der Objektorientierung untersuchen; Wisskirchen [116] diskutiert z.B. die objektorientierte Programmierung computergraphischer Systeme. Das Ziel dieser computergraphischen Systeme war und ist es, sowohl die Spezifikation computergraphischer Anwendungsprogramme, als auch die Implementierung von und die Erweiterung des Systems durch neue Renderingverfahren mit Hilfe von Methoden des Software-Engineerings zu unterstützen.

Eine *Klassenbibliothek* fasst eine Menge von einzeln anwendbaren Klassen zu einer Einheit zusammen; die Klassen werden bei der Verwendung direkt benutzt oder spezialisiert [117]. Eine Klassenbibliothek gibt im Allgemeinen keinen Kontrollfluss für das Anwendungsprogramm vor. Einzelne Klassen einer Klassenbibliothek stehen meist durch objektorientierte Konzepte, wie z.B. das Konzept der Assoziation, der Vererbung oder der Schnittstellenimplementierung miteinander in Beziehung. Anwendungen nehmen einzelne Klassen der Klassenbibliothek in Anspruch, indem im Anwendungsprogramm Instanzen dieser Klassen konstruiert, ihre Methoden aufgerufen und ihre Attribute manipuliert werden. Klassenbibliotheken können von einer Anwendung zur Implementierung anwendungsspezifischer Funktionalität herangezogen werden, indem z.B. neue Klassen von vorhandenen Klassen durch Vererbung abgeleitet werden.

Ein *Framework* repräsentiert einen Software-Architekturentwurf für eine bestimmte Kategorie von Anwendungen auf der Grundlage einer Menge kooperierender Klassen [102]; ein Framework besteht aus einer Menge kooperierender Klassen, die eine allgemeine, generische Lösung für ähnliche Probleme in einem bestimmten Kontext vorgibt [117]. Frameworks können eine Reihe ihrer Klassen offen legen, so dass eine einzelne Anwendung spezialisierte Klassen ableiten kann; einzelne Klassen des Frameworks können auch abstrakt entworfen sein, so dass die Anwendung solche Klassen durch Vererbung konkretisieren muss. Im Gegensatz zum Herangehen bei einer Klassenbibliothek implementiert eine Anwendung auf der Grundlage

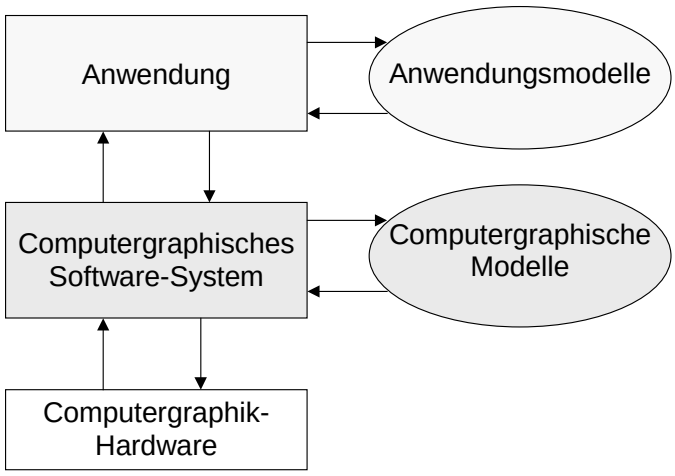

Abbildung 1. Software-Architektur computergraphischer Anwendungsprogramme.

eines Frameworks ihre Funktionalität, indem sie meist nur wenige, ausgezeichnete Klassen des Frameworks ergänzt oder spezialisiert.

Im folgenden Überblick finden sich Beispiele computergraphischer Systeme, die in Form einer Klassenbibliothek oder eines Frameworks realisiert wurden. Gemeinsam ist ihnen, dass sie für die Entwicklung computergraphischer Anwendungsprogramme ausgelegt sind. Die für diese Arbeit relevanten Konzepte der Systeme werden im Kapitel 2 näher besprochen.

- *OpenInventor* [112] ist eine computergraphische Klassenbibliothek, die auf der Basis von OpenGL die Modellierung von Szenengraphen unterstützt. Die Klassenbibliothek eignet sich durch ihre effiziente Nutzung von OpenGL in besonderem Maße zur Entwicklung interaktiver, graphischer Anwendungen. Das Konzept des Szenengraphen wurde in vielen anderen Systemen und Standards (z.B. VRML [51]) übernommen.
- *Java3D* [100] ist eine computergraphische Klassenbibliothek, die in Form eines Java-Paketes vorliegt. Sie enthält Klassen, die denen von OpenInventor ähneln, ist ausgelegt für Echtzeit-Rendering und unterstützt neben nichtimmersiven auch immersive Ausgabegeräte, wie sie insbesondere von Virtual-Reality-Anwendungen benötigt werden. Java3D legt nicht fest, welches Renderingsystem zu seiner Implementierung genutzt wird.
- *Generic-3D* [8] ist ein computergraphischer Framework, der Komponenten zur Konstruktion computergraphischer Systeme bereitstellt – ein konkretes System entsteht durch Instanziierung und Konfiguration der einzelnen Komponenten. Es unterstützt verschiedene Renderingverfahren (z.B. Ray-Tracing und Radiosity) und unterschiedliche Szenenrepräsentationen (z.B. Szenengraphen und Displaylisten).
- *MAM* [28] ist eine computergraphische Klassenbibliothek für interaktive, animierte 3D-Graphik. Es basiert auf dem Prinzip des symmetrischen Modellierens von Szenen und Verhalten in Form zweier durch Constraints verbundener Graphen, dem Szenengraphen und dem Verhaltensgraphen. Es verallgemeinert das Szenengraph-Konzept von OpenInventor und erweitert es auf die explizite Beschreibung zeitabhängiger und ereignisabhängiger Prozesse.
- *BOOGA* [95] ist ein computergraphisches Framework, das zur Entwicklung computergraphischer Anwendungsprogramme eingesetzt wird und durch seine Software-Architektur die Erweiterbarkeit im besonderen Maße unterstützt. Zum Rendering wird ein Ray-Tracer bereitgestellt, der in Form einer Komponente gegen andere Bildsyntheseverfahren ausgetauscht werden kann.

Computergraphische Systeme in Form von Klassenbibliotheken oder Frameworks versuchen mit Hilfe einer objektorientierten Software-Architektur sowohl ein großes Leistungsspektrum als auch eine kompakte Programmierung computergraphischer Anwendungsprogramme zu erreichen. In dieser Arbeit wird sich zeigen, dass 1. Objektorientierung in der Implementierung eines Systems wenig Auswirkung auf diese Ziele hat und 2. diese Systeme der Dynamik computergraphischer Entwicklung insbesondere in den Bereichen des Echtzeit-Renderings und des nichtphotorealistischen Renderings häufig nicht gewachsen sind. Dadurch kann erklärt werden, warum Software-Entwickler häufig statt einer objektorientierten Klassenbibliothek oder eines Frameworks elementare Renderingsysteme (z.B. OpenGL) bevorzugen.

1.1.2 Renderingsysteme

Die computergraphischen Systeme, die in dieser Arbeit untersucht werden, sind größtenteils Renderingsysteme. Zunächst folgt ein Überblick über eine Reihe charakteristischer Vertreter von Renderingsystemen:

- *OpenGL* ist ein Renderingsystem für 2D- und 3D-Computergraphik, das zur Entwicklung interaktiver, animierter Anwendungen eingesetzt wird [108]. OpenGL definiert eine auf das Echtzeit-Rendering ausgelegte Rendering-Pipeline und unterstützt sowohl geometrische Primitive als auch Bildoperationen. OpenGL ist auf nahezu jeder Plattform verfügbar – entweder als native Implementierung des jeweiligen Betriebssystem-Herstellers (z.B. SUN-OpenGL, Microsoft-OpenGL) oder in Form der freien OpenGL-konformen Implementierung *Mesa* [77].
- *RenderMan* [104] ist ein Renderingsystem für photorealistische und photosurrealistische 3D-Graphik, das zur Realisation hochwertiger Einzelbilder oder Bildsequenzen eingesetzt wird. RenderMan selbst definiert dabei ein Austauschformat für 3D-Szenen (*RIB*-Format) und eine Shading-Sprache [3]. RenderMan ermöglicht die Spezifikation von Szenen, ohne dabei die Art des Renderings, ob Scanline-basiert, Ray-Tracing-basiert oder Radiosity-basiert, festzulegen. Zwei Implementierungen sind verfügbar, die RenderMan-Implementierung von Pixar und die freie Implementierung in Form der Blue Moon Rendering Tools *BMRT* [39].
- *POV-Ray* [78] ist ein Renderingsystem für photorealistische und photosurrealistische 3D-Graphik, das ebenfalls zur Realisation von hochwertigen, Ray-Tracing-basierten Bildern eingesetzt wird. POV-Ray definiert in Analogie zu RenderMan eine komplexe Spezifikationssprache für Szenen. POV-Ray verfügt sowohl über ein Ray-Tracing-Renderingverfahren als auch über ein kombiniertes Ray-Tracing/Radiosity-Renderingverfahren. Die Implementierung von POV-Ray ist frei erhältlich.
- *Radiance* [110] ist ein Renderingsystem für physikalisch-basierte Beleuchtungssimulation, das zur Realisation – im Sinne ihrer physikalischen Korrektheit – hochwertiger photorealistischer Bilder dient. Radiance verfügt nur über eine elementare Szenenbeschreibungssprache, die primär für andere computergraphische Systeme gedacht ist, die maschinell Radiance-Szenenbeschreibungen erzeugen. Die Szenenbeschreibungssprache ermöglicht eingeschränkt die Programmierung von Shadern. Die Implementierung von Radiance ist frei erhältlich.

Die Vielzahl der darüber hinaus existierenden Renderingsysteme, die meisten universitären Ursprungs, können bezüglich ihrer Software-Architektur und ihres Leistungsumfanges einem der obigen Renderingsysteme zugeordnet werden. Hinzu kommt, dass hier nur solche Systeme betrachtet werden, über deren Aufbau grundlegendes bekannt, publiziert oder einsehbar ist. Kommerzielle Systeme sind daher nicht in die Betrachtungen mit einbezogen. Überdies nutzen sie meist eines der vorgestellten Renderingsysteme als Kernbestandteil und legen häufig ihren Entwicklungsschwerpunkt auf die Benutzerschnittstelle und die Szenenmodellierung.

Den vorgestellten Renderingsystemen ist gemeinsam, dass sie über einen längeren Zeitraum inkrementell entwickelt wurden. Ihr Leistungsspektrum orientiert sich an dem von ihnen eingesetzten Renderingverfahren, so dass wir sie insbesondere anhand ihres Beleuchtungsmodells – ob lokal oder global – unterscheiden können. Als Echtzeit-Renderingsystem ist OpenGL ein De-fakto-Standard; im Ray-Tracing-Bereich konkurrieren mehrere Systeme, unter ihnen POV-Ray; im Bereich Radiosity gilt Radiance als ein exponierter Vertreter; RenderMan gilt als ein Renderingsystem mit ausgefeilten Möglichkeiten für eine Programmierung anwendungsspezifischer Effekte und Gestaltungen.

Konventionelle Renderingsysteme verfügen aufgrund ihrer langjährigen Entwicklung über eine auf ihr Renderingverfahren abgestimmte, umfassende Funktionalität und erreichen eine hohe Geschwindigkeit und Stabilität. Ein Neuentwurf dieser Systeme in einem einheitlichen *Implementierungsframework*, der die Nutzung der verschiedenen Renderingverfahren ermöglichen würde, scheidet wegen der technischen und organisatorischen Komplexität eines solchen Vorhabens aus. Ein Weg zur integrativen Nutzung verschiedener Renderingverfahren besteht jedoch darin, einen *Schnittstellenframework* zu entwickeln, der die einzelnen Systeme durch eine einheitliche Schnittstelle ansteuert.

1.2 Software-Architektur computergraphischer Systeme

Die Software-Architektur eines computergraphischen Systems lässt sich anhand des Systemaufbaus, der Schnittstellen und der Erweiterungsfähigkeiten charakterisieren. Das Verständnis dieser Charakteristiken ist die Voraussetzung für Überlegungen, wie die diesen Systemen innewohnende Komplexität durch einen verallgemeinernden Software-Architekturansatz bewältigt werden kann. Der Begriff Komplexität wird hier – im Gegensatz zur Bedeutung in anderen Bereichen (z.B. Laufzeit-Komplexität in der Algorithmischen Geometrie) – für den Grad des Komplex-Seins gebraucht, wobei *komplex* für „having many interrelated parts, patterns, or elements that are hard to separat, analyze, or solve“ (Webster’s New Encyclopedic Dictionary) steht. Komplexität bezeichnet im folgenden also den *Grad der Vielschichtigkeit und der Verwobenheit des Aufbaus* eines computergraphischen Systems.

1.2.1 Systemaufbau

Komplexe Systeme sind gewöhnlich aus Subsystemen aufgebaut, die Bestandteile des Ganzen bilden, dabei aber unabhängig modelliert sein können (z.B. bei umfangreichen Software-Bibliotheken) oder unter einer einheitlichen Schnittstelle, der Fassade, zusammengefasst werden können (z.B. bei sog. „Black-Box“-Systemen). Ein Systemaufbau mit Subsystemen ist dort zu finden, wo ein computergraphisches System konfigurierbar und unabhängig erweiterbar sein soll. Nur Subsysteme, deren Funktionalität tatsächlich benötigt wird, werden von Anwendungsprogrammen tatsächlich genutzt. Der Systemaufbau wird durch die Aufteilung in Subsysteme komplexer, bekommt aber durch die Konfigurierbarkeit eine hohe Flexibilität.

Ein Beispiel eines solchen Subsystemansatzes findet sich bei OpenGL (siehe Abbildung 2). Modellierungs- und Visualisierungstechniken für komplexe 3D-Modelle werden vom *OpenGL Optimizer* Subsystem [91] bereitgestellt. Die geometrische Modellierung von Extrusi-

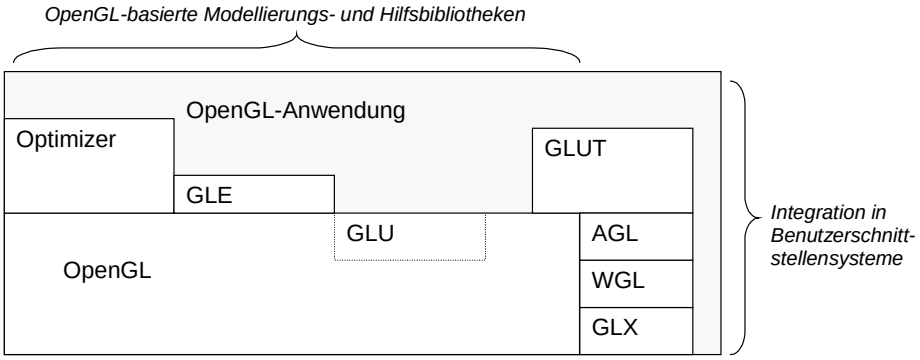

Abbildung 2. Software-Architektur einer OpenGL-basierten Anwendung.

onskörpern übernimmt die *OpenGL Tubing and Extrusion Extension* (GLE). Weiter existieren verschiedene Erweiterungen, um OpenGL in ein User-Interface-System einzubetten, so z.B. für X11 (GLX), Microsoft Windows (WGL) und das User-Interface-System von Apple (AGL). Alternativ steht mit dem *OpenGL Utility Toolkit* (GLUT) ein plattformneutrales, einfaches User-Interface-System bereit.

Im Fall von OpenGL zeigt die *OpenGL-Utilities*-Erweiterung (GLU) die Standardtransformation und Standardprojektionen sowie Standardgeometrien bereitstellt, wie Subsysteme, die sich über einen längeren Zeitraum bewährt haben, in den Kern eines Systems migrieren. Nur ein klar definiertes Konzept für solche „Plug-Ins“ ermöglicht es, dass sich Erweiterungen entwickeln (aber auch konkurrieren) und dann in das Gesamtsystem inkorporiert werden.

1.2.2 Systemschnittstellen

Die Komplexität computergraphischer Systeme zeigt sich an der Schnittstelle zwischen Renderingsystem und Anwendungsprogramm. Die Schnittstelle umfasst im Allgemeinen auf hoher Abstraktionsstufe eine Sammlung von Konstrukten zur Spezifikation von Szenen und auf niedriger Abstraktionsstufe eine Sammlung von Renderingoperationen und Renderingprimitiven.

Die Systemschnittstellen lassen sich unterteilen in solche, die eine Programmierschnittstelle, ein sog. *Application Programming Interface* (API) bereitstellen und somit zur Entwicklung von Anwendungsprogrammen verwendet werden können, und solche, die ausschließlich ein Szenen-Spezifikationsformat vorgeben, das von dem als externes Programm vorliegenden Renderingsystem implementiert wird. Nur APIs ermöglichen die Konstruktion interaktiver Anwendungen, da aus Gründen der Kopplung und Effizienz nicht mit Hilfe eines externen Renderingsystems interaktiv gearbeitet werden kann; auch ist eine Einbettung der Ausgabe der generierten Bilder in eine von der Anwendung kontrollierte Benutzerschnittstelle nur mit Hilfe eines API möglich.

OpenGL stellt ein C-API bereit und lässt sich in Form einer Software-Bibliothek vollständig in Anwendungsprogramme einbinden. Die Nutzung von RenderMan gestaltet sich anders: Eine Anwendung generiert Szenenbeschreibungen im RIB-Format (RenderMan Interface Bytestream). Zusätzlich können anwendungsspezifische Shader geschrieben werden, die von einem externen Shader-Compiler übersetzt werden müssen. Zur Bildsynthese interpretiert die RenderMan-Implementierung Szenenbeschreibungen im RIB-Format und benötigt die dort referenzierten Shader (siehe Abbildung 3). Die derzeitigen RenderMan-Implementierungen (Photorealistic RenderMan von Pixar und BMRT) lassen sich nicht in Form einer Software-Bibliothek integrieren, wobei hier kein technischer, als vielmehr ein strategischer Grund vor-

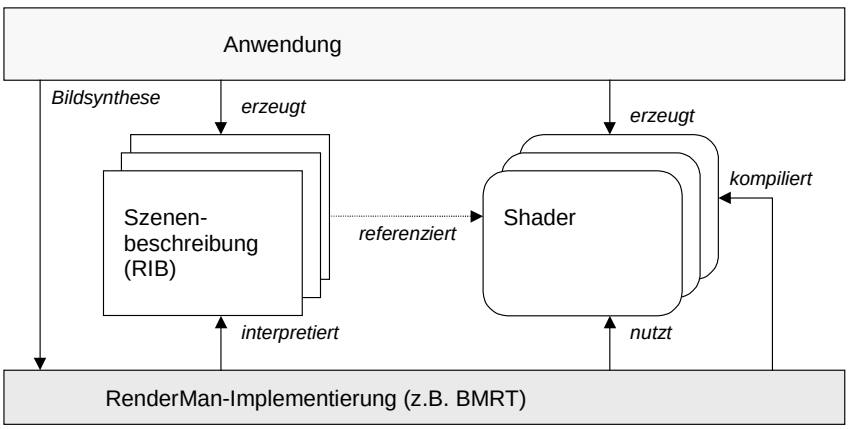

Abbildung 3. Schnittstellenstruktur eines RenderMan-basierten Anwendungsprogramms.

liegt: Anwendungsprogramme werden gezwungen, die Benutzung des Renderingsystems offen zu legen, denn das Renderingsystem muss separat von der Anwendung installiert und betrieben werden. Für RenderMan, POV-Ray und Radiance sind bislang keine APIs verfügbar, die die Szenenmodellierung und Bildsynthese in eine Anwendung vollständig zu integrieren vermögen.

Je nach Typ der Schnittstelle werden von Renderingsystemen Datentypen bereitgestellt und Szenenformate definiert. Die Menge der Schnittstellenkonstrukte ist fast immer in quantitativer Hinsicht groß. RenderMan z.B. umfasst ca. 120 Befehle, OpenGL ca. 250 Befehle und MAM ca. 300 Klassen. Über jedes Schnittstellenkonstrukt muss darüber hinaus der Anwendungsentwickler detaillierte Kenntnisse besitzen, so z.B. Parameter-Beschränkungen oder Aufruf-Beschränkungen. Für die Nutzung eines computergraphischen Systems sind daher nicht nur Kenntnisse der Schnittstelle, sondern auch der zugrunde liegenden Annahmen und Beschränkungen unabdingbar.

1.2.3 Systemerweiterung

Computergraphische Systeme dienen der Entwicklung computergraphischer Anwendungen. Nicht immer erfüllen die Systeme alle Anforderungen, die die Anwendung stellt, und nicht immer sind solche Anforderungen anfänglich bekannt – Erweiterbarkeit ist damit eine grundlegende Qualität, soll ein computergraphisches System auf lange Sicht für die Entwicklung und Weiterentwicklung einer Anwendung genutzt werden können. Die Erweiterungsmechanismen sind vielfältig und wesentlich bestimmt von der Art der Systemschnittstelle.

Steht ein computergraphisches System nur als Black-Box-System zur Verfügung, wie z.B. das Lichtsimulationssystem Radiance, kann es nicht erweitert werden. Auf der Basis der vorhandenen Konstrukte lassen sich allenfalls Generatoren oder Makro-Mechanismen implementieren; die Renderingfähigkeiten des Systems können nicht erweitert werden. Andere Systeme können Mechanismen bereitstellen, um partiell die Funktionalität zu erweitern, etwa im Fall von RenderMan, das die Programmierung von sog. Shadern und damit die Implementierung von neuen Renderingstilen erlaubt.

Das System kann auch vollständig erweiterbar sein, wenn es in Form einer White-Box-Klassenbibliothek oder eines White-Box-Frameworks [117] entworfen wurde, etwa im Fall von BOOGA [2] und Generic-3D [8].

Die Erweiterungsmöglichkeiten hängen außerdem von der Mächtigkeit und Körnung der Schnittstellenkonstrukte ab. OpenGL gilt als Musterbeispiel eines mächtigen computergraphischen Systems, da es durch eine große Sammlung elementarer Konstrukte die Realisation von Renderingaufgaben ermöglicht, die bei seinem Entwurf nicht absehbar waren. Als Beispiel sei die Implementierung der pixelpräzisen Anwendung des Phong-Beleuchtungsmodells durch Texturierung genannt. Als Beispiel für fehlende Erweiterbarkeit kann VRML angeführt werden, das die Spezifikation neuer Konstrukte nicht unterstützt – ein Zugang zu den jeweils tatsächlich genutzten Renderingsystemen fehlt.

> Die Software-Architektur computergraphischer Systeme muss hinsichtlich ihres Systemaufbaus, ihrer Systemschnittstellen und ihrer Systemerweiterbarkeit beurteilt werden. Diese drei Aspekte entscheiden wesentlich über die Verständlichkeit, Verwendbarkeit und Erweiterbarkeit des computergraphischen Systems.

1.3 Komplexität von Renderingverfahren

Aus der Sicht der Software-Architektur zeigt sich die Komplexität eines Renderingverfahrens an der Struktur der Verarbeitung der Szeneninformationen. Die Verarbeitung der Szeneninformationen wird durch die Rendering-Pipeline festgelegt; die Rendering-Pipeline ist eines der zentralen Elemente der Software-Architektur von Renderingsystemen. Eine Rendering-Pipeline besteht im Allgemeinen aus drei Stufen:

- **Anwendungsstufe**: In dieser Stufe werden Anwendungsmodelle zu computergraphischen Modellen verarbeitet bzw. die computergraphischen Modelle aktualisiert.
- **Geometriestufe**: In dieser Stufe werden Szenenobjekte geometrisch transformiert und auf die Kameraebene projiziert.
- **Rasterisierungsstufe**: In dieser Stufe werden Primitive rasterisiert und texturiert.

Insbesondere im Echtzeit-Rendering werden diese unter dem Gesichtspunkt der Renderinggeschwindigkeit und Parallelisierbarkeit betrachtet; einen Überblick über den technischen Aufbau der Rendering-Pipeline und die Optimierungsmöglichkeiten geben Möller und Haines [66].

1.3.1 Single-Pass-Rendering

Im einfachsten Fall lässt sich ein Renderingverfahren in nur einem Durchlauf (Pass) bewältigen. Dazu müssen Objekte aus der Anwendungsstufe in geeignete Objekte der Geometriestufe übersetzt werden. Dieser Vorgang muss wiederholt werden, wenn sich Anwendungsobjekte ändern. Die Objekte der Geometriestufe müssen anschließend in Primitive der Rasterisierungsstufe übertragen werden. Dies ist gewöhnlich ein zeitkritischer Vorgang.

Zum Beispiel werden im Fall von OpenGL Geometrieobjekte in OpenGL-Primitive zerlegt und von dem OpenGL-Rasterisierer unmittelbar verwertet, indem die resultierenden Fragmente in die jeweils aktivierten Framebuffer-Komponenten (Color-Buffer, Depth-Buffer, Stencil-Buffer, Accumulation-Buffer) eingetragen werden, sofern bestimmte Tests der Rasterisierungspipeline (Stencil-Test, Depth-Test, Alpha-Test) bestanden sind [108].

Historisch betrachtet prägte das Single-Pass-Rendering das Grundmuster, wie computergraphische Systeme 3D-Szenen spezifizieren: Die Objektbeschreibungen einer Szene werden linearisiert und einmalig verarbeitet. Der Szenengraph ist einer der markantesten Vertreter einer durch Traversierung linearisierten hierarchischen Beschreibung einer Szene, die so direkt von einem nachgeschalteten Verwerter, klassisch dem Rasterisierer, verwertet werden kann.

1.3.2 Multi-Pass-Rendering

Das Multi-Pass-Rendering ist zur Realisation von Renderingverfahren notwendig, die globale Zusammenhänge zwischen den Geometrieobjekten erfordern (z.B. Ray-Tracing) oder auf akkumulierenden bildgebenden Verfahren beruhen (z.B. Bildprojektion mit physikalisch-basiertem Linsensystem [47]). Solche Verfahren müssen Szenenbeschreibungen mitunter mehrfach der nachgeschalteten Stufe, z.B. dem Rasterisierer, übermitteln. In jedem Durchlauf werden dabei die Geometrieobjekte unterschiedlich ausgewertet. Objekte können z.B. in einem konkreten Durchlauf zwar rasterisiert, aber nicht in den Color-Buffer gezeichnet werden, um ausschließlich den Stencil-Buffer zu aktualisieren.

Zum Beispiel benötigt ein OpenGL-Renderingverfahren zum Zeichnen von Drahtgittermodellen, in denen ausschließlich die sichtbaren Anteile der „Drähte“ gezeichnet werden, zwei Durchläufe. Im ersten Durchlauf wird das Modell mit gefüllten Flächen „gezeichnet“, ohne in

den Color-Buffer zu schreiben, jedoch wird der Depth-Buffer aktualisiert. Im zweiten Pass wird der Color-Buffer wieder freigegeben und es wird das Drahtgittermodell gezeichnet – nur in bezug auf das Solid-Modell sichtbare Linien werden gezeichnet.

Eine Szenenbeschreibung muss für das Multi-Pass-Rendering mehrfach traversiert werden. In den einzelnen Durchläufen kann die Traversierung mitunter optimiert werden, z.B. indem auf bereits im ersten Durchlauf berechnete und vorgehaltene Information zurückgegriffen wird. Konzeptionell ist diese letztlich lineare Beschreibung einer Szene nur bedingt brauchbar, um komplexe Zusammenhänge zwischen Geometrieobjekten auszudrücken.

Multi-Pass-Rendering eröffnet überraschend viele Möglichkeiten, um mit klassischen Echtzeit-Renderingsystemen anspruchsvolle Renderingverfahren zu implementieren. Zur Solid-Modellierung wurde z.B. ein Multi-Pass-Algorithmus von Wiegand [114] entwickelt, der Solide, die durch Boolsche Mengenoperationen verknüpft werden, im Bildraum visualisiert; dieser Ansatz von Constructive-Solid-Geometry (CSG) arbeitet ausschließlich im Bildraum, ohne analytische Berechnungen im Objektraum zu benötigen und kann von daher in Echtzeit zur visuellen Solid-Modellierung eingesetzt werden. Viele der sog. „Advanced Graphics Programming Techniques“ für OpenGL [62], die durchgängig das Ziel verfolgen, den Grad an visueller Komplexität in Bildern zu steigern, beruhen auf Multi-Pass-Rendering.

Die Software-Architektur computergraphischer Systeme beruht auf Szenenspezifikationsformen, die in Hinblick auf die konkret verwendete Rendering-Pipeline und für Single-Pass-Rendering entworfen wurden. Den Anforderungen des Multi-Pass-Renderings werden sie nur bedingt gerecht. Mitunter bleibt für Multi-Pass-Rendering nur der Weg, durch temporäre Hilfsstrukturen Zusammenhänge aus der Szenenbeschreibung abzuleiten. Eine Erweiterung des Szenengraphen-Konzepts und der Traversierungsfunktionalität ist daher erforderlich, um Multi-Pass-Rendering technisch effizient zu unterstützen.

1.4 Dynamik computergraphischer Entwicklungen

Wissenschaft und Forschung im Bereich der Computergraphik unterliegen einer rasanten und manchmal eigenwilligen dynamischen Entwicklung, die insbesondere in den letzten Jahren stark in Wechselbeziehung mit ihrem industriellen Einsatz steht. Durch den Einsatz computergraphischer Systeme als Schlüsseltechnologie in Massenprodukten und Massenmärkten kann auch in Zukunft von einer rasanten Entwicklung ausgegangen werden. Drei ausgewählte Entwicklungen mögen dieses Phänomen verdeutlichen.

1.4.1 Echtzeit-Rendering

Ziel des Echtzeit-Renderings ist es, Renderingverfahren zu realisieren, die die Interaktion mit und die Animation von Szenen erlauben. Wir sprechen von einer Echtzeit-Darstellung, wenn der Bildaufbau so schnell erfolgt, dass für den Betrachter der Eindruck einer flüssigen, kontinuierlichen Darstellung entsteht. Die Rate, mit der Bilder angezeigt werden, wird in Frames-per-Second (fps) gemessen. Bei 6 fps entsteht ein erster Echtzeit-Eindruck, bei 15 fps kann im Allgemeinen problemlos gearbeitet werden und ab ca. 72 fps ist eine Steigerung des Echtzeit-Eindrucks durch den Benutzer nicht mehr wahrnehmbar [66]. Das Echtzeit-Rendering ist eng gekoppelt an die Entwicklung der Computergraphik-Hardware.

Das Echtzeit-Rendering stellt präzise Forderungen an die Software-Architektur computergraphischer Systeme, so z.B. die Forderung nach Level-of-Detail-Modellen für komplexe 3D-Objekte oder nach echtzeitfähigen Verfahren für Schattierung und Beleuchtung (z.B. Schatten und Reflexion). Software-Architekturen müssen gerade hierfür Erweiterbarkeit gewährleisten, um neue Verfahren unmittelbar integrieren zu können. Die Fallstudien in den Kapiteln 5 und 7 befassen sich am Beispiel von Multi-Pass-Rendering und Terrain-Rendering mit der Integration von echtzeitfähigen Verfahren in ein generisches Renderingsystem.

1.4.2 Nichtphotorealistisches Rendering

Unter nichtphotorealistischem Rendering (NPR) – wenn überhaupt schon eine klare Definition dieses Begriffs gegeben werden kann – werden bildgebende Verfahren verstanden, die nicht notwendig zum Ziel haben, eine Szene hinsichtlich Licht und Materialeigenschaften physikalisch korrekt wiederzugeben, sondern alle Freiheitsgrade klassischer, künstlerischer Gestaltung, Zeichnung und Malerei zulassen. Einen Überblick über nichtphotorealistisches Rendering geben Landsdown und Schoefield [56] sowie Strothotte und Schlechtweg [97].

Im nichtphotorealistischen Rendering können wir technisch zwischen bildpräzisen und objektpräzisen Verfahren differenzieren. Bildpräzise Verfahren operieren im Bildraum, wohingegen objektpräzise Verfahren auf analytischen Berechnungen beruhen. Kennzeichnend für alle NPR-Verfahren ist es, dass sie nicht in das Schema einer linearen, einmalig zu durchlaufenden Rendering-Pipeline passen. Sowohl das eigentliche Rendering, der Zeichenvorgang, als auch die Vorarbeiten gestalten sich vielschichtiger. Die Fallstudie in Kapitel 6 gibt einen Einblick in die Schwierigkeiten und Möglichkeiten des Software-Engineerings für NPR.

NPR als relativ junge computergraphische Disziplin wird insbesondere in Zukunft neue Aufgaben für den Entwurf der Software-Architektur computergraphischer Systeme bereithalten. Zentrale Anforderungen sind u.a. die Bereitstellung von Frameworks für NPR-Anwendungen und die Verbindung von NPR mit Echtzeit-Renderingsystemen.

1.4.3 Nichtstandard-Rendering

Unter dem Begriff „Nichtstandard-Rendering“ sollen im folgenden alle Renderingverfahren verstanden werden, die eine Szene nicht notwendig in das Zielmedium Bild übertragen, sondern auch oder ausschließlich andere Zielmedien benutzen. Das Sound-Rendering bietet ein Beispiel hierfür. Es bezeichnet die Wiedergabe digitaler Audio-Dokumente und erfordert technisch das Vorhandensein spezieller Hardware. Insbesondere im Bereich audiovisueller Medien ist das Rendering animierter Graphik oder Videobilder mit dem Rendering eines assoziierten Sounds verbunden.

Die Frage, die sich in diesem Zusammenhang der Software-Architektur computergraphischer Systeme stellt, lautet: Wie können Nichtstandard-Renderingverfahren in ein allgemeines Konzept des Renderings aufgenommen werden? Mögliche Wege für eine solche Integration zeigen z.B. die Bemühungen im Bereich der Medien-Standardisierung, etwa die der Motion-Picture-Expert Group MPEG. Im MPEG-4 Standard werden Audio-Datenströme und visuelle Datenströme in ein gemeinsames Zeitkoordinatensystem gebracht und können simultan gerendert werden.

Die Software-Architektur computergraphischer Systeme muss der Dynamik computergraphischer Entwicklung gerecht werden. Ohne eine Software-Architektur, die diese Unvollständigkeit heutiger Renderingverfahren anerkennt und dem durch ihre Transparenz und Erweiterbarkeit entgegentritt, wird ein computergraphisches System langfristig den Entwicklungen nicht folgen können.

1.5 Herausforderungen für die Software-Architekturen

Die Software-Architektur eines computergraphischen Systems muss die Komplexität des Systemaufbaus, die Komplexität der Renderingverfahren und die Dynamik computergraphischer Entwicklungen als wesentliche Faktoren berücksichtigen. Von einigen Schritten und Überlegungen, diesen Herausforderungen gerecht zu werden, handelt diese Arbeit. Die wesentlichen Aspekte hierzu lassen sich wie folgt kurz skizzieren:

- *Verallgemeinerung des Renderingbegriffs.* Es ist notwendig, den Begriff des Renderings zu verallgemeinern, ihn weiter als in seiner historischen Bedeutung, der der Bildsynthese, zu fassen, indem darunter alle Formen der Übertragung graphisch-geometrischer Sachverhalte in ein Zielmedium verstanden werden. Die Einsatzmöglichkeiten eines Renderingkonzepts werden dadurch vergrößert – von der klassischen Bildsynthese, über das Sound-Rendering, bis hin zum Rendering multimedialer Dokumente.
- *Verallgemeinerung der Szenenrepräsentation.* Es ist notwendig, die Fülle der Übertragungen und des zu Übertragenden ausdrücken zu können. Dazu muss die Szenenbeschreibung zu einer Beschreibung eines graphisch-geometrischen Sachverhalts verallgemeinert werden. Eine solche Beschreibung muss die Vielfalt der Attributierung und die Vielfalt der Renderingverfahren unterstützen.
- *Systematisierung der Renderingkomponenten.* Es ist notwendig, die am Rendering in dieser allgemeinen Form beteiligten Komponenten zu analysieren und zu systematisieren. Ziel ist es, Komponenten zu erhalten, die in hohem Maße nutzbar für möglichst viele, wenn auch nie für alle, Renderingverfahren sind. Die Komponenten müssen darüber hinaus implizit anpassungsfähig und für spezielle Renderingverfahren erweiterbar sein, so dass die Komponenten es erlauben, die Spezifika jedes Renderingverfahrens auszudrücken.

Ziel dieser Arbeit ist es nicht, ein neues, umfassendes Renderingsystem zu konstruieren, sondern die bestehenden in ihrer Vielfältigkeit zu beschreiben, einzuordnen und in einem generischen Renderingsystem zu integrieren, denn die heutigen Renderingsysteme könnten wegen des gewaltigen Entwicklungsaufwands praktisch nicht neu implementiert werden.

Als zentrale Forderungen an die Software-Architektur computergraphischer Systeme ergeben sich die Verallgemeinerung des Renderingbegriffs und die Verallgemeinerung der Szenenrepräsentation. Dazu ist eine Systematisierung aller am Rendering beteiligten Komponenten notwendig. Ziel ist es insbesondere, Echtzeit-Renderingverfahren, nichtphotorealistische Renderingverfahren und Nichtstandard-Renderingverfahren innerhalb eines generischen Renderingssystems darstellen zu können.

2 ARCHITEKTUR-ANALYSE COMPUTERGRAPHISCHER SYSTEME

Dieses Kapitel analysiert die Software-Architektur computergraphischer Systeme, wobei im Mittelpunkt der Betrachtungen Renderingsysteme stehen. Der erste Abschnitt gibt einen Überblick über Kriterien, mit denen eine Software-Architektur bewertet werden kann. Im zweiten Abschnitt wird der technische und konzeptionelle Aufbau ausgewählter Systeme analysiert. Der dritte Abschnitt enthält eine Evaluation der Software-Architekturen hinsichtlich ihrer Verwendbarkeit und Erweiterbarkeit. Im letzten Abschnitt des Kapitels schließlich werden Forderungen an eine ideale Software-Architektur formuliert, die sich aus dem kritischen Vergleich ergeben.

2.1 Architekturmerkmale computergraphischer Systeme

Die Software-Architektur computergraphischer Systeme kann hinsichtlich ihres technischen Aufbaus, ihres konzeptionellen Aufbaus und ihrer Arbeitsweise untersucht werden. In diesem Abschnitt wird ein Überblick darüber gegeben, wie die Software-Architektur hinsichtlich dieser Merkmale untersucht werden kann.

2.1.1 Merkmale des technischen Aufbaus

Technisch können wir die computergraphische Systeme anhand ihrer Schnittstellen und Systemarchitektur charakterisieren. Hinsichtlich der Schnittstellen können wir folgende Grundtypen computergraphischer Software-Systeme unterscheiden, wobei als Kriterium die Sichtbarkeit der Implementierung hinter der Schnittstelle herangezogen wird [102]:

- *Black-Box-Systeme* sind derart gekapselt, dass keine Details neben der Schnittstelle sichtbar sind; insbesondere ist ihre Implementierung nicht zugänglich. Ihre Funktionalität kann nicht über softwaretechnische Konstrukte, z.B. Ableitung spezialisierter Klassen, erweitert oder modifiziert werden. Die Systeme sind in Anwendungen nur dann integrierbar, wenn technisch auf der Seite des Black-Box-Systems entsprechende Vorkehrungen (z.B. explizites API und linkfähige Bibliothek) getroffen wurden.

- *White-Box-Systeme* kapseln ihre Funktionalität mit einer Schnittstelle, erlauben es jedoch über softwaretechnische Konstrukte ihre Funktionalität zu erweitern und zu modifizieren. Im Fall objektorientierter Software-Architekturen werden dazu meist Vererbung und Templates eingesetzt. Ihre Implementierung kann darüber hinaus ebenfalls zur Verfügung gestellt sein. White-Box-Systeme sind in Anwendungen vollständig integrierbar.
- *Gray-Box-Systeme* stellen kontrolliert einen Teil ihrer Implementierung zur Verfügung. Dieser bereitgestellte Teil kann allerdings auch als Teil der Schnittstelle verstanden werden, so dass der Begriff nicht präzise definiert werden kann [102]. Gray-Box-Systeme, wie sie insbesondere in der hier vorliegenden Analyse verstanden werden, lassen sich dadurch kennzeichnen, dass der verfügbare Teil zum Zweck der Verfeinerung bereitgestellt wird; eine grundsätzliche Erweiterung der Funktionalität ist damit meist nicht möglich.

In Bezug auf die Systemarchitektur können wir zwei Grundtypen computergraphischer Systeme unterscheiden, die einen unterschiedlich abstrakten Zugang bereitstellen:

- *Computergraphische Bibliotheken.* Eine elementare Computergraphik-Bibliothek stellt Konstrukte zur Bildsynthese zur Verfügung und abstrahiert dabei vor allem den Zugriff auf die Hardware. Beispiele für elementare Bibliotheken sind OpenGL und Direct3D. Bibliotheken können daneben Konstrukte zur Modellierung von Szenen bereitstellen.
- *Computergraphische Frameworks.* Ein Framework stellt einen Rahmen bereit, mit dessen Hilfe eine computergraphische Anwendung konstruiert werden kann, indem anwendungsspezifische Datenstrukturen und Algorithmen in den Framework aufgenommen werden.

Die Entwicklungen im Bereich der komponentenorientierten[*] Softwarekonstruktion, die ausführlich bei Szyperski [102] diskutiert werden, lassen sich grundsätzlich auf den Bereich der Computergraphik übertragen. Erste Ansätze für einen komponentenorientierten Entwurf eines computergraphischen Systems finden sich in BOOGA [95] und in Form von interaktiven, animierten 3D-Widgets [26]. In dieser Arbeit steht jedoch der komponentenorientierte Entwurf einer Systemimplementierung nicht im Vordergrund; stattdessen wird hauptsächlich die Komponentenbildung im Sinne der *Identifikation von Grundbausteinen* für Renderingsysteme untersucht.

2.1.2 Merkmale des konzeptionellen Aufbaus

Konzeptionell können wir computergraphische Systeme charakterisieren, indem wir ihr Software-Modell betrachten. Wesentliche Bestandteile des Software-Modells sind:

- *Renderingkomponenten.* Jedes computergraphische System definiert eine Sammlung von Klassen oder Datentypen, deren Instanzen für die Formulierung von Renderingaufgaben verwendet werden. Objektorientierte Renderingkomponenten stehen meist miteinander in Assoziations- oder Aggregationsbeziehung. Zwischen den einzelnen Klassen von Renderingkomponenten kann darüber hinaus ein Spezialisierungs- und Generalisierungsverhältnis bestehen. Renderingkomponenten können auch in Form von Funktionen und Datenstrukturen vorliegen. Spezielle Renderingkomponenten sind *Renderingprimitive:* Sie repräsentieren geometrische Objekte eines Renderingsystems. Weitere Renderingkomponenten sind z.B. graphische Attribute und geometrische Transformationen.

[*] Der Begriff „Komponente“ wird in dieser Arbeit einerseits synonym zum Begriff „Bestandteil“ verwendet, andererseits im Sinne des komponentenorientierten Software-Engineerings. Die Unterscheidung ergibt sich i.A. aus dem Kontext bzw. wird explizit dargelegt.

- *Szenenrepräsentation.* Sie legt fest, wie Renderingkomponenten zu einer Szene kombiniert und miteinander in Bezug gesetzt werden. Hierarchische Repräsentationsformen wie etwa Bäume und gerichtete, azyklische Graphen sind konkrete Typen von Szenenrepräsentationen [9]. Zu ihren Aufgaben zählen die räumliche Anordnung geometrischer Objekte, die Zuordnung optischer und weiterer graphischer Attribute und die Spezifikation von geometrischen Transformationen.
- *Rendering-Pipeline.* Sie beschreibt den technischen Ablauf der Bildgenerierung. Die Renderingkomponenten, die in einer Szenenbeschreibung enthalten sind, müssen dazu geeignet interpretiert werden. Rendering-Pipelines gliedern sich in Stufen; Renderingkomponenten werden beim Stufenübergang jeweils konvertiert bzw. transformiert.

2.1.2.1 Typen der Szenenrepräsentation

Der Typ der Szenenrepräsentation ist ein entscheidendes Merkmal der Software-Architektur eines computergraphischen Systems, denn die Szenenmodellierung stellt eine Hauptfunktionalität dar. Zwei Grundtypen von Szenenrepräsentationen, eine hierarchische und eine sequentielle Szenenrepräsentation, lassen sich unterscheiden.

- *Hierarchische Szenenrepräsentation.* Der Szenengraph, ein gerichteter azyklischer Graph, ist die häufigste Form der Szenenrepräsentation in computergraphischen Systemen (z.B. MAM, Java3D, OpenInventor). Blattknoten repräsentieren Szenengeometrien, während innere Knoten graphische Attribute, geometrische Transformationen und Subgraphen verwalten. Der Szenengraph unterstützt die hierarchische geometrische Modellierung und den Ausdruck gemeinsamer graphischer Attribute. Während einer Traversierung wird insbesondere die Kohärenz in derAttributierung und Transformation der Szenenobjekte ausgenutzt, die zur Optimierung des Renderings wesentlich beitragen. Allerdings muss der Szenengraph als Modellierungsstruktur unterschieden werden von der Struktur der internen Objektdarstellung. Ein Ray-Tracing-System benötigt eine räumliche Zugriffsstruktur, in der alle Objekte enthalten sind. Ein Szenengraph wäre in diesem Fall nur ein Mittel zur Konstruktion einer internen Darstellung, die alle Szenenobjekte zusammen mit ihren Attributen enthält. Erfolgt die hierarchische Modellierung auf der Basis einer Baumstruktur, kann die gemeinsame Nutzung von Submodellen nicht direkt ausgedrückt werden; die multiple Nutzung von Szenen-Teilgraphen ist jedoch eine wesentliche Eigenschaft, da sie eine kompakte Spezifikation des Gleichartigen in komplexen Szenen ermöglicht.
- *Sequentielle Szenenrepräsentation.* Alternativ zum Szenengraph wird in einigen Systemen (z.B. Radiance, POV-Ray) eine Liste von Szenenobjekten verwaltet. Jedes Szenenobjekt kennt vollständig seine graphischen Attribute und seine geometrischen Transformationen. Die Editierung einzelner Szenenobjekte gestaltet sich aufgrund dieser Struktur einfacher, denn für jedes Objekt sind alle auf ihn einwirkenden Parameter direkt ersichtlich im Gegensatz zum Szenengraphen, bei dem für ein Objekt erst zum Zeitpunkt seiner Traversierung festgestellt werden kann, welche graphischen Attribute und welche geometrischen Transformationen auf das Objekt einwirken.

Als universeller Typ von Szenenrepräsentation hat sich in heutigen computergraphischen Systemen der Szenengraph durchgesetzt. Der Szenengraph erlaubt es, kompakt Szenen zu modellieren, Gemeinsamkeiten in der Attributierung und in den geometrischen Transformationen auszudrücken und kann direkt durch eine Benutzerschnittstelle editiert werden.

2.1.2.2 Traversierung der Szenenrepräsentation

Evaluation auf der Basis einer Traversierungsschnittstelle. Die Basisklasse aller Szenengraphknoten definiert die Traversierungsschnittstelle, die alle Klassen von Szenengraphknoten implementieren müssen. Wird das System erweitert, muss diese Schnittstelle von den neu entworfenen Szenengraphklassen implementiert werden; neu eingefügte Methoden in diesen Klassen können von der Evaluationsstrategie nicht berücksichtigt werden, da sie der Schnittstelle nicht bekannt sind.

Evaluation durch Visitor-Objekte. Das Visitor-Entwurfsmuster [34] kapselt die Auswertungsstrategie in sog. Visitor-Objekten. Zur Evaluation wird ein Visitor-Objekt durch die hierarchischen Strukturen durchgereicht. Dazu definiert die Basisklasse aller Szenengraphknoten eine Traversierungsmethode, die als Argument ein Visitor-Objekt besitzt. „Besucht“ ein Visitor-Objekt einen Knoten, dann wird die Auswertungsmethode des Visitor-Objekts aufgerufen und der momentane Knoten als Argument übergeben. Es können drei Implementierungsvarianten unterschieden werden.

1. Die Visitor-Basisklasse kennt alle Klassen von auszuwertenden Renderingkomponenten und legt für jede Klasse eine einzelne Auswertungsmethode in ihrer Schnittstelle fest.
2. Die Visitor-Basisklasse definiert eine einzige Auswertungsmethode für allgemeine Renderingkomponenten; konkrete Visitor-Klassen implementieren ihre typspezifische Funktionalität dadurch, dass sie die Auswertungsmethode überschreiben und durch dynamische Typermittlung (Laufzeittypinformation, RTTI) Klassen von Renderingkomponenten unterscheiden können. Zur Implementierung können eigene Sprungtabellen mit Methode-Objekt-Zeigern zum Einsatz gelangen.
3. Die Visitor-Klasse kann auch auf der Basis von Multidispatching, das den Prozess der Methodenselektion aufgrund mehrerer polymorpher Argumente bezeichnet, implementiert werden, sofern dies die Implementierungssprache unterstützt.

2.1.3 Immediate-Mode- und Retained-Mode-Systeme

Ein weiteres Merkmal für die Software-Architektur computergraphischer Systeme ist die Art, wie Renderingkomponenten durch das Renderingsystem verarbeitet werden.

- *Immediate-Mode-Renderingsysteme.* Diese Renderingsysteme verarbeiten die ihnen mitgeteilten Renderingkomponenten unmittelbar, ohne sie in einer eigenen Container-Struktur zu speichern. Wenn ein Bild generiert wird, müssen die notwendigen Renderingkomponenten dem Renderingsystem mitgeteilt werden. Dadurch können insbesondere dynamische Szenen effizient gerendert werden, da die Erzeugung, Auswahl und Modifikation von Renderingkomponenten während der Übermittlung vorgenommen werden kann. Klassische Vertreter von Immediate-Mode-Systemen sind im Bereich der 3D-Graphik OpenGL und Direct3D; im Bereich der 2D-Graphik ist PostScript zu nennen.
- *Retained-Mode-Renderingsysteme.* Diese Renderingsysteme verarbeiten die ihnen mitgeteilten Renderingkomponenten, indem sie die Komponenten zunächst in einer Container-Struktur speichern und von dort nach Anforderung verarbeiten. Diese Renderingsysteme stellen Methoden bereit, um die in einer Struktur enthaltenen Renderingkomponenten zu editieren. Weiter zeichnen sie sich durch die Möglichkeit aus, die interne Darstellung und damit das Rendering zu optimieren, z.B. durch Sortierung der geometrischen Objekte nach gleichen Attributen. Die globale Szeneninformation ermöglicht die Bereitstellung von Selektionsfunktionalität und unterstützt die Implementierung von Renderingverfahren mit

globalen Illuminationsmodellen. Klassische Vertreter von Retained-Mode-Renderingsystemen sind PHIGS [79] und HOOPS [113].

Lange Zeit die Frage ungeklärt, welcher der beiden Ansätze für computergraphische Systeme der bessere sei. Ausdrücklich zeigte sich dies an der Konkurrenz zweier exponierter Vertreter dieser Ansätze, PEX und OpenGL, im Zusammenhang mit dem Wunsch nach einer standardisierten elementaren 3D-Graphikschnittstelle [32]. Die Entwicklung im Bereich der Computergraphik-Hardware hat gezeigt, dass ein hybrider Ansatz optimale Ergebnisse liefert. Im Fall von OpenGL können nahezu alle Befehle, einschließlich geometrischer Transformationen und Beleuchtungsberechnungen, von spezieller Hardware ausgeführt werden. Damit wird OpenGL mit seiner großen Ausdruckskraft, als klassischer Vertreter eines Immediate-Mode-Renderingsystems, optimal unterstützt. Der deklarative Ansatz der Retained-Mode-Renderingsysteme besitzt diese Ausdruckskraft im Allgemeinen nicht: Es können keine neuen bildgebenden Verfahren implementiert werden, da (zueinander orthogonale) elementare Renderingbefehle nicht zur Verfügung stehen, wie die Untersuchungen von Segal und Akeley gezeigt haben [89]. Eine kompakte Retained-Mode-Darstellung von statischen Renderingkommandofolgen wird jedoch in OpenGL durch Displaylisten ermöglicht, die allerdings aus Effizienzgründen nicht editierbar sind. Insgesamt lassen sich so die Vorteile beider Darstellungsarten zumindest im Fall von OpenGL in einer Anwendung kombinieren.

Die Anforderungen, die an eine Szenenspezifikationsform gestellt werden, sind meist anwendungsabhängig. Zum Beispiel benötigt eine Virtual-Reality-Anwendung Culling- und Occlusion-Techniken, die vom Szenengraph unterstützt werden müssen (z.B. Szenengraph-Culling im OpenGL Optimizer [91]), wohingegen eine Anwendung zur interaktiven Illustration eher Freiheitsgrade bei der Gestaltung der Renderingverfahren benötigen wird. Immediate-Mode-Systeme haben den Vorteil, dass sie unterschiedliche Szenenspezifikationsformen effizient und leicht implementieren können, für Retained-Mode-Systeme gilt dies nicht.

2.2 Analyse der Software-Architekturen

In diesem Abschnitt werden konkrete Renderingsysteme anhand ihres technischen und konzeptionellen Aufbaus sowie ihrer Arbeitsweise vorgestellt.

Bei der Analyse der Software-Architekturen geht es nicht vordringlich um den *objektorientierten Entwurf der Implementierung*, sondern um den *konzeptionellen Entwurf*. Die in der Analyse illustrierten Objektmodelle sind daher Abstraktionen der implementierten Systeme – die aufgezeigten Objektmodelle verwenden so weit wie möglich eine einheitliche Terminologie, um Strukturen und Muster sichtbar werden zu lassen und den Vergleich zu ermöglichen; die entsprechenden Begriffe, die von den Renderingsystemen gebraucht werden, sind erläuternd angefügt.

Das Design zeigt sich bei objektorientierten Renderingsystemen direkt in der Klassenhierarchie, in anderen Systemen ist es nur indirekt erkennbar und mitunter nicht dokumentiert. Das Modell der Renderingkomponenten ist jedoch unabhängig von der technischen Realisierung: Wesentlich ist, dass das *Objektmodell* nicht nur datentechnisch in Erscheinung tritt, sondern zugleich während der Modellierung im Anwender entsteht und dort als *mentales Modell* die Arbeitsweise mit dem Renderingsystem prägt.

Die Analyse bezieht bewusst klassische, nicht-objektorientierte Renderingsysteme mit in die Untersuchung ein. Das Paradigma der Objektorientierung hat seit seinen Anfangsjahren in der Computergraphik eine bedeutende Rolle gespielt [116]. Es wiederspiegelt nicht nur den Stand der Technik, insbesondere des Software-Engineerings, sondern lässt sich fast „natürlich“ auf

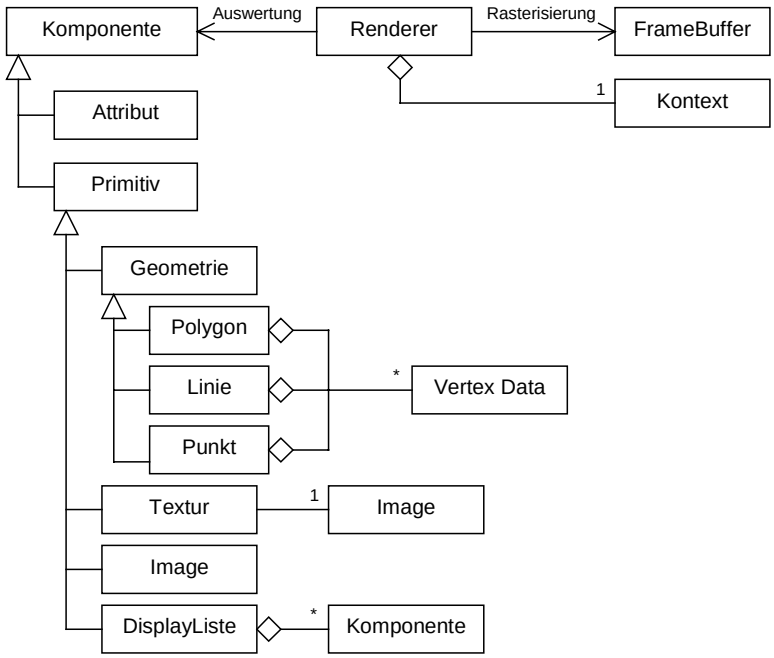

Abbildung 4. Konzeptionelles Objektmodell von OpenGL.

viele Fragestellungen der Computergraphik übertragen: Objektorientierung ermöglicht einen meist intuitiven Ausdruck von computergraphischen Konzepten. Allerdings ist zu beobachten, dass bislang kein vollständig objektorientiertes Renderingsystem auch nur annähernd die Popularität erlangt hat oder die technische Leistungsfähigkeit besitzt wie klassische, prozedural implementierte Renderingsysteme. Alle Versuche, durch objektorientierte Ansätze effiziente und effektive Renderingsysteme zu bauen, hatten einen zwar wissenschaftlichen, jedoch hinsichtlich der Verbreitung begrenzten Erfolg.

2.2.1 Echtzeit-Renderingsysteme

Echtzeit-Renderingsysteme werden dort eingesetzt, wo interaktive, animierte Computergraphik in Anwendungen benötigt wird. Sie orientieren sich dabei meist an der Rendering-Pipeline, die durch die Computergraphik-Hardware geprägt ist. Wir unterscheiden hier zwischen elementaren Renderingsystemen wie OpenGL und Direct3D, und darauf basierenden „high-level“ Graphikbibliotheken wie OpenInventor oder Java3D. Die Analyse in diesem Abschnitt wird für OpenGL und OpenInventor durchgeführt. Der Schwerpunkt der Untersuchungen liegt auf den Renderingkomponenten; die Integration in eine Benutzerschnittstelle sowie interaktive Fähigkeiten werden nur am Rande betrachtet.

2.2.1.1 OpenGL

OpenGL [108] ist eine prozedurale Renderingbibliothek für interaktive 2D- und 3D-Graphik und gilt aufgrund seiner weiten Verbreitung und seiner Unterstützung durch verschiedene Hardware als Industriestandard für low-level Echtzeit-Rendering. OpenGL besitzt ist ein Black-Box-System und Immediate-Mode-System. Die Bibliothek stellt Renderingkomponenten in Form von Funktionen und Datentypen bereit, die eine direkte und einfache Kontrolle über alle fundamentalen Graphikoperationen gewährleisten, und ist mit einer sprachunabhängigen, prozeduralen Anwendungsprogrammierschnittstelle (API) ausgestattet.

Das konzeptionelle Objektmodell von OpenGL (siehe Abbildung 4) besteht aus Typen von Renderingprimitiven und Renderingattributen, der Verarbeitungseinheit (Renderer), dessen

Zustand in einem Kontext verwaltet wird und der einen Framebuffer zur Ausgabe bzw. zur Eingabe nutzt. An Renderingprimitiven werden geometrische Objekte und Texturen [32] bereitgestellt. An geometrischen Objekten werden Punkte, Linien und Polygone bereitgestellt; sie werden durch Vertex-Data (d.h. Vertex-Koordinaten, Vertex-Normalen, Vertex-Farben und Vertex-Texturkoordinaten) definiert. OpenGL unterscheidet spezielle Typen konvexer Polygone, nämlich Dreiecke, Dreiecksstreifen, Dreiecksfächer, Vierecke und Viereckснetze, um den höchstmöglichen Grad an Kompaktheit in der Beschreibung zu erreichen. Konzeptionell sind weiter Lichtquellen, geometrische Transformationen und graphische Attribute definiert. Sequenzen von Renderingkommandos können in sog. Displaylisten kompiliert und in der Graphik-Hardware vorgehalten werden.

Eine explizite Szenenrepräsentation stellt OpenGL nicht bereit, denn Renderingprimitive werden quasi algorithmisch mit Hilfe von Renderingkommandos spezifiziert, nicht jedoch in Objektform. OpenGL verwirklicht damit die Idee, mit Hilfe einer feinkörnig faktorisierten „Graphik-Assemblersprache“ ein universell einsetzbares Renderingwerkzeug bereitzustellen [89]. Verschiedene Szenenrepräsentationsformen sind gleichermaßen effizient auf der Basis von OpenGL implementierbar; Beispiele hierfür sind der Szenengraph der OpenInventor-Bibliothek und der Szenengraph im OpenGL Optimizer.

Die Rendering-Pipeline von OpenGL unterscheidet geometriebasierte und pixelbasierte Renderingdaten; Daten beider Varianten können direkt durch Kommandos oder in kompilierter Form aus Displaylisten aufgerufen werden. Geometrische Objekte bzw. Pixeldaten werden in einer Rasterisierungsstufe in den Framebuffer übertragen. Die Einstellungen, mit denen die Arbeitsweise der Rendering-Pipeline kontrolliert werden kann, werden durch den OpenGL-Kontext verwaltet, ebenso die Verwendung des Framebuffers. Der Framebuffer repräsentiert ein Aggregat, je nach Konfiguration, bestehend aus Color-Buffer, Depth-Buffer, Stencil-Buffer und Accumulation-Buffer. Restriktionen in den Renderingeigenschaften, die sich aus der Echtzeit-Natur ergeben, sind z.B. ein lokales Beleuchtungsmodell, z-Buffer-basiertes Entfernen verdeckter Linien und Flächen, Verzicht auf komplexe geometrische Objekte und das Fehlen globaler Information über eine dargestellte Szene.

OpenGL wurde für Echtzeit-Rendering entworfen. Insbesondere mit der gestiegenen Unterstützung des Texturierungsmechanismus und der Framebuffer-Operationen durch computergraphische Hardware, sofern sie OpenGL direkt unterstützt, lassen sich jedoch zunehmend komplexe photorealistische und photosurrealistische Beleuchtungseffekte und Gestaltungstechniken realisieren. Beispiele hierfür sind z.B. Phong-Schattierung von Oberflächen [21], Echtzeit-Schatten und –Reflexion [53], Cube-Environment-Mapping und Bump-Mapping [52].

> Das Objektmodell von OpenGL stellt polygonale Objekte und Texturen als grundlegende graphische Primitive bereit, deren Erscheinung und Auswirkung durch eine Vielzahl im Kontext verwalteter Attribute kontrolliert wird. OpenGL definiert keine bestimmte Form der Szenenrepräsentation, ermöglicht aber die effiziente Implementierung von Szenengraphen. Es bietet eine detaillierte Kontrolle seiner Rendering-Pipeline an und kann als Graphik-Assembler zur Realisierung unterschiedlicher Echtzeit-Renderingverfahren eingesetzt werden. Als Black-Box-Renderingsystem ist OpenGL nicht im Kern erweiterungsfähig. In Anwendungen kann die OpenGL-Bibliothek vollständig integriert werden.

2.2.1.2 OpenInventor

OpenInventor [101] ist eine objektorientierte 2D- und 3D-Graphikbibliothek, die zum Bau interaktiver Anwendungen verwendet wird und auf dem Renderingsystem OpenGL basiert.

OpenInventor ist ein Gray-Box- und Immediate-Mode-System; es wurde in C++ implementiert, jedoch sind die Sourcen nicht zugänglich. OpenInventor stellt eine Sammlung von Klassen bereit, die Szenenobjekte, Szenengraphen und Interaktionstechniken modellieren. Der OpenInventor-Szenengraph erreichte große Verbreitung und findet sich heute in abgewandelter und erweiterter Form z.B. in VRML und MPEG-7.

Das Objektmodell von OpenInventor (siehe Abbildung 5) definiert eine Reihe von Klassen von Szenengraphknoten, die grob in Gruppen-Knoten, Property-Knoten (Attribut-Knoten) und Shape-Knoten (Geometrie-Knoten) eingeteilt werden. Gruppen-Knoten verwalten eine Menge von Kinderknoten, d.h. Subgraphen. Property-Knoten spezifizieren graphische und geometrische Attribute; sie umfassen Texturen, Zeichenstile, Material und geometrische Transformationen. Shape-Knoten repräsentieren geometrische Objekte und untergliedern sich in analytisch spezifizierte Geometrien (Kegel, Kugel, Torus, Freiformflächen), polygonale Geometrien, und buchstabenbasierte Geometrien. Lichtquellen-Knoten und Kamera-Knoten werden als einfache Knoten implementiert. Jeder Knoten enthält vollständig seine Funktionalität, z.B. im Fall von Shape-Knoten die Umsetzung in OpenGL-Primitive.

Alle Knoten exportieren ausgewählte Objektattribute in Form von sog. Fields. Eine Anwendung manipuliert einen OpenInventor-Knoten mit Hilfe sog. Sensoren, die bestimmte Fields modifizieren, z.B. zum Zweck der Animation. Manipulatoren sind spezielle Objekte, die Interaktionstechniken bereitstellen [112]. Auf die Konzepte zur Programmierung interaktiver, animierter Anwendungen wird im Rahmen dieser Darstellung nicht eingegangen.

Szenen werden mit OpenInventor durch gerichtete, azyklische Szenengraphen modelliert. Das Rendering erfolgt durch Traversierung des Szenengraphen. Die Knoten repräsentieren Renderingkomponenten, wobei Knoten mehrfach referenziert sein dürfen. Shapes, graphische Attribute und geometrische Transformationen werden gleichwertig als Szenengraphknoten modelliert; eine hohe Ausdruckskraft in der Szenenspezifikation ist damit möglich, da nur gezielt dort, wo eine Kontext-Änderung notwendig ist, ein entsprechender Knoten in die Hierar-

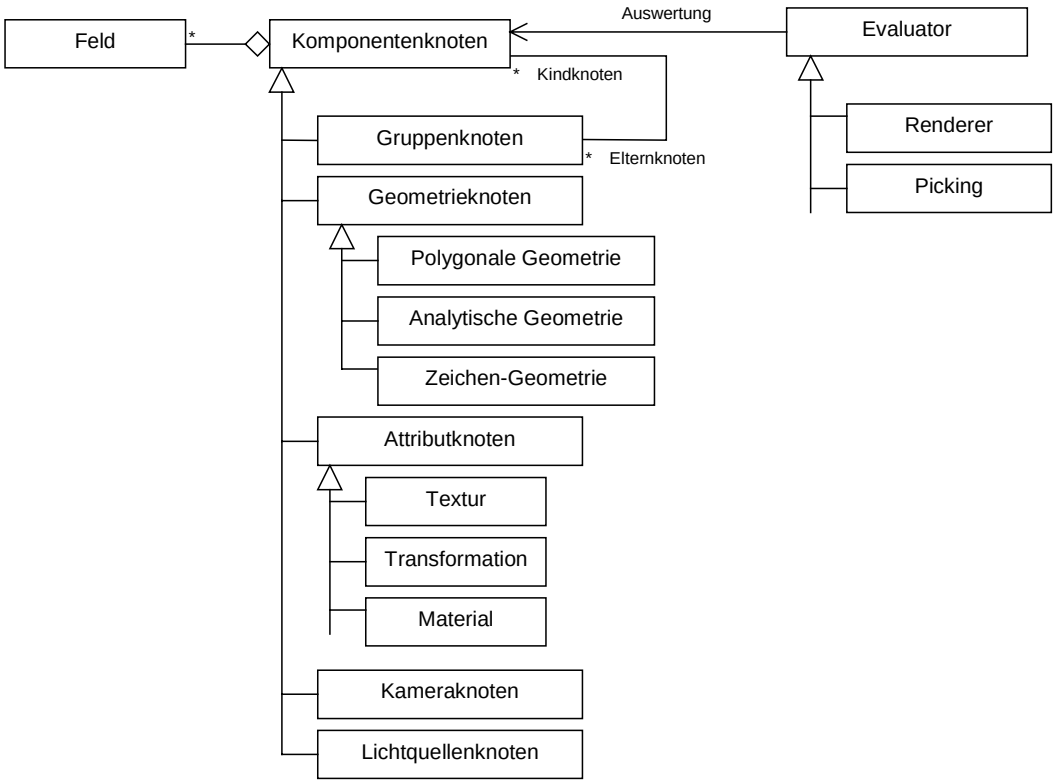

Abbildung 5. Konzeptionelles Objektmodell von OpenInventor.

chie einfügt werden muss.

Die Anordnung der Knoten orientiert sich an der Rendering-Pipeline von OpenGL, d.h. die Knoten werden während der Pre-Order-Traversierung ausgewertet und in OpenGL-Kommandos umgesetzt. Der OpenInventor-Szenengraph ist somit eine Struktur, die die visuelle Programmierung von OpenGL ermöglicht, nicht jedoch eine Szenen-Spezifikation auf vergleichbarem abstrakten Niveau wie im Fall von RenderMan. Die Flexibilität ist zugleich ein häufig in der Literatur (z.B. [93][9]) genannter Kritikpunkt: die Konstruktion von OpenInventor-Szenengraphen setzt detaillierte Kenntnisse der Arbeitsweise von OpenGL voraus, andererseits ist dadurch eine effiziente Nutzung von OpenInventor ohne Geschwindigkeitseinbussen im Vergleich zu einer direkten OpenGL-Programmierung möglich.

OpenInventor stellt keine Mechanismen bereit, um andere Renderingsysteme außer OpenGL anzusteuern. Auch können für OpenGL entwickelte Renderingverfahren, die auf Multi-Pass-Rendering beruhen (z.B. Echtzeit-Schatten und -Reflexion), nicht in OpenInventor genutzt werden.

> Das Objektmodell von OpenInventor unterscheidet Gruppierungskonstrukte, geometrische Objekte, graphische und geometrische Attribute, Lichtquellen und Kameras. Sie werden als gleichwertige Elemente einer Szenenbeschreibung behandelt und erzielen damit eine hohe Flexibilität und Ausdruckskraft in der Szenenspezifikation. Die Szenenmodellierung erfolgt in Form gerichteter, azyklischer Szenengraphen. Renderingkomponenten sind in OpenInventor mit Szenengraphknoten gleichgesetzt. OpenInventor ist auf die Rendering-Pipeline von OpenGL ausgerichtet und optimiert. Als Gray-Box-System kann OpenInventor durch objektorientierte Mechanismen erweitert und modifiziert werden. In Anwendungen lässt sich diese Klassenbibliothek vollständig integrieren.

2.2.1.3 HOOPS

HOOPS [113], das hierarchische, objektorientierte Programmiersystem, ist eine 3D-Graphikbibliothek, die zur Entwicklung interaktiver Anwendungen dient. HOOPS ist ein Black-Box- und Retained-Mode-System, das objektbasiert, aber nicht in einer objektorientierten Programmiersprache implementiert. HOOPS führt das Echtzeit-Rendering je nach Verfügbarkeit mit einer Reihe von low-level Renderingsystemen (z.B. OpenGL, Xlib) aus. HOOPS besteht aus zwei logischen Komponenten, einer Datenbank zur Verwaltung graphischer Objekte und einer abstrakten Geräteschnittstelle.

Das objektbasierte Modell von HOOPS (siehe Abbildung 6) definiert geometrische 2D- und 3D-Objekte und Attribute. HOOPS-Attribute werden unterteilt in Display-Attribute für die graphische Gestaltung geometrischer Objekte, Modellierungs- und Sichtattribute zur Festlegung der geometrischen Transformationen, der Projektionen und der Ausgabefenster, Selektionsattribute zur Spezifikation interaktiven Pickings, Treiberattribute zur Spezifikation gerätespezifischer Optionen und Heuristikattribute zur Optimierung des Renderings. Die Attributsammlung geht somit über rein graphische Attribute hinaus.

Szenen werden in HOOPS hierarchisch in einer Baumstruktur (Segmentbaum) dargestellt; Knoten (Segmente) sind Container für geometrische Objekte, Attribute und Subsegmente. Segmentbäume unterstützen die Vererbung von Attributen; Segmentbäume ermöglichen außerdem die Sperre von Attributwerten und die nachträgliche Überschreibung von Attributwerten in Subgraphen. Ein Segmentbaum kann nicht mehrfach referenziert werden, nur ausgewählte geometrische Objekte (die in Segmenten enthalten sind) lassen sich per Referenz in Segmente einfügen. HOOPS optimiert Segmentbäume, indem geometrische Objekte mit gleichen Attri-

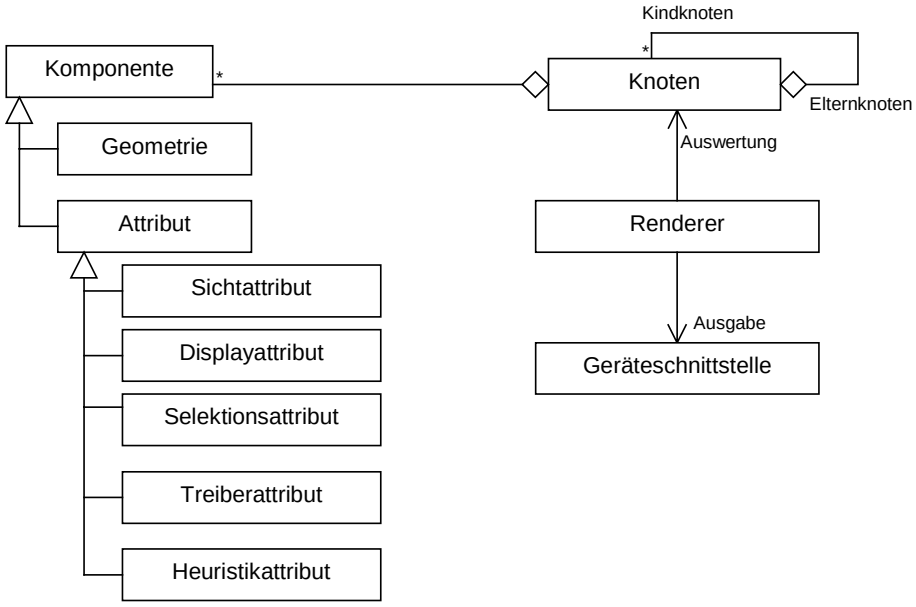

Abbildung 6. Konzeptionelles Objektmodell von HOOPS.

buten gebündelt werden, um die Anzahl der Kontextänderungen bei der Rendering-Traversierung zu minimieren. Ein Segmentbaum ist die datentechnische Repräsentation einer Sammlung von HOOPS-Objekten, die in einer Datenbank abgelegt werden können. Die einzelnen Objekte lassen sich durch Angabe des Zugriffspfades eindeutig lokalisieren.

Segmentbäume sind unabhängig von der Rendering-Pipeline des zugrundeliegenden Renderingsystems. Während der Traversierung eines Segmentbaumes werden geometrische Objekte und Attribute mit Hilfe der Kommandos der abstrakten Geräteschnittstelle ausgewertet. HOOPS implementiert zur Optimierung der Rendering-Geschwindigkeit spezielle Traversierungsformen, die geometrische Objekte selektiv neu zeichnen bzw. inkrementell einfügen, ohne dabei das Gesamtbild neu generieren zu müssen. Diese einheitliche Schnittstelle ermöglicht es, unterschiedliche 2D- und 3D-Renderingsysteme anzusteuern. Auch Ausgaben in Datei- und Druckformaten werden bereitgestellt.

HOOPS dient zum Entwurf komplexer, datenbankbasierter 3D-Graphikanwendungen insbesondere im CAD/CAM-Bereich. Seine Funktionalität legt den Schwerpunkt auf eine hierarchische Szenenmodellierung.

> Das Objektmodell von HOOPS unterscheidet graphische Objekte und Segmente. Graphische Objekte beinhalten geometrische Objekte und Attribute; die Attribute umfassen sowohl graphische Attribute als auch nichtgraphische Attribute. Segmentbäume ermöglichen die hierarchische Szenenmodellierung. HOOPS unterstützt die Abbildung von Szenen auf unterschiedliche Ausgabegeräte und Ausgabeformate. Ein direkter Einfluss auf die Rendering-Pipeline ist nicht gegeben. Als Black-Box-System kann HOOPS nicht erweitert oder modifiziert werden. Die Graphikbibliothek lässt sich in Anwendungen vollständig integrieren.

2.2.1.4 Weitere Systeme

Direct3D [94] ist ein Echtzeit-Renderingsystem von Microsoft, das speziell für die PC-Plattform entwickelt wurde. Direct3D verfügt sowohl über einen Retained-Mode als auch über einen Immediate-Mode. Seine Fähigkeiten und seine Programmierschnittstelle sind äquivalent zu OpenGL. Direct3D ist insbesondere auf die Spiele-Entwicklung ausgerichtet. Die Software-

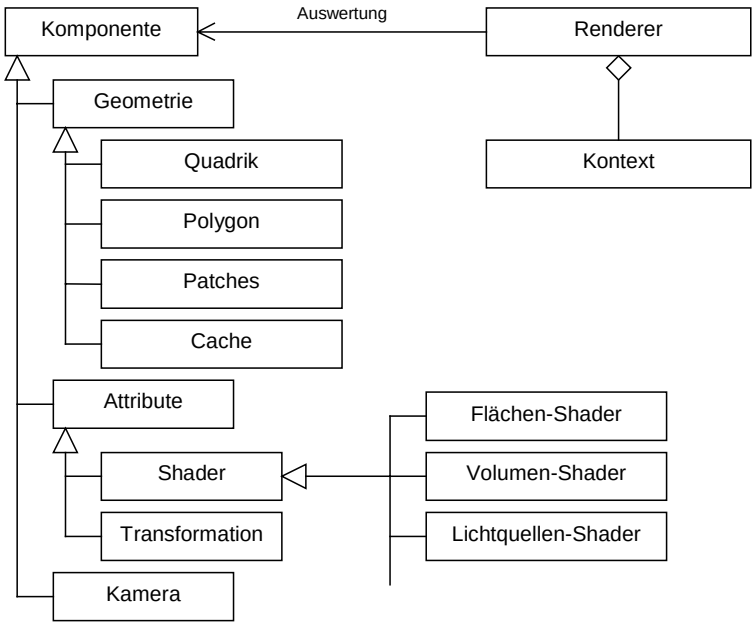

Abbildung 7. Konzeptionelles Objektmodell von RenderMan.

Architektur wird hier nicht näher untersucht, da sie in wesentlichen Grundzügen mit denen von OpenGL übereinstimmt.

Heidi [5] ist eine von Autodesk entwickelte portable, objektbasierte Schnittstelle für Echtzeit-Rendering. Es abstrahiert das konkret zugrundeliegende Renderingsystem durch Definition einer einheitlichen Schnittstelle, die je nach Verfügbarkeit und Anforderungen von unterschiedlichen Renderern implementiert wird. Derzeit werden OpenGL und Direct3D unterstützt. Obwohl eine Kapselung durch Objekte besteht, ist keine weitergehende Organisationsform für Szenen verfügbar; dadurch entsteht im Vergleich zu einer direkten Programmierung mit OpenGL oder Direct3D ein Overhead. Die Software-Architektur von Heidi wird hier nicht näher untersucht, da sie mit der Software-Architektur von HOOPS vergleichbar ist.

2.2.2 Photorealistische Renderingsysteme

Photorealistischen Renderingsystemen liegen Renderingverfahren zugrunde, die globale Beleuchtungsberechnungen ausführen. Anders als im Fall von Echtzeit-Renderingsystemen stehen polygonale Modelle und eine einheitlich auf Hardware direkt abbildbare Rendering-Pipeline derzeit nicht zur Verfügung. Die Beleuchtungsmodelle sind mitunter grundverschieden, z.B. im Fall von Radiosity und Ray-Tracing, und ihre Kombination ist technisch komplex. Das jeweilige Renderingverfahren prägt das Objektmodell: Was im Echtzeit-Rendering nur mit ganz anderen Mitteln darstellbar ist, lässt sich im Fall dieser Systeme meist einfach ausdrücken, z.B. Bump-Mapping durch Oberflächenstörung und die Darstellung impliziter Flächen im Fall von Ray-Tracing. Stellvertretend für diese Kategorie von Renderingsystemen betrachten wir hier RenderMan, Vision und POV-Ray.

2.2.2.1 RenderMan

RenderMan [104] definiert eine Szenenbeschreibungssprache, die insbesondere zur Erzeugung photorealistischer und photosurrealistischer Bilder [3] Einsatz findet. Szenen werden in Skriptform hierarchisch modelliert. Die Stärke von RenderMan liegt insbesondere bei der graphischen Gestaltung von Szenenobjekten. Dazu definiert RenderMan eine Shading-Sprache, die es dem Nutzer ermöglicht, anwendungsspezifische Schattierungs- und Beleuchtungsverfahren zu

programmieren und in die RenderMan-Szenenbeschreibung zu integrieren. Bilder von RenderMan-Szenen werden dadurch erzeugt, dass die Szenenbeschreibung von einer RenderMan-Implementierung interpretiert wird.

Das Objektmodell (siehe Abbildung 7) unterscheidet geometrische Objekte (Shapes), Attribute und Kameras. Geometrische Objekte werden unterteilt in Quadriken, Polygone und parametrische Flächen. Geometrische Objekte eines Typs lassen sich in einem Cache ablegen und können über einen Identifier in der Szenenbeschreibung mehrfach referenziert werden (Object Instancing). Attribute umfassen Transformationen und Shader, die weiter in Flächen-Shader, Volumen-Shader, Lichtquellen-Shader, Transformations-Shader, Displacement-Shader und Imaging-Shader unterteilt werden. RenderMan definiert eine eigene Shader-Sprache; ein Compiler übersetzt einzelne Shader in binäre Module, die bei der Bildgenerierung von der RenderMan-Implementierung dynamisch nachgeladen werden.

Die Szenenmodellierung erfolgt hierarchisch; eine Szenenbeschreibung wird intern durch einen Szenenbaum repräsentiert, der nicht explizit, d.h. durch den Nutzer, modelliert werden kann. Im Gegensatz zu einem gerichteten, azyklischen Graphen als Szenenrepräsentationsform sind somit keine Mehrfachreferzierungen von Subgraphen möglich. Allerdings verfügt RenderMan über das *Primitive Instancing*, das es erlaubt, geometrische Objekte zu definieren, die mittels eines Identifizierers mehrfach in der Szenenbeschreibung referenziert werden können. Graphische Attribute werden vom Primitive Instancing nicht berücksichtigt, d.h. es können ausschließlich geometrische Objekte damit gekapselt werden.

RenderMan separiert strikt die Aufgaben des Renderings von denen der Modellierung. Die Szenenbeschreibungssprache ermöglicht die Modellierung von Szenen, definiert aber kein Renderingverfahren. Es ist Aufgabe einer RenderMan-Implementierung, Modelle mit Hilfe eines bestimmten Renderingverfahrens in Bilder umzusetzen. Ein expliziter Zugang zu den Renderingverfahren sowie zu den Szenenobjekten besteht nicht. Aufgrund der Shader-Sprache ist RenderMan partiell programmierbar.

> Das Objektmodell von RenderMan definiert geometrische Objekte, graphische Attribute und Kameras. Graphische Attribute umfassen Transformationen und Shader, die die Erscheinung von Objekten definieren. Da Shader anwendungsspezifisch mit Hilfe einer Shader-Sprache programmiert, kompiliert und in Szenenbeschreibungen eingebunden werden können, lässt sich eine RenderMan-Implementierung partiell erweitern. Die Szenenbeschreibungssprache ist universell einsetzbar und macht keine Annahmen über das verwendete Renderingverfahren. RenderMan kann als Gray-Box-Renderingsystem betrachtet werden, wobei es ausschließlich durch Shader erweitert werden kann. Als externe Renderingsysteme können die derzeitigen RenderMan-Implementierungen nicht in Anwendungen integriert werden.

2.2.2.2 Vision

Im Vision-Projekt [92] wurde eine Software-Architektur für physikalisch-basiertes Rendering entworfen und implementiert. Vision zieht als Kriterium für die objektorientierte Zerlegung die physikalische Beschreibung der Lichtausbreitung heran und wurde als Framework zur konsistenten Implementierung globaler Beleuchtungsmodelle eingesetzt. Das System ist nicht frei verfügbar; grundsätzlich wäre Vision als White-Box-System oder Gray-Box-System denkbar.

Vision besteht aus logischen Subsystemen für Szenenmodellierung, Rendering und globale Beleuchtung. Das Objektmodell der Szenenmodellierung (siehe Abbildung 8) unterscheidet geometrische Objekte und Attribute. Attribute werden unabhängig von geometrischen Objekten modelliert und werden grob in geometrische Transformationen und optische Attribute einge-

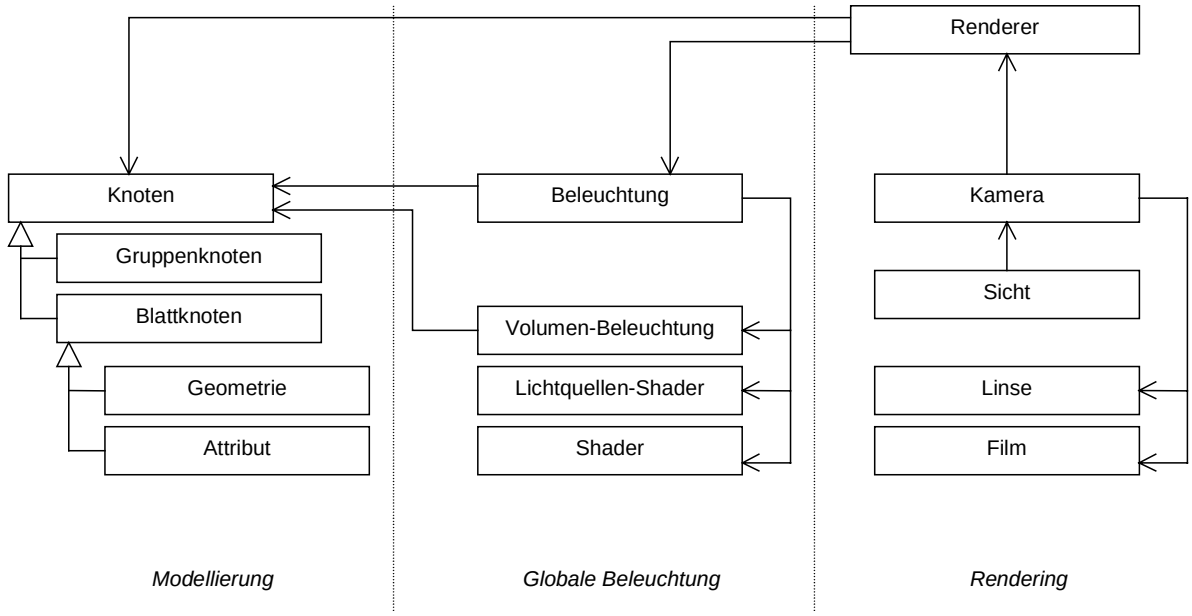

Abbildung 8. Konzeptionelles Objektmodell von Vision.

teilt. Das Objektmodell des Renderingsubsystems besteht aus Klassen für Sichten, Kameras, Linsen und Filme. Das Objektmodell für das Beleuchtungssubsystem definiert Klassen für Lichtquellen-Shader, allgemeine Shader und Volumen-Beleuchtung. Das Shader-Konzept und ihre Programmierung mit Hilfe einer Shader-Sprache orientieren sich an RenderMan.

Die Szenenmodellierung erfolgt hierarchisch durch einen Szenengraphen, dessen Aufgabe es ist, geometrische Objekte mit geometrischen Transformationen und optischen Attributen zu assoziieren. Der Szenengraph ist zugleich zentrale Datenquelle des Rendering- und Beleuchtungssubsystems. Attribute werden dabei nicht auf Geometrien „angewendet“, sondern zusammen mit den Geometrien dem Renderingverfahren zur Verfügung gestellt; die Szenendarstellung ist somit unabhängig von einem bestimmten Renderingverfahren.

Die Rendering-Pipeline ist an den Bedürfnissen globaler Beleuchtungsmodelle ausgerichtet; eine Reihe globaler Beleuchtungsverfahren wurden in Vision implementiert. Bemerkenswert ist, dass globale Beleuchtungsverfahren, die im Allgemeinen als zeitkritisch gelten, auf einer Szenenbeschreibung operieren, die durch einen Szenengraph dargestellt wird. Für die Traversierung, die zur Feststellung der globalen geometrischen Verhältnisse notwendig ist, wurden eine Reihe von Optimierungstechniken entwickelt [93]. Es ist denkbar, dass die Subsysteme für die Modellierung und für das Rendering in Verbindung mit anderen Renderingverfahren genutzt werden könnten, da ihre Konzepte unabhängig von dem konkret verwendeten Renderingverfahren sind.

Das Objektmodell von Vision definiert als Renderingkomponenten geometrische Objekte, Transformationen, optische Attribute sowie renderingbezogene Objekte wie Sicht, Kamera, Linse und Film. Ebenso werden zur Implementierung globaler Beleuchtungsverfahren Klassen bereitgestellt. Vision unterscheidet zwischen den logischen Subsystemen für Rendering, Modellierung und globaler Beleuchtung. Zur Modellierung nutzt Vision einen Szenengraph, der die Renderingkomponenten miteinander in Beziehung setzt. Analog zu RenderMan kann über die Shader-Sprache das System partiell programmiert werden. Vision ist als Framework zur Implementierung globaler Illuminationsmodelle konzipiert. Es könnte, wäre es frei verfügbar, als White-Box- oder Gray-Box-Renderingsystem in Anwendungen integriert oder in seiner Funktionalität erweitert werden.

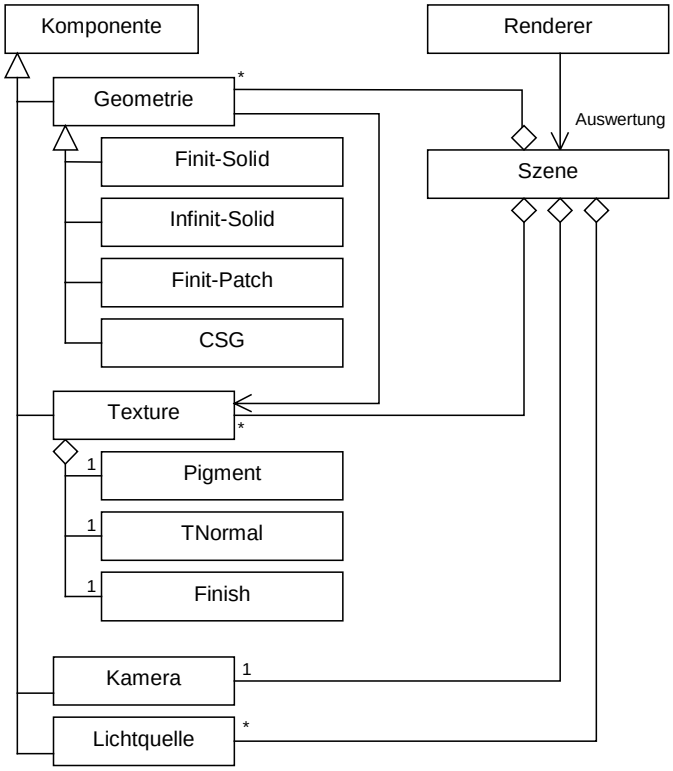

Abbildung 9. Konzeptionelles Objektmodell von POV-Ray.

2.2.2.3 POV-Ray

Das Renderingsystem POV-Ray [78] generiert photorealistische und photosurrealistische Bilder und definiert zur Spezifikation von Szenen eine Szenenbeschreibungssprache. Das System entstand als klassischer Ray-Tracer, der mittlerweile um ein Radiosity-Verfahren zur Berechnung der globalen, diffusen Lichtverteilung einer Szene erweitert wurde. POV-Ray stellt eine Reihe vorgefertigter Schattierungs- und Texturierungsmechanismen bereit, jedoch keine eigenständige Shading-Sprache wie im Fall von RenderMan.

Im Objektmodell von POV-Ray wird ein Bild (Frame) als Sammlung von geometrischen Objekten, Lichtquellen und Texturen definiert. Die Erscheinung eines geometrischen Objektes hängt von der mit ihm direkt assoziierten Textur ab. Eine Textur ist ein komplex aggregiertes Objekt, das durch Oberflächen-Reflexionseigenschaften (Finish), Oberflächennormalen (TNormal), Oberflächenfarben (Pigment) definiert ist. Zusätzlich besteht die Möglichkeit, die Lichtberechnung im Inneren eines Shapes (Interior) und das Medium, durch das Licht geschickt wird (Media), zu spezifizieren. Shapes werden unterteilt in finite Solide, infinite Solide, finite Patches und CSG-Modelle. Es besteht die Möglichkeit, POV-Ray-Objekte vorab zu instanziieren und somit innerhalb einer Szene mehrfach zu verwenden.

Die Szenenmodellierung erfolgt nicht hierarchisch, d.h. eine POV-Ray-Szene ist eine ungeordnete Sequenz von geometrischen Objekten, Lichtquellen und Texturen. Jedes Objekt ist in einem lokalen Koordinatensystem definiert. Für eine Szene werden nach dem Parsing intern räumliche Zugriffsstrukturen zur Beschleunigung der Objekt-Strahl-Intersektionsberechnung aufgebaut (slab-bounding-boxes). Radiosity-Berechnungen können optional durchgeführt werden, auch dazu werden intern entsprechende räumliche Zugriffsstrukturen aufgebaut.

Die Rendering-Pipeline von POV-Ray führt bei der Bildberechnung zunächst ein Preprocessing der Szene zum Aufbau der Zugriffsstrukturen und zur Berechnung der Radiosity durch.

Das Rendern des Bildes erfolgt mit Hilfe von Ray-Tracing. Andere Renderingverfahren, z.B. Echtzeit-Rendering, werden nicht unterstützt.

> Das Objektmodell von POV-Ray definiert geometrische Objekte, Lichtquellen, Kamera und Texturen. Geometrische Objekte sind mit Texturen assoziiert. Eine Szene wird durch eine Liste von geometrischen Objekten und Lichtquellen, die implizit durch eine Szenenbeschreibungssprache aufgebaut werden, repräsentiert; eine hierarchische Szenenmodellierung wird nicht unterstützt. POV-Ray steht als Black-Box-Renderingsystem zur Verfügung und kann in seiner jetzigen Form nicht in Anwendungen integriert werden.

2.2.3 Hybride Renderingsysteme

Die hier vorgestellten hybriden Renderingsysteme haben eines gemeinsam: Sie sind Prototypen universitären Ursprungs und haben im Vergleich zu klassischen Systemen geringe „Marktanteile“. Dabei besitzen sie eine innovative Software-Architektur auf der Grundlage des objektorientierten und komponentenorientierten Software-Entwurfs. Als hybride Renderingsysteme implementieren sie mehrere Renderingverfahren innerhalb eines Systems.

2.2.3.1 BOOGA

Berne's Object-Oriented Graphics Architecture (BOOGA) [95] ist ein computergraphisches System, in dessen Software-Architektur konsequent objektorientierte Technologie eingesetzt wurde. Die Software-Architektur unterscheidet drei Schichten: Die Basisschicht bildet eine Graphikbibliothek, die sowohl Container-Klassen, elementare, geometrische Klassen und Klassen zur Bildspeicherung bereitstellt. Darauf baut eine Framework-Schicht auf, die über geometrische Objekte und Textur-Objekte verfügt. Weiter unterstützt sie die Modellierung mit Szenengraphen. Die oberste Schicht bildet die Komponentenschicht: Die bereitgestellten Komponenten (im Sinne der komponentenorientierten Software-Entwicklung) implementieren komplexe Operationen, z.B. das Parsing von Szenenbeschreibungen und die Darstellungsverfahren von Szenen. Die Software-Architektur von BOOGA kann mit „Komponentenframework“ treffend charakterisiert werden.

Für die hier vorgenommene Analyse ist das Objektmodell von BOOGA nur in der Framework-Schicht von Interesse, denn darin finden sich die für Renderingaufgaben zuständigen Renderingkomponenten. Das Objektmodell (siehe Abbildung 10) unterscheidet BOOGA-Objekte zunächst nach ihrer Dimensionalität. Die jeweiligen Objekte werden weiter unterteilt in geometrische Objekte (Primitive), Gruppenkomponenten (Aggregate), Texturen, Lichtquellen und Kameras. Geometrische Objekte werden direkt von einer Basisklasse für Geometrien durch konkrete Geometrie-Klassen implementiert. Die Erscheinung von geometrischen Objekten wird durch Textur-Klassen definiert; eine Shading-Sprache für prozedurale Texturen ermöglicht die Spezifikation von Texturen; Texturdefinitionen lassen sich zur Laufzeit kompilieren und nachladen. Das konzeptionelle Klassendiagramm vereinfacht das in der Implementierung tatsächlich vorhandene Klassendiagramm insofern, dass Template-Basisklassen, die zur Realisation von 2D- und 3D-Varianten vorhanden sind, zusammengefasst wurden.

Szenen werden in Form von Szenengraphen modelliert; die Renderingkomponenten einer Szene werden in einem sog. World-Objekt zusammengefasst. Drei Grundtypen von Szenengraphknoten lassen sich identifizieren: Blattknoten, die konkrete Renderingkomponenten enthalten; Gruppenknoten (Aggregate), die Renderingkomponenten logisch oder räumlich zusammenfassen, und Verweiskomponenten. Eine Renderingkomponente kann mehrfach in

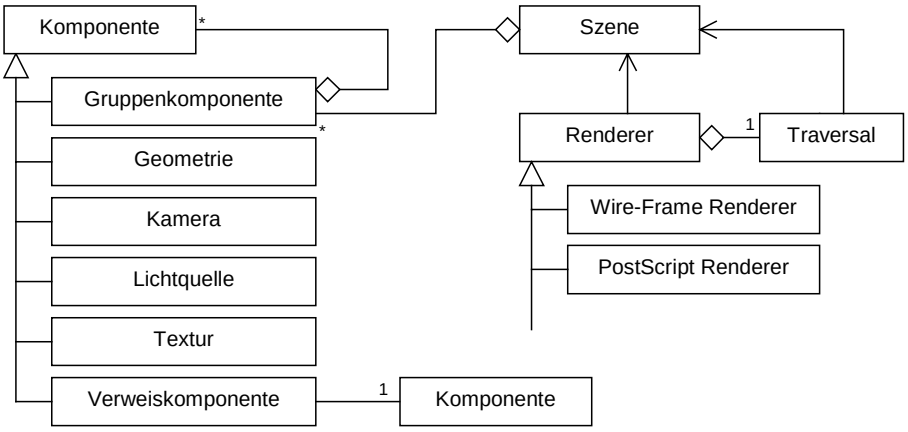

Abbildung 10. Konzeptionelles Objektmodell von BOOGA.

einem Szenengraph genutzt werden, indem jeweils eine Verweiskomponente (Shared-Objekte) auf die ursprüngliche Renderingkomponente einfügt wird; dieser Mechanismus basiert auf dem Brief-Umschlag-Entwurfsmuster [34]. BOOGA definiert eine Szenenmodellierungssprache (BSDL, BOOGA Scene Definition Language), die dadurch erweitert werden kann, dass für einen neuen Renderingprimitivtyp eine Zerlegung in elementare Primitive definiert wird.

Die Rendering-Pipeline wird in BOOGA in Renderer-Komponenten (Component) gekapselt. Ein Renderer wertet einen gegebenen Szenengraphen aus; technisch modelliert BOOGA die Traversierung durch das Visitor-Entwurfsmuster: An ein Besucher-Objekt, das durch den Szenengraphen wandert, wird die Auswertung der Szenengraphknoten delegiert. Als Beispiel-Renderer wurden ein OpenGL-Echtzeit-Renderer [13], ein Wireframe-Renderer und 2D-Renderer für verschiedene Ausgabeformate [95] entwickelt. Da die Szenengraphknoten nicht die Auswertungsstrategie implementieren, können mit Renderer-Objekten beliebige Renderingverfahren in BOOGA auf der Basis eines allgemein nutzbaren Szenengraphen integriert werden.

> Das Objektmodell von BOOGA definiert geometrische Objekte, Aggregate geometrischer Objekte, Texturen, Lichtquellen und Kameras. Eine Szene wird hierarchisch durch einen Szenengraphen repräsentiert. Die Auswertung eines Szenengraphen geschieht durch Renderer, die Szenengraphen traversieren und auswerten. Als objektorientiert implementiertes White-Box-System ist BOOGA vollständig modifizierbar, erweiterbar und in Anwendungen integrierbar.

2.2.3.2 Generic-3D

Generic-3D [8] ist ein computergraphischer Framework und repräsentiert ein generisches Kernsystem, mit dem konkrete computergraphische Systeme konstruiert werden können. Generic-3D definiert dazu eine Sammlung von Schnittstellenklassen und stellt Default-Implementierungen dieser Schnittstellen bereit, ohne dabei Annahmen über den Einsatz und die Kombination dieser Bausteine aufzustellen. Die Klassensammlung von Generic-3D ist entsprechend umfangreich und gekennzeichnet durch Klassen, die einen geringen Funktionsumfang besitzen. Dadurch wird erreicht, dass ein konkretes computergraphisches System, das von Generic-3D „abgeleitet“ wird, gezielt durch die Auswahl, Komposition und Konfiguration entsprechender vorgefertigter Klassen entstehen kann. Ein konkretes computergraphisches System entsteht programmiertechnisch somit durch Assoziation, Vererbung und Template-Instanziierung der Klassen des generischen Kernsystems; die Konfiguration richtet sich nach den Anforderungen, die an das konkrete System gestellt werden. Die von Generic-3D bereitge-

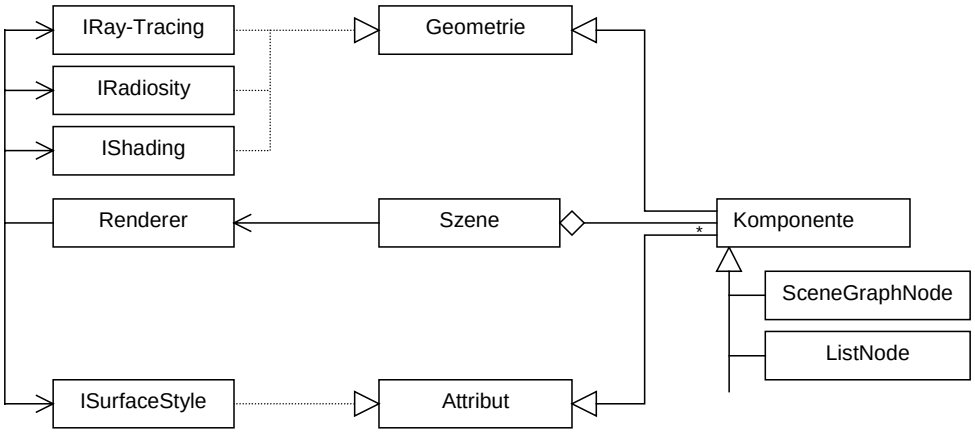

Abbildung 11. Teil des konzeptionellen Objektmodells von Generic-3D.

stellte Funktionalität lässt sich grob in die Kategorien Modellierung, Rendering und Interaktion einteilen.

Das Objektmodell von Generic-3D (siehe Abbildung 11) definiert kein universelles Renderingprimitiv – stattdessen werden Schnittstellen für funktionale Aspekte bereitgestellt, die von einer in einem konkreten System zu definierenden Basisklasse aller Renderingprimitive erfüllt werden müssen. Generic-3D definiert z.B. eine Radiosity-, eine Ray-Tracing-, eine Shading- und eine Surface-Stil-Schnittstelle. Die Auswahl der Schnittstellen für die Basisklasse erfolgt abhängig von den Renderingverfahren, die im konkreten System unterstützt werden sollen. Klassen geometrischer Objekte werden durchgängig unmittelbar von der Primitiv-Basisklasse abgeleitet [10]. Darüber hinaus werden Lichtquellen, Texturen und graphische Attribute in entsprechenden Klassenhierarchien repräsentiert. Anzumerken ist, dass bei der Frage nach dem Objektmodell für Generic-3D keine befriedigende Antwort gegeben werden kann: Es handelt sich um Software-Architekturfragmente, die bei einem konkreten System auf denkbar beliebige Weise miteinander in Verbindung stehen können; das Verständnis für den „Systembaukasten“ Generic-3D ist schwer.

Generic-3D unterstützt verschiedene Arten der Szenenmodellierung, z.B. Szenengraphen, Szenenbäume und Szenenobjektlisten, und wird vom jeweiligen konkreten System bestimmt, dessen Basisklasse alle notwendigen Schnittstellen bündeln muss. In einer Szenengraphenbasierten Modellierung würde zum Beispiel diese Basisklasse die Schnittstelle einer Knoten-Klassen implementieren, so dass jedes Primitiv als Knoten eines Szenengraphen verwendet werden kann. Diese Herangehensweise zur Konstruktion eines konkreten Systems führt einerseits zu einer Klassenhierarchie, die bei den Basisklassen eine große Anzahl von Mehrfachvererbungen bedingt, andererseits zu Klassen, die wegen der Fülle der durch die Schnittstellenvereinbarungen notwendigen Methoden eine vielschichtige Funktionalität in sich vereinen.

Generic-3D definiert die Rendering-Pipeline implizit über die zum Einsatz kommenden Renderingverfahren und die Szenenrepräsentationsform. Über die Effizienz und Praktikabilität einer solchen Lösung gibt es bislang keine Aussagen, gleichwenig ist über die Traversierung von Szenenkomponenten dokumentiert. Es ist zu vermuten, dass die Traversierungsfunktionalität implizit die Rendering-Pipeline festlegt und vom Szenenrepräsentationstyp (z.B. Liste, Baum, Graph) abhängt.

Generic-3D ist als C++-Klassenbibliothek implementiert. Die Schnittstellen-Vererbung ist durch Mehrfachvererbung verwirklicht; Schnittstellen-Klassen definieren hauptsächlich reinvirtuelle Methoden, die keine (in C++) keine Implementierung besitzen. Typ-parametrisierte Klassen sind durch Templates dargestellt.

Generic-3D ist ein Baukasten für computergraphische Systeme. Das System definiert allgemeine Renderingschnittstellen und Default-Implementierungen, die es erlauben, Renderingprimitive hinsichtlich verschiedener Renderingverfahren mit lokalen und globalen Beleuchtungsmodellen funktional auszustatten. Durch Aggregation, Vererbung und Template-Instanziierung kann ein einzelnes Renderingsystem mit den bereitgestellten Grundkomponenten konstruiert werden. Als White-Box-System und Framework ist Generic-3D vollständig erweiterbar und in Anwendungen integrierbar.

2.2.4 Nichtphotorealistische Renderingsysteme

Nichtphotorealistisches Rendering (NPR) umfasst eine Vielzahl künstlerischer und gestalterischer computergraphischer Techniken. Im Mittelpunkt steht dabei nicht, wie im Fall des Photorealismus, die getreue, exakte Wiedergabe einer geometrischen Szene unter Berücksichtigung einer physikalisch-basierten Lichtberechnung, sondern das kommunikative Ziel des Bildes, z.B. Skizzieren, Andeuten, Umreißen. Meist werden geometrische Modelle abstrahiert dargestellt. Technisch werden dazu Renderingverfahren verwandt, die analytisch-objektpräzise oder im Bildraum pixelpräzise arbeiten. Linienzeichnungen repräsentieren z.B. eine wichtige Klasse von NPR-Verfahren.

Wenig lässt sich über die Software-Architektur von NPR-Systemen gegenwärtig aussagen, da sie meist in Form von Verfahren, nicht aber in Form konkreter Systemen, vorliegen, oder in bestehende Graphiksysteme integriert sind. Die Objektmodelle und die Szenenmodellierung im NPR sind grundsätzlich gleichartig zu denen anderer Renderingsysteme; wenig ist über spezielle Objektmodelle für NPR bekannt. Die Attributierung umfasst meist abstrakte Attribute wie 'Wichtigkeit' und 'Unsicherheit', die durch das konkret verwendete Renderingverfahren graphisch umgesetzt werden.

2.3 Evaluation der Software-Architekturen

Den vorgestellten Software-Architekturen ist eines gemeinsam: Sie entwerfen ein für ihre Aufgabe zugeschnittenes Objektmodell und stellen unterschiedlich flexible Modellierungstechniken für Szenen bereit.

Dieser Abschnitt vergleicht und bewertet die vorgestellten Software-Architekturen im Hinblick auf zwei wesentliche Kriterien, das der Verwendbarkeit und das der Erweiterbarkeit. Unter *Verwendbarkeit* soll die Architektur-Transparenz verstanden werden, die entscheidet, wie effektiv mit einem computergraphischen System entwickelt werden kann – Verständlichkeit und Kompaktheit der Software-Architektur sind hier zwei entscheidende Faktoren. Unter *Erweiterbarkeit* soll verstanden werden, wie leicht anwendungsspezifische Anforderungen hinsichtlich der Objekttypen, der Renderingverfahren und der Modellierungstechniken in die jeweilige Software-Architektur integriert werden können.

2.3.1 Verwendbarkeit von Renderingsystemen

Alle vorgestellten Software-Architekturen verfügen grundsätzlich über eine für ihre Zwecke optimale Schnittstelle und ein optimales Objektmodell, da Schnittstellen und Objektmodelle zusammen mit den Systemen entstanden sind. Einige Überlegungen zur Verwendbarkeit müssen dennoch angestellt werden.

Verwendbarkeit von OpenGL

OpenGL fokussiert auf ein primitives, aber effizient auf Hardware abbildbares Objektmodell; da die Funktionalität aller OpenGL-Konstrukte dokumentiert und im OpenGL-Standard festgeschrieben ist, ist insbesondere die Laufzeit-Komplexität einer Renderingaufgabe für den Entwickler abschätzbar. Anwendungen haben dadurch die Möglichkeit, gezielt anwendungsspezifische Algorithmen und Datenstrukturen zu entwerfen, die den Graphik-Assembler optimal nutzen und dabei sichergehen können, dass die Implementierung auf allen Plattformen ein ähnliche Laufzeit-Komplexität besitzt. Die Sammlung der Renderingkonstrukte besteht aus ca. 250 Befehlen, die sich in wenige Befehlsgruppen einteilen lassen und zum größten Teil über eine konsistente Syntax verfügen. In der Praxis ist dieses System deshalb von Entwicklern für die Erstellung computergraphischer Systeme und Anwendungsprogramme grundsätzlich leicht handhabbar.

Eine Schwäche der Software-Architektur liegt allenfalls darin, dass Renderingverfahren, wie z.B. Verfahren zur Generierung von Schatten und Reflexion, in OpenGL eine zwar effiziente, aber auch zum Teil verwobene Implementierungsstruktur besitzen. Eine einfache Übernahme solcher Verfahren aus prototypischen Implementierungen in ein Anwendungsprogramm ist gewöhnlich durch einen Entwickler nur schwer möglich, da die Verfahren nicht selten eine vollständig unterschiedliche Traversierung des Szenengraphen erfordern und leicht mit vorhandenen
OpenGL-Programmabschnitten in Konflikt geraten; präzise Kenntnisse der OpenGL-Rendering-Pipeline und des OpenGL-Kontexts sind erforderlich, um mit OpenGL anspruchsvolle Renderingaufgaben zu lösen.

Verwendbarkeit von RenderMan

RenderMan stellt eine Fülle von Ausdrucksmöglichkeiten bereit und erreicht mit Hilfe seiner Shader-Sprache einen hohen Grad an gestalterischer Freiheit. Das Austauschformat, sprich die RenderMan-Szenenbeschreibungssprache, ist frei von Annahmen über das Renderingverfahren, so dass unterschiedliche Implementierungen mit unterschiedlichen Renderingverfahren eine RenderMan-Szene interpretieren können. Hinzu kommt die Fülle der geometrischen Primitive. Als Indiz für die hohe Nutzbarkeit und Flexibilität der RenderMan-Szenensprache kann die Tatsache herangezogen werden, dass die Sprache seit über 10 Jahren nahezu unverändert existiert, ohne dabei an Aktualität eingebüßt zu haben. Es existiert jedoch nur eine vollständige, nichtkommerzielle RenderMan-Implementierung, die Blue Moon Rendering Tools BMRT [39]. Die Komplexität der RenderMan-Spezifikation und das Fehlen einer offengelegten Referenzimplementierung mögen hierfür Gründe sein.

Andererseits besitzt RenderMan keine objektorientierte Software-Architektur. Eine Anwendung muss eine RenderMan-Szenenbeschreibungsdatei erstellen, kann also nicht einen objektbasierten Szenengraphen im eigenen Adressraum aufbauen und durch Einbettung eines RenderMan-Renderers direkt Bilder synthetisieren. RenderMan bleibt damit stets ein externes Renderingsystem.

Verwendbarkeit von POV-Ray

POV-Ray verfügt analog zu RenderMan über eine Vielzahl von Ausdrucksmöglichkeiten. Sein Objektmodell, das ebenfalls nicht explizit modelliert, sondern in der Szenenbeschreibungssprache indirekt enthalten ist, ermöglicht den Entwurf komplexer Szenen. Die Szenenbeschreibungssprache ist frei von Annahmen über das zugrundeliegende Renderingverfahren, wobei bislang nur die offizielle POV-Ray-Implementierung zur Verfügung steht. Aufgrund der historisch starken Ausrichtung auf das Ray-Tracing sind andere Renderingverfahren im Kontext POV-Rays nicht in Entwicklung.

Schwächen der POV-Ray-Software-Architektur liegen im Fehlen eines APIs, so dass ein objektbasierter Szenengraph nicht von einer Anwendung aufgebaut und editiert werden kann. In der Szenenbeschreibungssprache kommt andererseits das Objektmodell, auch wegen Kompatibilitätszugeständnissen an frühere POV-Ray-Versionen, nicht klar zum Ausdruck. Das Objektmodell findet sich z.B. nicht in der offiziellen Dokumentation. Berücksichtigt man, dass gerade für die mentale Konstruktion einer Szene die Kenntnis des zugrunde liegenden Objektmodells von großer Hilfe ist, liegt hierin ein leicht zu behebender Nachteil in der Nutzbarkeit.

Verwendbarkeit von BOOGA und Generic-3D

Die Software-Architekturen von BOOGA und Generic-3D haben hinsichtlich der Verwendbarkeit ein gemeinsames Merkmal: Ihre Verwendbarkeit ist abhängig vom Verstehen ihrer Software-Architektur und Komponenten. Sie stellen dem Anwendungsentwickler eine komplexe, objektorientierte Systemarchitektur zur Verfügung, deren Erlernen im Allgemeinen nur bedingt ökonomisch und nur bei langfristig geplantem Einsatz möglich ist. In BOOGA finden sich ca. 700 Klassen mit ca. 6500 Methoden – verglichen mit OpenGL oder POV-Ray also mehr als zehnmal so viel. Die Leistungsmerkmale sind andererseits nicht wesentlich größer: Die objektorientierte und komponentenorientierte Software-Architektur stellt für sich genommen eine elegante *Lösung für die Implementierung* eines Renderingsystems dar, ist aber für die Anwendungsentwicklung in ihren Schnittstellen denkbar schwierig zu erlernen und handzuhaben. Die Systemarchitekturen sind hinsichtlich der objektorientierten Implementierung optimal, nicht aber hinsichtlich des Objektmodells aus Nutzersicht. Die Systeme mögen als Beleg dafür dienen, dass die teilweise hohen Erwartungen an die Objektorientierung für computergraphische Systeme in dieser Form nicht erfüllbar sind.

BOOGA versucht durch Definition einer Komponentenschicht Abhilfe zu schaffen: Komponenten sind per Definition high-level Konstrukte, die mit einer klar festgelegten, explizit dokumentierten Schnittstelle ausgestattet sind. Allerdings profitiert der Entwickler davon nur bedingt: Es ist kaum möglich, die Bandbreite der Anforderungen auf universelle Komponenten abzubilden; der Bau eigener Komponenten setzt aber detaillierte Kenntnisse des Frameworks und der Graphikbibliothek voraus – eine unmittelbare Implementierung einer Funktionalität mit OpenGL ist im Vergleich meist praktikabler und ökonomischer.

Noch einen Schritt weiter geht Generic-3D: Es dient der Konstruktion eigener, anwendungsspezifischer computergraphischer Systeme. Die Bausteine und Mechanismen jedoch sind in ihren Folgen und gegenseitigen Einflüssen kaum durch einen Anwendungsentwickler abzuschätzen. Durch die intensive Nutzung von Aggregation, Vererbung und Templates wird zwar eine typsichere, redundanzfreie Implementierung konkreter Systeme gewährleistet, jedoch stellt die Entwicklung eines computergraphischen Systems sicherlich nicht den Hauptanwendungsfall eines computergraphischen Systems dar.

> Bei der Betrachtung existierender Renderingsysteme liegt die Verwendbarkeit von Black-Box-Systemen deutlich höher als die von White-Box-Systemen: Die Schnittstellen in Black-Box-Systemen sind explizit für die Nutzung durch Anwendungsentwickler entworfen, sind kompakt und nur dort erweiterbar, wo dies ausdrücklich gewollt ist. Insbesondere objektorientierte, computergraphische Systeme erschweren aufgrund ihres implementierungszentrierten Entwurfs eine effektive Nutzung und besitzen durch die Feinkörnigkeit ihrer Klassenmodelle eine zusätzliche Hürde für die unmittelbare Verwendung des Systems durch einen Entwickler – ein Paradoxon, wenn wir uns die Ziele, die Objektorientierung verfolgt, vergegenwärtigen.

2.3.2 Erweiterbarkeit von Renderingsystemen

Die Erweiterbarkeit eines Renderingsystems ist bei seiner langfristigen Nutzung eine entscheidende Qualität. Wofür im einzelnen Erweiterbarkeit bereitgestellt werden soll, wird im folgenden Abschnitt diskutiert.

In Darstellungen objektorientierter Software-Architekturen findet sich häufig die Behauptung, sie seien erweiterbar als direkte Folge ihrer objektorientierten Konstruktion. Dies ist nur insofern richtig, als dies technisch zutrifft. Erweiterbarkeit wird wesentlich im Entwurf verankert: Sind in der Software-Architektur nicht explizit die Vorkehrungen getroffen, findet Erweiterung in der Praxis nicht statt – weil sie nicht effektiv und transparent durchgeführt werden kann. Eine Software-Architektur muss festlegen, an welchen Stellen mit welchen Mitteln wozu erweitert werden kann – und dies sind fast immer wenige, ausgewählte Elemente einer Software-Architektur. Vor allem ist es notwendig, sich beim Architekturentwurf daran zu erinnern, dass ein System nicht für andere Renderingsystem-Entwickler, sondern für Anwendungsentwickler gebaut wird.

Erweiterbarkeit soll im folgenden unter den Aspekten der Integration neuer Renderingprimitive, neuer Renderingverfahren und neuer Szenenrepräsentationsformen untersucht werden; sie spiegeln die Hauptanwendungsfälle von Erweiterung wider.

2.3.2.1 Integration neuer Renderingprimitive

Die Integration neuer Renderingprimitive ist dann notwendig, wenn die vorhandenen Renderingprimitive in ihrer Funktionalität nicht ausreichend sind oder mit ihrer Hilfe nur unzureichend oder ineffizient anwendungsspezifische Primitive dargestellt werden können. Ein Beispiel dafür ist eine zusammenhängende Menge von Tetraedern, deren äußere Hülle gezeichnet werden soll. Eine Repräsentation durch Dreiecke wäre denkbar, ist jedoch im Fall von Intersektionsabfragen und beim Frame-Rendering nicht effizient. Ein neues Renderingprimitiv „Tetraedermenge“ könnte hier Eigenschaften der Datenstruktur nutzen, um optimale Implementierungen bereitzustellen.

Entwicklung neuer Renderingprimitive mit Black-Box-Systemen

Bei Black-Box-Systemen, z.B. OpenGL und HOOPS, und teilweise auch bei Gray-Box-Systemen (z.B. RenderMan) ist eine Erweiterung um neue Renderingprimitive nicht möglich. Sie können nur durch eine zusätzliche Systemschicht integriert werden, die auf der Konvertierung der neuen Primitive in elementarere Primitive beruht.

Das API von OpenGL ist nicht erweiterbar: Die Schnittstelle als auch die Rendering-Pipeline ist komplett verschlossen. OpenGL wird über zusätzliche Systemschichten erweitert; ein klassischer Vertreter ist zum Beispiel die OpenGL Utility Library, die eine Reihe von Solid-Körpern aus OpenGL-Primitiven modelliert. Eine andere Erweiterungsmöglichkeit, etwa durch Programmierung des OpenGL-Kernels, ist derzeit technisch nicht machbar.

RenderMan sieht keine Erweiterbarkeit durch neue Renderingprimitive vor, stellt jedoch eine Shader-Sprache zur Verfügung, mit der bedingt neue Renderingprimitive programmiert werden können. Dadurch, dass RenderMan kein spezielles Renderingverfahren favorisiert, wäre es kaum möglich, allgemein eine Erweiterung vorzunehmen, ohne dabei bestimmte Renderingverfahren zu unterstellen.

POV-Ray kann als Black-Box-Renderingsystem nicht erweitert werden. Es sei angemerkt, dass die POV-Ray-Implementierung Entwicklern frei zur Verfügung steht, jedoch nicht als White-Box-System intendiert ist. Es wäre grundsätzlich möglich, in POV-Ray mit vertretbarem Aufwand neue Renderingprimitive zu integrieren, obwohl POV-Ray keine objektorientierte

Programmiersprache verwendet: POV-Ray besitzt aber einen transparenten, objektbasierten Aufbau. Eine objektorientierte Implementierung würde allerdings die technische Abwicklung erheblich vereinfachen und zugleich einen „geordneten“ Weg zur Erweiterung darstellen.

Entwicklung neuer Renderingprimitive mit White-Box-Systemen

Objektorientiert implementierte Graphiksysteme (z.B. Generic-3D, BOOGA und Vision) ermöglichen über den Vererbungsmechanismus und die Instanziierung von Templates die Integration neuer Renderingprimitive. Die jeweiligen Basisklassen legen fest, welche Methoden (allgemein: welche Schnittstellen) von abgeleiteten Klassen implementiert werden können bzw. müssen.

In BOOGA wird angenommen, dass ein neuer Primitivtyp fest vorgegebene Schnittstellen implementiert. Bei Generic-3D hängt es von der Konfiguration des abgeleiteten Renderingsystems ab. Die Definition von Schnittstellen verspricht ein sicheres Handling bei der Erweiterung, denn die Implementierung von Schnittstellen kann maschinell vom Compiler überprüft werden. Hierbei gilt es jedoch zu bedenken, dass die Erweiterbarkeit per Definition auf solche Erweiterungen eingeschränkt ist, die durch die Methodenausstattung der Schnittstellen überhaupt möglich sind.

Gerade in diesem Punkt zeigt sich die eigentliche Schwäche einer stark auf Schnittstellen-Vererbung basierenden Erweiterbarkeit: Nicht immer sind alle Schnittstellen für eine neue Klasse von Renderingprimitiven sinnvoll implementierbar. Soll beispielsweise für ein nichtphotorealistisches Renderingverfahren ein neues Renderingprimitiv „Pinselstrich“ eingeführt werden, das im Bildraum eine Linie mit einer vorgegebenen Pinselform zeichnet, so sind Schattierungs-, Ray-Tracing- und Radiosity-Schnittstellen nicht sinnvoll implementierbar, da es sich um ein Bildraumprimitiv handelt.

> Die Erweiterbarkeit um neue Renderingprimitive ist für den langfristigen, effizienten Einsatz eines computergraphischen Systems wesentlich, da nur so anwendungsspezifische Daten optimal für das Rendering umgesetzt werden können. Black-Box-Systeme können dem nur bedingt Rechnung tragen: Zusätzliche Systemschichten können neue Renderingprimitive auf der Basis einer Zerlegung in elementare Renderingprimitive ermöglichen. White-Box-Systeme ermöglichen die Erweiterung durch Spezialisierung von Renderingprimitiv-Basisklassen.

2.3.2.2 Integration neuer Renderingverfahren

Die Entwicklungen im Bereich des photorealistischen und des nichtphotorealistischen Renderings verdeutlichen, dass ein computergraphisches System auch in Zukunft nicht an einem einzigen Renderingverfahren ausgerichtet sein darf. Ein computergraphisches System muss Vorkehrungen treffen, sein Objektmodell für verschiedene Renderingverfahren auswerten zu können.

Renderingverfahren als architekturbestimmendes Merkmal

Die Software-Architekturen heutiger computergraphischer Systeme sind bestimmt von der Implementierung des genutzten Renderingverfahrens. Deswegen stellen z.B. nichtphotorealistische Renderingsysteme vollkommen neuartige Ansprüche an die Software-Architektur, da sie bezüglich der Illumination und des Zeichnens grundsätzlich anders arbeiten. Aber auch Echtzeit-Renderingverfahren, die Schatten, Reflexion und Spiegelung durch Multi-Pass-Rendering implementieren, sind nicht mit den traditionellen Software-Architekturen realisierbar.

OpenGL bietet in allen Stufen seiner Echtzeit-Rendering-Pipeline Freiheitsgrade, die zur Implementierung neuer Renderingverfahren genutzt werden können. Allerdings wird die Implementierung aus der Sicht des Software-Engineerings nicht durch einen Framework unterstützt. Immerhin: die sehr gut faktorisierte Sammlung von computergraphischen Primitiven und Primitivoperationen verhilft OpenGL zu einer erstaunlichen Flexibilität und Leistungsfähigkeit.

RenderMan und POV-Ray können als Black-Box-Systeme nicht mit neuen Renderingverfahren ausgestattet werden. RenderMan unterstützt bedingt im Rahmen der Shader-Sprache neue Renderingverfahren, z.B. durch Implementierung von nichtphotorealistischen Renderingstilen in Form von Surface-Shadern [3]. In BOOGA und Generic-3D sind die Schnittstellen auf Radiosity- und Ray-Tracing-Verfahren ausgelegt und optimiert. Es ist anzunehmen, dass die Integration anderer Verfahren mit erheblichem Redesign verbunden wäre und nicht durch die vorhandene Software-Architektur unmittelbar profitiert.

Nichtstandard-Renderingverfahren

Neben dem Fallbeispielen des photorealistischen, nichtphotorealistischen und Echtzeit-Renderings gibt es weitere Renderingverfahren, die zunehmend an Bedeutung gewinnen. Der Begriff *Nichtstandard-Rendering* versucht, den Verfahren, die bezogen auf das übliche Verständnis des Begriffs Rendering kein Rendering darstellen (z.B. Sound-Rendering), aber letztlich auch einen graphisch-geometrischen Sachverhalt in ein Zielmedium übertragen, einen Namen zu geben. Der Begriff basiert auf der Definition von *Standard* als "something set up by general consent as a rule; principle established by authority or custom" (New Webster’s Encyclopedic Dictionary). Beispiele für Nichtstandard-Renderingverfahren sind:

- *MPEG-Rendering*: Unter dem Begriff „MPEG-Rendering“ kann sowohl die Übertragung einer (audio-visuellen) Szene in ein Multimedia-Format, z.B. MPEG-4 [68] oder MPEG-7 [69], als auch die Rückübertragung von MPEG-kodierten Datenströmen in eine abstrakte, objektbasierte Beschreibung verstanden werden. MPEG-4 unterstützt eine Reihe von sog. Medien-Objekten (z.B. statische Bilder, Video-Objekte, Audio-Objekte) und erklärt eine Szenenstruktur, die es ermöglicht, diese Objekte zu platzieren, zu transformieren und zu gruppieren. MPEG-7 ist als Standard für ein „Multimedia Content Description Interface“ geplant. Es soll insbesondere eine „Description Definition Language“ enthalten, die die Entwicklung eigener Multimedia-Objekte innerhalb eines MPEG-7-Dokuments ermöglicht. Im Zusammenhang mit MPEG stellt sich die Frage, wie ein generisch gehaltenes Renderingsystem die Interpretation und Umsetzung solcher multimedialen Objekte methodisch einheitlich angehen kann. Die Übertragung von und die Rückübertragung in MPEG-Dokumente können somit als spezielle Formen eines verallgemeinerten Renderings verstanden werden.
- *Sound-Rendering*: Darunter wird die Übertragung von nichtgeometrischen Primitiven (z.B. Sounds) unter Berücksichtigung geometrischer Primitive mit Audio-Attributen (z.B. Sound-Emitter, Sound-Reflector, Sound-Blocker) in einen Audio-Strom verstanden [11]. Das Sound-Rendering in 3D-Umgebungen steht in enger Wechselbeziehung zur Szenengeometrie (z.B. sich bewegende Sound-Quelle und ihre Verdeckung durch Szenenobjekte) [103]. Im Zusammenhang mit Multimediasystemen erhält das Sound-Rendering und seine methodisch einheitliche Integration in den Gesamtrenderingvorgang eines Multimedia-Dokuments eine besondere Bedeutung.

Insgesamt muss somit gefordert werden, dass die Software-Architektur eines generischen Renderingsystems sich nicht an einem konkreten Renderingverfahren orientieren darf. Eine Trennung von den auszuwertenden Objekten und dem Prozess ihrer Auswertung ist notwendig.

Die Erweiterbarkeit eines computergraphischen Systems durch neue Renderingverfahren ist für die langfristige Nutzung des Systems entscheidend, da nur so der Dynamik in der Entwicklung von Renderingverfahren Rechnung getragen wird. Das Renderingverfahren und sein Implementierungsmodell darf dabei kein Kriterium für die Gesamtarchitektur sein. Insbesondere muss für zukünftige Entwicklungen, z.B. dem nichtphotorealistischen Rendering und den verschiedenen Ansätzen im Nichtstandard-Rendering, Vorsorge getroffen werden.

2.3.2.3 Integration neuer Szenenrepräsentationstypen

Die Integration neuer Typen von Szenenrepräsentationen bzw. die Modifikation vorhandener Typen kann notwendig werden, wenn die Repräsentation anwendungsspezifisch optimiert werden soll oder wenn ein neues Renderingverfahren einen anderen Repräsentationstyp erzwingt. Zum Beispiel kann eine Anwendung im Bereich der technisch-wissenschaftlichen Visualisierung Felder von 3D-Glyphen dadurch repräsentieren, dass ein neuer Typ eines Szenengraphenknoten bereitgestellt wird, der intern die verschiedenen Glyphen-Geometrien vorhält und diese an den Feldpositionen direkt zeichnet. Als weiteres Beispiel kann ein Renderingverfahren betrachtet werden, das verdeckte Linien analytisch berechnet und diese mit PostScript ausgibt, aber dazu globale Szeneninformation benötigt.

Szenenrepräsentationstypen in Black-Box-Systemen

OpenGL unterstützt keinen speziellen Szenenrepräsentationstyp, kann jedoch aufgrund seines stackbasierten Kontexts, der alle Renderingparameter und Renderingmodalitäten repräsentiert, effizient zur Implementierung unterschiedlicher Szenenrepräsentationstypen herangezogen werden. Eine Reihe von Szenenrepräsentationstypen haben sich so unabhängig entwickelt: OpenInventor z.B. definiert den klassischen Szenengraphen; OpenGL Optimizer stellt einen Szenengraph bereit, der insbesondere in der Lage ist, große polygonale Modelle, wie sie im CAD/CAM-Bereich verarbeitet werden, durch Simplifikations- und Cullingtechniken interaktiv zu visualisieren.

HOOPS legt im Gegensatz zu OpenGL den Szenenrepräsentationstyp fest. Segmentbäume in HOOPS können nicht erweitert werden; alternative Szenenrepräsentationstypen sind auch aufgrund der Konstruktion als Retained-Mode-System nicht realisierbar.

Die RenderMan-Szenenrepräsentation bietet eine allgemein einsetzbare Szenenbeschreibungsform, die sich insbesondere als Austauschformat einsetzen lässt. Die Szenenbeschreibungssprache ist nicht erweiterbar; ausschließlich neue Shader können berücksichtigt werden, nicht jedoch andere, neue Renderingkomponenten.

POV-Ray stellt ebenfalls eine Szenenbeschreibungssprache bereit, die jedoch keine hierarchische Anordnung von Renderingkomponenten (mit der Ausnahme von CSG-Objekten) ermöglicht. Intern sind geometrische Objekte und Lichtquellen in Listenform gespeichert. Eine Erweiterung durch einen Szenengraph ist in der jetzigen Form nicht praktikabel und nur über ein Redesign der Implementierung erreichbar.

Szenenrepräsentation in White-Box-Systemen

BOOGA stellt einen Szenengraphen bereit, der auf der Basis des Visitor-Entwurfsmusters arbeitet und somit eine flexible Erweiterung der Auswertungsstrategien ermöglicht. So wurden z.B. unterschiedliche Renderingverfahren, die auf der gleichen Szenenrepräsentation operieren können, in Form von Renderer-Komponenten entworfen und nahtlos integriert. Die Szenen-

repräsentation lässt sich jedoch nicht hinsichtlich neuer Renderingkomponenten, z.B. neuer graphischer Attribute, erweitern.

Generic-3D ermöglicht die Nutzung verschiedener Szenenrepräsentationstypen. Jedes von Generic-3D abgeleitete Renderingsystem muss den Repräsentationstyp durch Vererbung in die Basisklasse seiner Renderingkomponenten integrieren. Von Generic-3D werden sowohl hierarchische (z.B. DAG) als auch sequentielle (z.B. Liste) Szenenrepräsentationstypen bereitgestellt. Insofern erweist sich Generic-3D als sehr flexibel, jedoch ist durch die Kopplung von Renderingkomponenten und Szenenrepräsentation eine hohe Abhängigkeit von Inhalt und Struktur gegeben. Eine Anwendung kann insbesondere nicht unterschiedliche Szenenrepräsentationstypen gemeinsam einsetzen.

Szenenrepräsentation und Renderingverfahren

Unabhängig vom konkreten Typ der Szenenrepräsentation muss gewährleistet sein, dass die Traversierung spezifisch zum jeweiligen Renderingverfahren definiert werden kann. Die Traversierungsmechanismen der vorgestellten Systeme unterstellen, dass die Traversierung in einem zwar selektiven und steuerbaren, doch einmaligen Durchwandern der Renderingkomponenten besteht. Dies ist ausreichend, wenn alle Szenenobjekte durch eine 1:1-Übertragung in die Zielstrukturen überführt werden können. Im Fall von klassischem OpenGL-Rendering trifft dies zu: Geometrische Objekte werden in OpenGL-Kommandofolgen umgesetzt und graphische Attribute sowie geometrische Transformationen modifizieren den OpenGL-Kontext. Auch im Fall von POV-Ray ist dies eine ausreichende Strategie: Durch das Parsing der Szenenbeschreibung wird die sequentielle Sammlung von Renderingprimitiven inkrementell aufgebaut.

Problematisch wird es, wenn wir neuere Renderingverfahren im Bereich des Echtzeit-Renderings und des nichtphotorealistischen Renderings betrachten: Hier besteht die Notwendigkeit, den Szenengraphen oder bestimmte Szenensubgraphen mehrfach während des Renderings eines Bildes zu traversieren. Ein Renderingverfahren könnte zum Beispiel in einem ersten Durchgang die Szenenobjekte in den Stencil-Buffer, der für den zweiten Durchgang als Maske dient, zeichnen.

Als universeller Szenenrepräsentationstyp hat sich der Szenengraph etabliert. Er ermöglicht die kohärente, hierarchische Spezifikation von attributierten Szenen. Die Traversierung kann so optimiert werden, dass einzelne Objekte durch Pfadangabe direkt ausgewertet werden können. Auch lassen sich aus einem Szenengraph sekundäre Repräsentationen automatisiert generieren.

> Der Szenengraph, eine universeller Typ der Szenenrepräsentation, ermöglicht die hierarchische Beschreibung von graphisch-geometrischen Sachverhalten. Insbesondere werden darin Gemeinsamkeiten in der Attributierung von Szenenobjekten direkt ausgedrückt. Die Auswertung eines Szenengraphen kann auf der Grundlage des Visitor-Entwurfsmusters erfolgen, so dass verschiedene Auswertungsstrategien auf ein und denselben Szenengraphen angewendet werden können.

2.4 Qualitätskriterien für Software-Architekturen computergraphischer Systeme

Als Ergebnis der Analyse und der Evaluation der Software-Architektur computergraphischer Systeme können wir nun Kriterien für die Software-Architektur-Qualität aufstellen. Diese

Qualitätskriterien dienen einerseits zur Bewertung eines Systems, andererseits als Leitlinie für die Software-Architektur des im nächsten Kapitel vorgestellten generischen Renderingsystems.

2.4.1 Faktorisierung der Funktionalität

Eine Faktorisierung der Funktionalität in orthogonale Bestandteile und deren Gliederung in Subsysteme fördert die Flexibilität und Effizienz, mit der Aufgaben formuliert werden können. Faktorisierung zeigt sich konkret an folgenden Eigenschaften:

- **Trennung von Geometrie und Attributen.** Werden geometrische Objekte mit Attributen hierarchisch und dynamisch assoziiert, dann wird zum einen eine hohe Ausdruckskraft durch die Kombinationsmöglichkeiten erzielt, zum anderen Kohärenzinformation darstellbar. Der Szenengraph bietet hierfür eine effiziente Repräsentationsform.
- **Trennung von Renderingkomponenten und Szenenrepräsentation.** Renderingkomponenten, konkret geometrische Objekte und Attribute, müssen von der Szenenrepräsentation getrennt modelliert werden, um sowohl die Software-Architektur der Renderingkomponenten, als auch die der Szenenrepräsentation unabhängig entwickeln und weiterentwickeln zu können.
- **Trennung von Szenenrepräsentation und Renderingverfahren.** Die Szenenrepräsentation muss vom konkret verwendeten Renderingverfahren getrennt werden, um verschiedene Renderingverfahren auf der Basis der gleichen Szenenrepräsentation einsetzen zu können.

2.4.2 Strukturierung und Systematisierung der Komponenten

Die Strukturierung und Systematisierung aller an einem computergraphischen System beteiligten Komponenten dient der Transparenz des Systems und der mit ihm implementierten Anwendungen. Konkret bedeutet dies:

- **Strukturierung und Systematisierung geometrischer Objekte.** Die Sammlung geometrischer Objekte muss anhand ihrer deklarativen Bestandteile strukturiert werden, z.B. in polygonale, analytische und parametrische Objekte. Eine kanonische, geometrische Objekthierarchie kann unabhängig vom Renderingverfahren hergeleitet werden, wenn funktionale Aspekte (z.B. Schnittpunktberechnung oder Renderingalgorithmus) den Geometrie-Objekten je nach verwendetem Renderingverfahren beigefügt werden können, ohne dabei die Klasse des Geometrieobjekts selbst modifizieren zu müssen.
- **Strukturierung und Systematisierung der Attribute.** Attribute umfassen geometrische Transformationen, optische Attribute, logische Attribute, heuristische Attribute und renderingspezifische Attribute. Eine kanonische Attributhierarchie kann aufgrund der unterschiedlichen Fähigkeiten einzelner Renderingverfahren nur bedingt hergeleitet werden.
- **Strukturierung und Systematisierung der Szenenrepräsentationskomponenten.** Elemente hierarchischer Szenenrepräsentationen beinhalten Einzelkomponenten, Gruppenkomponenten, Auswahlkomponenten und Stellvertreterkomponenten.
- **Strukturierung und Systematisierung der Renderingverfahren.** Renderingverfahren, die Spezialisierungen, Vorstufen oder Nachstufen von anderen Renderingverfahren sind,

müssen strukturiert und systematisch modellierbar sein. Dies setzt voraus, dass ein Renderingverfahren als eigenständige Entität entworfen werden kann.

2.4.3 Abstraktion der Komponenten

Die Abstraktion der Komponenten ist eine wesentliche Voraussetzung für ihren effektiven Einsatz: Nur wenn die Komponenten komplexe Sachverhalte und Verfahren kapseln, entsteht ein Nutzen für die Anwendungsentwicklung. Konkret zeigt sich Abstraktion wie folgt:

- **Szenengraph als deklarative Beschreibung.** Der Szenengraph dient der hierarchischen Assoziation von geometrischen Objekten mit Attributen; er gibt nicht notwendig die Rendering-Pipeline noch die Auswertungsreihenfolge wieder. Die Transformation der Szenenbeschreibung in eine für das jeweilige Renderingverfahren optimale interne Repräsentation ist davon unabhängig.
- **Expliziter Entwurf einer Systemschnittstelle.** Die Systemschnittstelle ist zentrales Element eines Systems, denn sie entscheidet über die Nutzbarkeit. Sie muss explizit entworfen werden, denn sie ergibt sich nicht automatisch aus den Schnittstellen der Implementierungskomponenten. Die Systemschnittstelle kann Teilmenge dieser Schnittstellen sein oder auf der Basis des Fassaden-Entwurfsmusters [34] modelliert werden.

2.4.4 Integrationsmöglichkeiten für neue Komponenten

Die Offenheit einer Software-Architektur zeigt sich an den Möglichkeiten zur Erweiterung eines Systems. Wesentliche Kriterien sind hierfür:

- **Integration neuer Renderingkomponenten.** Anwendungsspezifische Renderingkomponenten müssen als vollwertige Bestandteile integrierbar sein, ohne dabei in ihrer Funktionalität im Vergleich zu den bereits eingebauten Renderingkomponenten eingeschränkt zu werden.
- **Integration neuer Renderingverfahren.** Computergraphische Systeme müssen in der Lage sein, neue Renderingverfahren zu integrieren, um sich ändernden (qualitativen als auch quantitativen) anwendungsspezifischen Anforderungen anpassen zu können, ohne dabei das System neu entwerfen und implementieren zu müssen.
- **Erweiterbarkeit der Szenenrepräsentation.** Die Konstrukte einer Szenenrepräsentation sind nicht nur Container für Renderingkomponenten, sondern können – da sie über Kontext- und Kohärenzinformation verfügen – Renderingkomponenten situationsabhängig modifizieren oder erzeugen. Die Erweiterbarkeit der Szenenrepräsentation wird erleichtert, wenn Renderingkomponenten von der hierarchischen Struktur separat modelliert werden; es ergibt sich eine klare Aufgabenteilung: Szenenrepräsentationen organisieren, erzeugen und modifizieren Renderingkomponenten; Renderingkomponenten sind von Renderingverfahren auszuwertende Entitäten.

Die nachfolgende Tabelle fast die wesentlichen Merkmale der Systemarchitektur der untersuchten computergraphischen Systeme zusammen. Auf dem Fundament der bisherigen Untersuchungen wird im Kapitel 3 ein Konzept für ein generisches Renderingsystem und im Kapitel 4 ein Konzept für einen generischen Szenengraphen vorgestellt.

Tabelle 1: Computergraphische Systeme und die Merkmale* ihrer Software-Architektur.

Rendering-system	**OpenGL**	**OpenInventor**	**HOOPS**	**RenderMan**	**Vision**	**POV-Ray**	**BOOGA**	**Generic-3D**
Systemarchi-tektur	*Bibliothek*	*Bibliothek*	*Bibliothek*	*Anwendung*	*Framework*	*Anwendung*	*Framework*	*Framework*
Systemaufbau	*Black-Box*	*Gray-Box*	*Black-Box*	*Gray-Box*	*Gray-Box / White-Box*	*Black-Box*	*White-Box*	*White-Box*
Systemschnitt-stelle	*C-API*	*C++-API*	*C-API*	*Szenenbeschrei-bungssprache*	*Szenenbeschrei-bungssprache, C++-API*	*Szenenbeschrei-bungssprache*	*C++-API*	*C++-API*
Systemer-weiterung	*Bibliotheken*	*Spezialisierte Klassen*	--	*Shader*	*Shader, speziali-sierte Klassen*	--	*Spezialisierte Klassen u. Kom-ponenten*	*Spezialisierte, in-stantiierte Klas-sen*
Rendering-verfahren	*Real-Time*	*Real-Time*	*Real-Time*	*Photorealismus*	*Photorealismus*	*Photorealismus*	*Real-Time, Photorealismus*	*Real-Time, Photorealismus*
Trennung von Geometrie und Attributen	++	++	++	++	++	--	- *(keine expl. Attr.-Hierarchie)*	- *(keine expl. Attr.-Hierarchie)*
Trennung von Rendering-komp. und Sze-nenrepr.	*keine Szenen-repräsentation definiert*	--	--	--	--	--	--	--
Trennung von Szenenrepr. und Renderingverf.	*keine Szenen-repräsentation definiert*	--	+ *(Auswertung durch abstrakte Geräteschnitt-stelle)*	+ *(Shader-Programmie-rung)*	+ *(Shader-Programmie-rung)*	--	+ *(Auswertung durch Renderer)*	--
Strukturierung geom. Objekte	*nur hardware-nahe Objekte*	- *(nur OpenGL Ob-jekte)*	++	++	++	- *(favorisiert typi-sche Ray-Tracer Objekte)*	- *(keine umfassen-de Sammlung)*	- *(direkte Bindung an Rendering-verfahren)*
Strukturierung der Attribute	*nur hardware-nahe Attribute*	- *(nur OpenGL Att-ribute)*	++	++	++	++	--	- *(direkte Bindung an Rendering-verfahren)*
Strukturierung der Szenenrepr.	*keine Szenen-repräsentation definiert*	+ *(enge Bindung an die Rendering-Pipeline von OpenGL)*	- *(nur baumartige Repräsentation)*	++	++	- *(nur sequentielle Repräsentation)*	++	++
Strukturierung der Rendering-verfahren	+ *(programmier-bare Rendering-Pipeline)*	--	+ *(Auswertung durch abstrakte Geräteschnitt-stelle)*	++	++	--	++	- *(Renderingver-fahren werden mit Hilfe von Schnittstellenver-erbungen impl.)*
Szenengraph als deklarative Be-schreibung	*keine Szenen-repräsentation definiert*	- *(enge Bindung an OpenGL)*	++	++	++	- *(deklarative seq. Repräsentation)*	++	--
Explizite Schnittstelle	++ *(orthogonale Funktionen)*	-- *(Implementie-rungsklassen = Schnittst.)*	++ *(vollständige Kapselung durch expl. API)*	++ *(Kapselung durch Szenenbeschrei-bungssprache)*	++ *(Kapselung durch Szenenbeschrei-bungssprache)*	++ *(Kapselung durch Szenenbeschrei-bungssprache)*	+ *(Komponente = Schnittst.)*	-- *(Implementie-rungsklassen = Schnittst.)*
Integration neu-er Rendering-komponenten	-- *(hardwarenahe Schnittstelle)*	+ *(durch speziali-sierte Knotenklas-sen)*	--	--	- *(durch speziali-sierte Klassen)*	--	++	++
Integration neu-er Rendering-verfahren	++ *(Werkzeug zur Impl. hardware-naher Verf.)*	--	--	+ *(teilweise mög-lich durch Shader)*	+ *(Restriktion auf physikalisch-basierte Verf.)*	--	++ *(durch Renderer-Komponenten)*	-- *(nur Auswahl aus vorhandenen Verf.)*
Erweiterung der Szenenrepräsen-tation	*keine Szenen-repräsentation definiert*	+ *(durch speziali-sierte Knotenklas-sen)*	--	--	--	--	++ *(durch speziali-sierte Knoten-klassen)*	- *(nur Auswahl aus vorhandenen Grundtypen)*

* Die Merkmale sind mit *sehr gut unterstützt* (++), *bedingt unterstützt* (+), *schlecht unterstützt* (-) oder mit *nicht unterstützt* (--) bewertet.

3 ARCHITEKTUR EINES GENERISCHEN RENDERINGSYSTEMS

In diesem Kapitel wird ein Entwurf für die Software-Architektur eines generischen Renderingsystems vorgestellt. Die Idee ist es, einen Framework zu schaffen, der einerseits bestehende Renderingsysteme vollständig integriert und andererseits neue Renderingfunktionalität effizient und effektiv zu implementieren erlaubt. Die Software-Architektur sowie ausgewählte Schlüsselmechanismen eines solchen *generischen Renderingsystems* werden nachfolgend erläutert, wobei insbesondere das objektorientierte Systemmodell diskutiert wird.

3.1 Konzepte eines generischen Renderingsystems

Renderingsysteme sind das Ergebnis intensiver Planung, langwieriger Implementierung und ausgefeilter Optimierung: Sie erfordern komplexe Algorithmen und Datenstrukturen, ein tiefes Verständnis der unterliegenden physikalischen Beleuchtungsmodelle, und sie greifen bisweilen direkt auf die Graphik-Hardware zu. Von daher wäre die Implementierung eines neuen, objektorientierten Renderingsystems, das alle gängigen Renderingverfahren in vergleichbarer Qualität wie die bestehenden Renderingsysteme zur Verfügung stellen wollte, weder ökonomisch noch praktisch durchführbar.

Erfahrungswerte bieten universitäre und freie Softwareprojekte, die die Implementierung eines Renderingsystems zum Ziel hatten, wie zum Beispiel das Blue-Moon-Rendering-Toolkit-Projekt (seit 1990) von Gritz [39], das Vision-Projekt (1992-1999) von Slussalek [92] oder das Mesa-Projekt (seit 1993) von Paul [77], das eine vollständige Implementierung des OpenGL-Standards zum Ziel hat.

Angesichts der Komplexität der Implementierung von Renderingsystemen wollen wir eine Software-Architektur für ein Renderingsystem entwerfen, das die Hauptziele hat,

- die Funktionalität bestehender Renderingsysteme objektorientiert zu ummanteln,
- ein konsistentes und erweiterungsfähiges Klassenmodell zu unterlegen,
- die Schnittstellen zu den einzelnen Renderingsystemen zu vereinheitlichen,
- aber zugleich ihre individuellen Renderingfähigkeiten darin zu erhalten.

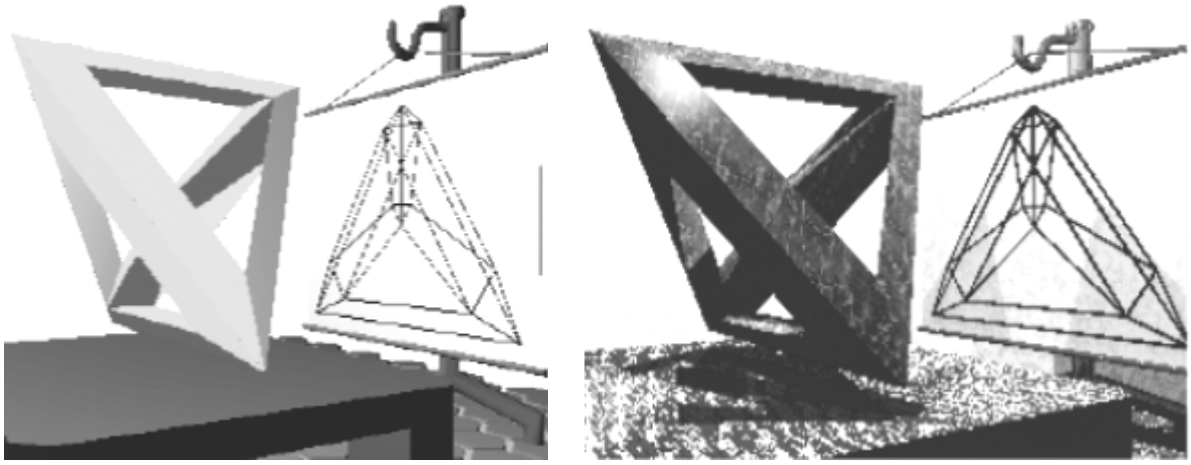

Abbildung 12. Zwei Bilder eines Lehrfilms über Geometrie. Das Echtzeit-Rendering zur Planung der Animationssequenz erfolgt mit OpenGL (links), das finale Rendering geschieht mit POV-Ray. Die Anwendung beruht auf dem Virtuellen Renderingsystem.

Ein Renderingsystem mit einer solchen Software-Architektur kann einerseits ein *generisches Renderingsystem* genannt werden, denn es repräsentiert die Gemeinsamkeiten einer Gruppe von Renderingsystemen, andererseits kann es als *virtuelles Renderingsystem* begriffen werden, denn es dient als Framework zur Ansteuerung existierender Renderingsysteme, stellt selbst aber nicht notwendig eine eigene Implementierung der jeweiligen Renderingverfahren bereit. Wird im folgenden von dem *Virtuellen Renderingsystem* gesprochen, so ist das konkret vom Autor entwickelte generische Renderingsystem gemeint.

Der Nutzen eines generischen Renderingsystems ist vielfältig: Die Ansteuerung verschiedener Systeme erzwingt es, dass alle am Rendering beteiligten Komponenten, insbesondere Renderingprimitive und Renderingalgorithmen, einem gemeinsamen Grundmuster folgen, das auf einer einheitlichen Methodik und Terminologie beruht. Dazu muss ein generisches Renderingsystem eine Sammlung von Renderingkomponenten definieren. Da es eine einheitliche Schnittstelle zu unterschiedlichen Renderingsystemen und Renderingverfahren repräsentiert, erhöht es einerseits den Abstraktionsgrad in der Anwendungsentwicklung, andererseits wird die Nutzung konkreter Renderingsysteme erleichtert. Gleichwohl ist es wichtig, dass individuelle Fähigkeiten und damit Stärken einzelner Renderingsysteme in dieser Software-Architektur vollständig ausgedrückt werden können. Andernfalls wäre ein generisches Renderingsystem nicht in professionellen Anwendungen einsetzbar. Weiterhin kann eine Anwendung auf der Basis eines generischen Renderingsystems das für die Bildsynthese genutzte Renderingsystem wechseln, ohne dabei den Quelltext der Anwendung ändern zu müssen. Es lassen sich damit zum Beispiel Echtzeit-Rendering, photorealistisches Rendering und nichtphotorealistisches Rendering softwaretechnisch in einem Framework integrieren.

Die in diesem Kapitel vorgestellte Software-Architektur wurde prototypisch in Form des *Virtuellen Renderingsystems (*VRS*)* implementiert [27]. VRS ist als objektorientiertes White-Box-System konzipiert und „ummantelt“ derzeit OpenGL, das Lichtsimulationssystem Radiance, POV-Ray und RenderMan. Speziell für OpenGL wurde die Implementierung optimiert, so dass VRS keine wesentlichen Geschwindigkeitseinbußen mit sich bringt. Darüber hinaus wurden einige der sog. Advanced Graphics Programming Techniques [62] objektorientiert in VRS integriert, z.B. Echtzeitverfahren für Schatten, Reflexion und Bump-Mapping.

Abbildung 12 zeigt zwei Bilder eines Lehrfilmes, der mit VRS modelliert und gerendert wurde. Der Lehrfilm visualisiert und animiert ein komplexes Polyeder. Das Polyeder besitzt versteckte Symmetrien, die dadurch sichtbar werden, dass ihre Kanten auf eine ebene Fläche

(der Leinwand) projiziert werden. OpenGL wird durch VRS zum Echtzeit-Rendering eingesetzt und POV-Ray zum Rendering der photorealistischen Bildersequenz.

3.1.1 Anforderungen an ein generisches Renderingsystem

Ein generisches Renderingsystem inkorporiert *konkrete Renderingsysteme* in seine Implementierung. Es implementiert von daher nicht nativ ein eigenes Renderingverfahren, sondern definiert einen Framework, in dem solche Verfahren (in Form existierender Renderingsysteme) integriert und durch eine einheitliche Schnittstelle angesprochen werden können. Die Merkmale der Software-Architektur eines generischen Renderingsystems lassen sich wie folgt zusammenfassen:

- *Objektbasierte Renderingkomponenten*. Die Renderingkomponenten werden objektorientiert modelliert. Insbesondere werden damit explizit die Schnittstellen zu Renderingobjekten festgelegt. Geometrische Objekte und graphische Attribute sind Vertreter wichtiger Objektkategorien. Auch Renderingverfahren sollten objektbasiert modelliert werden, um zu gewährleisten, dass Renderingverfahren als eigenständige Objekte identifizierbar bleiben und nicht in der Implementierung eines Geometrieobjektes oder eines graphischen Attributes vermengt werden. Eine objektorientierte Schnittstelle stellt zugleich die Voraussetzung dar, um an Objektverteilungsmechanismen (z.B. CORBA) und Objekteinbettungsverfahren (z.B. ActiveX) teilhaben zu können [102].
- *Deklarative Renderingkomponenten*. Die Renderingkomponenten sollten einen deklarativen Charakter besitzen: Ihre Nutzung und ihre Verwendung sollte ohne Kenntnisse und Annahmen der Implementierung möglich sein. Zugleich muss gewährleistet werden, dass sie soweit wie möglich gleichermaßen für unterschiedliche Renderingsysteme einsetzbar sind. Insgesamt soll dadurch die Nutzbarkeit der Komponenten erhöht werden.
- *Erweiterbare Renderingkomponenten*. Die Software-Architektur muss durch neue Renderingkomponenten erweitert werden können. Anwendungsspezifische Renderingkomponenten, z.B. spezielle geometrische Objekte, Attribute oder Renderingverfahren, müssen gleichberechtigt mit den eingebauten Renderingkomponenten integrierbar sein und sollten vom Framework so viel Unterstützung wie möglich bei ihrer Implementierung erhalten.
- *Effizientes Rendering*. Konkrete Renderingsysteme, die in ein generisches Renderingsystem integriert werden, müssen effizient verwendet werden können. Effizient bedeutet hier, dass kein wesentlicher Geschwindigkeitsverlust eintreten darf. Insbesondere für Echtzeit-Renderingsysteme ist es notwendig, Renderingkomponenten direkt auf das zugrundeliegende Renderingsystem abzubilden; die Konvertierung muss explizit kontrollierbar und substituierbar sein, um zeitkritische Anwendungen oder bei Verarbeitung großer Datenmengen gezielt optimieren zu können.

Die genannten Anforderungen sind Ausgangspunkt für den Entwurf und die Implementierung des Virtuellen Renderingsystems VRS.

3.1.2 Systemarchitektur des Virtuellen Renderingsystems

Die Systemarchitektur des Virtuellen Renderingsystems ist in Abbildung 13 dargestellt. Der Kern von VRS besteht aus einer Sammlung von allgemein einsetzbaren Renderingkomponenten, die geometrische Objekte (*Shapes*), graphische und nichtgraphische Modifikatoren (*Attribute*), Auswertungsalgorithmen (*Handler*), Auswertungsstrategien (*Techniques*) und Auswertungsmaschinen (*Engines*) umfassen. Zugleich sind für jedes konkrete Renderingsystem spe-

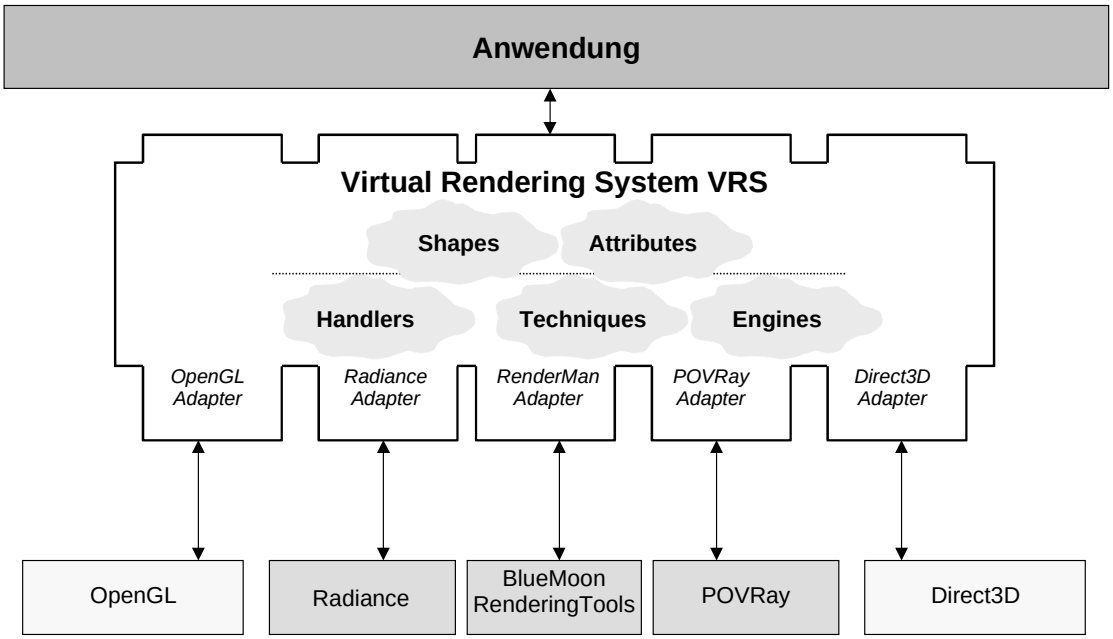

Abbildung 13. Systemarchitektur und Renderingkomponenten des Virtuellen Renderingsystems.

zielle Renderingkomponenten vorhanden, die für den Zugriff auf die jeweiligen spezifischen Fähigkeiten verantwortlich sind.

Die Integration eines konkreten Renderingsystems erfolgt durch die Implementierung eines Adapter-Subsystems. Ein *Adapter-Subsystem* umfasst alle Handler-, Technik- und Engine-Klassen, die zur Umsetzung von Shapes und Attributen für das konkrete Renderingsystem benötigt werden.

Echtzeit-Renderingsysteme, wie z.B. OpenGL und Direkt3D, werden in VRS in Bibliotheksform eingebunden, wohingegen externe Renderingsysteme, z.B. Radiance, BMRT und POV-Ray, durch Generierung von Szenenbeschreibungen im jeweiligen Austauschformat angesprochen werden. Anzumerken ist, dass eine Anwendung, die auf der Grundlage von VRS implementiert wird, direkten Zugriff auf die Echtzeit-Renderingsysteme beibehält.

3.1.3 Hauptkategorien der Renderingkomponenten

Die Software-Architektur des generischen Renderingsystems unterscheidet fünf Hauptkategorien von Renderingkomponenten: Shapes, Attribute, Handler, Techniques und Engines. Der Entwurf dieser Komponenten wurde dabei weniger von Implementierungsüberlegungen geleitet, als vielmehr von der Frage, wie die am Rendering beteiligten Komponenten aus Anwendungsprogrammiersicht strukturiert werden können.

3.1.3.1 Shapes

Shapes im Sinne der Computergraphik repräsentieren geometrische Objekte im zwei- und dreidimensionalen Raum. Es werden im Allgemeinen polygonale Objekte, analytische Objekte, parametrische Flächen und Objektaggregate unterschieden. Im generischen Reneringsystem wird diese Definition erweitert: Shapes sind Träger von Informationen im Zielmedium. Shapes definieren weder, wie sie für das Zielmedium umgesetzt werden (z.B. graphisches Rendering), noch die Attribute, die auf sie einwirken und ihre Umsetzung mitunter modifizieren (z.B.

Texturen, die die Oberfläche eines Primitives gestalten). VRS definiert eine umfangreiche Sammlung von Standard-Shapes, die alle in Renderingsystemen üblichen zweidimensionalen und dreidimensionalen geometrischen Primitive umfasst; eine Erweiterung für audiovisuelle Shapes ist in Entwicklung. Zu den Shapes gehören u.a. geometrische Objekte, wie z. B. polygonale, analytische und parametrische Flächen. Im Fall eines audiovisuellen Renderings können hingegen auch Klänge und Geräusche als Shapes modelliert sein.

> Der Begriff „Shape“ bezeichnet im generischen Renderingsystem die Träger von Informationen im Zielmedium. Shapes definieren weder das Verfahren zu ihrer Umsetzung in die Zielmedien noch die Attribute, die sie und ihre Umsetzung gestalten.

3.1.3.2 Attribute

Attribute repräsentieren Modifikatoren, die auf Renderingkomponenten und ihre Umsetzung einwirken können. Attribute im Sinne der Computergraphik umfassen graphische und nichtgraphische Parameter der optischen und geometrischen Eigenschaften von Shapes und Szenen. Attribute steuern insbesondere die Umsetzung von Shapes (z.B. geometrische Transformation eines Shapes). Attribute sind jedoch unabhängig von den Shapes modelliert, um eine flexible Assoziation mit ihnen zu erlauben. Standard-Attribute, die VRS bereitstellt, umfassen geometrische Transformationen, optische Attribute wie Materialeigenschaften, Textur und Zeichenstil, und logische Attribute wie z.B. Gruppenzugehörigkeiten und Objekt-Identifier. Attribute sind größtenteils spezifisch für ein bestimmtes, konkretes Renderingsystem modelliert. Ein nichtphotorealistisches Renderingsystem kann z.B. das Attribut „Unsicherheit“ durch geeignete Variation in der graphischen Darstellung zum Ausdruck bringen, wohingegen ein Echtzeit-Renderingsystem das Attribut „Linien-Antialiasing“ bereitstellen wird.

> Der Begriff „Attribut“ bezeichnet im generischen Renderingsystem Modifikatoren, die auf Renderingkomponenten einwirken und diese gestalten sowie ihre Umsetzung in die Zielmedien kontrollieren. Attribute haben daher einen engen Bezug zum konkret verwendeten Renderingverfahren.

3.1.3.3 Handler

Handler repräsentieren Auswertungsalgorithmen für Shapes und Attribute, die meist einer einzelnen Shape- oder Attributklasse zugeordnet sind. Zum Beispiel repräsentieren *Painter* spezielle Handler, die für das Zeichnen eines Shapes mit Hilfe eines konkreten Renderingsystems verantwortlich sind. Insbesondere sind daher Shapes und Attribute frei von Funktionalität, die zu ihrer Umsetzung in einem konkreten Renderingsystem oder Renderingverfahren benötigt wird. Handler werden gewöhnlich von einem Anwendungsprogramm nicht direkt angesprochen, sondern von Engines aufgerufen. Handler, die für konkrete Renderingsysteme Auswertungsalgorithmen implementieren, sind Teil des zugehörigen Adapter-Subsystems. Die Funktionalität einer Engine ist hauptsächlich durch die in ihr installierten Handler festgelegt.

> Ein Handler implementiert einen Auswertungsalgorithmus für bestimmte Renderingkomponenten. Das Konzept des „Handlers“ wurde im generischen Renderingsystem eingeführt, um Auswertungsalgorithmen eigenständig als Objekte zu repräsentieren. Handler kapseln Auswertungsstrategien und lösen die Festlegung der Implementierung der Funktionalität zum Übersetzungszeitpunkt, da Handler während der Programmausführung in einer Engine installiert und deinstalliert werden können.

3.1.3.4 Techniques

Techniken repräsentieren im generischen Renderingsystem Auswertungsstrategien für Shapes und Attribute; ein Handler hingegen befasst sich mit der Auswertung eines einzelnen Shapes oder Attributs. Techniken greifen bei der Umsetzung ihrer Arbeit auf Handler zurück. Bildsynthese-Techniken zum Beispiel werten Shapes und Attribute aus, indem sie Painter verwenden, um die Shapes und Attribute auf Konstrukte des zugrundeliegenden Renderingsystems abzubilden. Techniken werden gewöhnlich von einem Anwendungsprogramm nicht direkt angesprochen, sondern von Engines aufgerufen. Techniken stellen konzeptionelle Bestandteile von Engines dar und legen maßgeblich deren Arbeitsweise fest. Techniken, die Auswertungsstrategien für ein spezielles Renderingsystem implementieren, sind Teil des zugehörigen Adapter-Subsystems.

> Eine Technik implementiert eine Auswertungsstrategie für Renderingkomponenten. Techniken ermitteln geeignete Handler für Renderingkomponenten und delegieren die Ausführung der Umsetzung in die Zielmedien an die Handler. Das Konzept der „Technik“ wurde in VRS eingeführt, um Auswertungsstrategien als eigenständige Objekte zu repräsentieren.

3.1.3.5 Engines

Engines repräsentieren Maschinen, die auf der Grundlage einer Auswertungsstrategie und Auswertungsalgorithmen Renderingkomponenten für Zielmedien auswerten. Das Konzept der Engine verallgemeinert das häufig in objektorientierten Graphiksystemen anzutreffende Konzept einer „Auswerter“-Klasse. Eine Engine verfügt über einen objektbasierten, generischen *Kontext*. Der Kontext repräsentiert zu jedem Zeitpunkt die aktuellen Attribute, die bei der Auswertung von Renderingkomponenten potentiell angewendet werden. Der Kontext ist generisch, d.h. er ist in der Lage alle Klassen von Attributen zu verwalten; er stellt hierfür eine einheitliche Verwaltungsfunktionalität bereit. Während der Auswertung von Renderingkomponenten ist der Kontext zuständig für die Assoziation der Shapes mit den Attributen.

> Eine Engine ist eine Auswertungsmaschine. Die Auswertungstrategie ist durch die in der Engine zum Einsatz gelangende Technik festgelegt, die wiederum die Auswertung der Renderingkomponenten den zugeordneten Handlern überträgt. Die Engine stellt einen generischen Kontext zur Verwaltung von Attributen, Handlern und Techniken bereit. Die Handler und Techniken, die in einer Engine installiert sind, legen die Arbeitsweise und Funktionalität dieser Engine fest.

Renderingkomponenten sind als Klassen implementiert. Im folgenden ist mit *Renderingobjekt* eine Instanz einer solchen Klasse gemeint. Abbildung 14 zeigt die Beziehungen zwischen den grundlegenden Klassen von Renderingkomponenten.

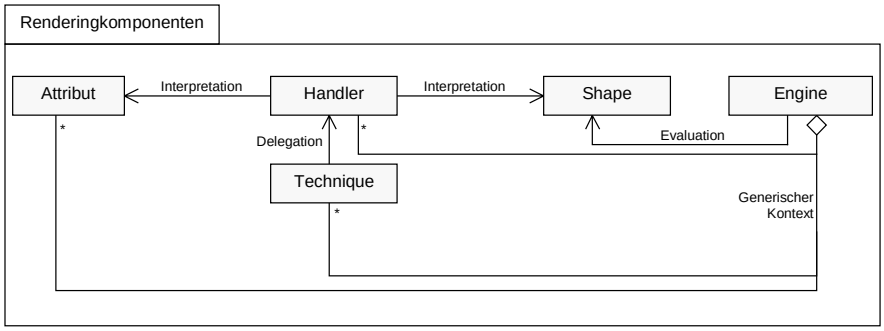

Abbildung 14. Klassenmodell der Hauptkategorien von Renderingkomponenten.

3.2 Mechanismen des generischen Renderingsystems

Eines der Schlüsselkonzepte eines generischen Renderingsystems liegt in seiner Fähigkeit, Renderingobjekte für unterschiedliche Zielmedien unterschiedlich aufzubereiten. Dies bedeutet in diesem Zusammenhang:

- Shapes und Attribute können grundsätzlich auf jedes konkrete Renderingsystem abgebildet werden, sofern ein von der Auswertungsstrategie benötigter Handler existiert und in der Engine installiert wurde.
- Shapes und Attribute, die spezifisch für eine Anwendung oder für ein konkretes Renderingsystem sind, können vollständig und gleichwertig zu den vorhandenen Shapes und Attributen mit Hilfe von spezialisierten Handlern integriert werden.
- Handler für Shapes und Attribute können spezifisch für jede Engine zur Laufzeit konfiguriert werden, indem sie in der Engine installiert bzw. deinstalliert werden.
- Techniken und Handler werden analog zu Attributen verwaltet, denn sie sind spezialisierte Attribute.

3.2.1 Generischer Kontext

Jede Engine verfügt intern über einen objektbasierten, generischen Kontext, der eine Menge von Attributen verwaltet (siehe Abbildung 15). Handler werden vom Kontext gesondert von den Attributen behandelt, obwohl Handler spezialisierte Attribute sind. Der Kontext kann sich auf wenige, generische Verwaltungsformen beschränken, z.B. Attributstacks, Attributlisten und Handler-Tabellen, um die Fülle der Attribute und Handler effektiv zu verwalten.

Die Hauptaufgabe des Kontexts ist es, während des Renderings Shapes mit Attributen und Handlern zu assoziieren. Der Kontext wird somit während des Renderings benutzt, um z.B. hierarchisch angeordnete Attribute und Handler während der Traversierung eines Szenengraphen zu verwalten. Ein Kontext stellt Attribute und Handlern, die in ihm abgelegten sind, mit Hilfe von Zugriffsoperationen bereit.

Der generische Kontext ähnelt dem graphischen Kontext von OpenGL, der als Zustandsmaschine die Renderingparameter der gesamten Rendering-Pipeline verwaltet. Dieser Ansatz bietet die feinstmögliche Kontrolle über Kontextänderungen, z.B. Attributänderungen, da eine Anwendung den Kontext inkrementell ändern und durch Nutzung der Attributkohärenz die Zahl der Änderungen minimieren kann. Insofern ähnelt eine Engine einer Zustandsmaschine: Die Zustände sind verteilt auf die Menge der Attribute und Handler. Durch Hinzufügen und Entfernen von Attributen sowie durch Installation und Deinstallation von Handlern wird der Zustand des Kontexts verändert.

Der generische Kontext ist im Unterschied zu OpenGL objektbasiert und nicht auf eine fest vorgegebenen Menge von Attributtypen beschränkt. Der Kontext operiert auf der Grundlage von Objekten und nicht auf der Grundlage atomarer Attributwerte, um die Schnittstelle hinreichend dicht zu bündeln. Die Zahl der tatsächlich von einer Anwendung im Regelfall benötigten Attributklassen und Handlerklassen ist relativ gering.

3.2.2 Kapselung von Modifikatoren durch Attribut-Objekte

Ein Attribut besteht im Allgemeinen aus einer Sammlung konzeptionell zusammengehörender Parameter, die auf die Umsetzung einwirken und diese ganz oder teilweise kontrollieren. Unter den Attributen finden sich insbesondere optische Eigenschaften und geometrische Transforma-

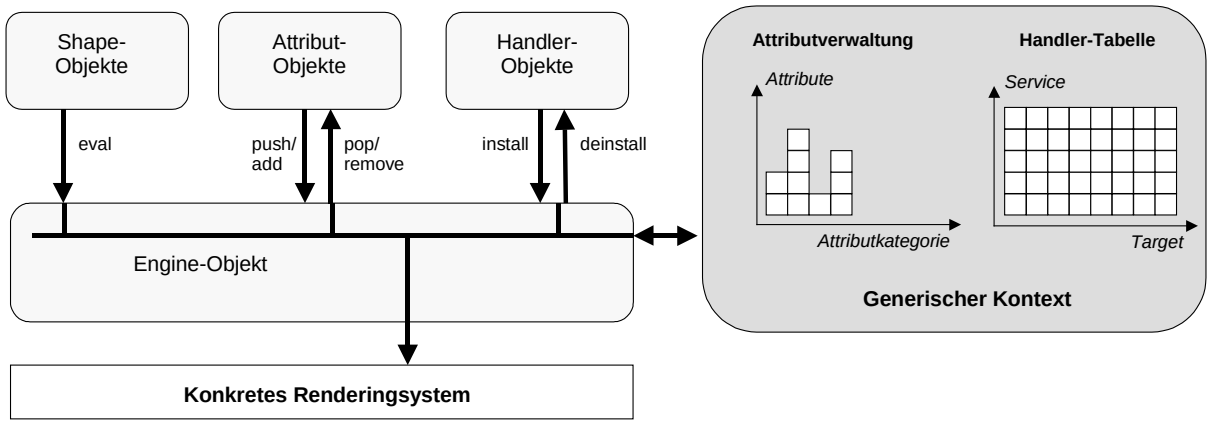

Abbildung 15. Verwaltung von Renderingkomponenten durch die Engines.

tionen. Alle hier betrachteten konkreten Renderingsysteme verfügen über wenige gemeinsame Attributtypen; zu ihnen zählen Farbe, Material, Textur und Lichtquellen. Die Attributtypen unterscheiden sich wesentlich im Hinblick auf spezielle Rendering-Eigenschaften, die an das jeweilige konkrete Renderingsystem geknüpft sind. So wird z.B. für das Renderingsystem RenderMan ein Attribut „Flächen-Shader“, das die Programmierung pixelpräziser Schattierungsverfahren unterstützt, bereitgestellt, um dieses spezielle und für RenderMan-Anwendungen wichtige Leistungsmerkmal in VRS zur Verfügung zu stellen.

Eine Attributklasse legt fest, welcher *Attributkategorie* sie angehört, unter der ihre Instanzen im generischen Kontext abgelegt werden. Ist eine Attributkategorie noch nicht vertreten, dann wird ein entsprechender Container für Objekte dieser Attributkategorie im Kontext angelegt, wenn zur Laufzeit erstmalig ein Objekt dieser Attributkategorie in den Kontext abgelegt wird. Attributkategorien werden mit Hilfe von systemweit eindeutigen Identifiern implementiert. Attribute können vom generischen Kontext mit Hilfe der Attributkategorie erfragt werden.

Im Allgemeinen definiert jede Attributklasse eine eigene Attributkategorie, z.B. legt die Attributklasse „Farbe“ die Attributkategorie „Farbe“ fest, d.h. die Kategorie ist in diesem Fall identisch mit der Klasse. Mehrere Attributklassen können jedoch eine gemeinsame Attributkategorie vereinbaren. In diesem Fall werden alle Objekte der beteiligten Attributklassen in einem gemeinsamen Container abgelegt. Diese Situation findet sich überall dort, wo Instanzen von spezialisierten Attributklassen als Instanzen ihrer gemeinsamen Basisklasse behandelt werden sollen. Zum Beispiel definieren in VRS die Lichtquellen-Attribute *PointLight*, *DistanceLight*, *SpotLight* and *AreaLight* die Attributkategorie *Lightsource*.

> Die generische Verwaltung von Attributen ist notwendig, um die Vielfalt der Attribute, die sich in den konkreten Renderingsystemen und unterschiedlichen Renderingverfahren finden, mit einem einheitlichen Mechanismus zu handhaben. Die Sammlung der Attributklassen schließt Optik- und Transformationsattribute ein, kann aber für jedes Renderingsystem und jedes Renderingverfahren erweitert werden. Ferner können Anwendungen eigene Attributklassen und Attributkategorien definieren und so den generischen Mechanismus der Attributverwaltung für sich nutzen, um anwendungsspezifische Attribute zu implementieren.

3.2.3 Kapselung von Auswertungsalgorithmen durch Handler-Objekte

Ein Handler implementiert einen Auswertungsalgorithmus für Renderingkomponenten. Handler werden analog zu Attributen in Kategorien eingeteilt; die Kategorien werden als *Services*

bezeichnet. Die Objekte, auf denen ein Handler operieren kann, werden als *Targets* bezeichnet. Die Handler-Tabelle implementiert Multidispatching, d.h. ein Verfahren, wie Methoden anhand zweier Parameter, dem Service und dem Target, ausgewählt werden. Die Auswertungsstrategie bestimmt im Allgemeinen die Services, die von einer Engine tatsächlich benötigt werden. Eine Bildsynthese-Engine, die Bilder mit Hilfe von Strahlenverfolgung erzeugt, benötigt zum Beispiel den Service *Intersektionsberechnung*. Handler für diesen Service müssen für bestimmte Targets, in diesem Fall Shape-Klassen, bereitgestellt werden.

Der generische Kontext einer Engine verwaltet Handler in einer *Handler-Tabelle*. Diese Tabelle entspricht einem zweidimensionalen Array, in dem für einen bestimmten Service und ein bestimmtes Target der entsprechende Handler eingetragen und nachgeschlagen werden kann. Jede Tabellenzelle enthält einen Stack von Handlern. Der oberste Eintrag des Stacks ist der momentan aktive Handler für einen gewünschten Service und ein gewünschtes Target. Handler lassen sich analog zu Attributen hierarchisch in einer Szenengraphenstruktur anordnen. Service und Target werden als systemweit eindeutige Identifier implementiert; sie bilden den Zugriffsschlüssel auf die Handler-Tabelle. Die Handler-Tabelle wird dynamisch zur Laufzeit aufgebaut, d.h. ein Handler-Stack wird dann alloziert, wenn zum ersten Mal ein Handler eines bestimmten Services und eines bestimmten Targets installiert wird.

Zur Illustration betrachten wir den Service „Simplifikation“. Handler dieser Art berechnen für ein gegebenes Shape eine vereinfachte Darstellung auf der Basis von weniger komplexen Shapes. Konkret heißt dies, dass z.B. analytische Shapes in eine polygonale Repräsentation umgewandelt werden. Für alle analytischen Shapes sind Simplifikations-Handler in der Handler-Tabelle eingetragen. Dieser Service ist allgemein einsetzbar und kann von allen Auswertungsstrategien in Anspruch genommen werden, falls diese für ein gegebenes Shape keine passenden Handler finden. Wird z.B. für die OpenGL-Engine versucht, ein Shape vom Typ „Kugel“ zu zeichnen, dann wird zunächst der Simplifikationsservice aufgerufen, um eine polygonale Repräsentation herzuleiten. Für die polygonale Repräsentation, d.h. für ein Polygon-Shape, existiert in der OpenGL-Engine ein Handler.

Vordefinierte Services des Virtuellen Renderingsystems sind:

- *Shape-Simplifier*: Ein Shape-Simplifier ist verantwortlich für die Zerlegung eines Shapes in eine Sammlung „einfacherer“ Shapes. Zum Beispiel zerlegt der Torus-Simplifier ein Torus-Shape in ein polygonales Netz. Dieser Mechanismus ermöglicht es, dass Renderingsysteme Shapes verarbeiten, für die sie selbst keine eigenen Handler bereitstellen. Die Arbeitsweise eines Simplifiers kann durch Attribute gesteuert werden. Zum Beispiel berücksichtigt der Torus-Painter das Detail-Attribut, das den Grad der Tessellation der Torusoberfläche festlegt.
- *Shape-Painter*: Ein Shape-Painter ist verantwortlich für die Abbildung eines Shapes in Strukturen eines konkreten Renderingsystems. Ein Shape-Painter ist stets für ein spezielles Renderingsystem geschrieben. Zum Beispiel verarbeitet der Mesh-Painter für OpenGL Mesh-Shapes dadurch, dass er die in einem Mesh-Shape enthaltenen geometrischen und graphischen Daten mit OpenGL-Anweisungen zum Zeichnen von Polygonen auswertet.
- *Attribut-Simplifier*: Ein Attribut-Simplifier ist für die Zerlegung eines Attributes in eine Sammlung „einfacherer“ Attribute verantwortlich. Attribut-Simplifier dienen dazu, ein Attribut eines konkreten Renderingsystems äquivalent für ein anderes Renderingsystem umzusetzen. Zum Beispiel kann ein POV-Ray-Texturattribut durch einen Simplifier in eine Farbe und ein Material zerlegt werden. Diese Simplifikation kann eingesetzt werden, um mit OpenGL eine Entsprechung in den Materialeigenschaften zu erhalten.

- *Attribut-Painter*: Ein Attribut-Painter ist für die Abbildung eines Attributs auf Strukturen eines konkreten Renderingsystems verantwortlich. Ein Attribut-Painter ist stets für ein spezielles Renderingsystem geschrieben. Zum Beispiel setzt der OpenGL-Attribut-Painter für Zeichenstile den entsprechenden Zeichenmodus im OpenGL-Renderingkontext.
- *Ray-Intersectors*: Ein Strahl-Objekt-Intersektor prüft, ob Shapes Schnittpunkte mit einem Strahl besitzen. Das Ergebnis wird in Form einer Liste von Intersektionspunkten zurückgegeben. Dieser Service wird z.B. beim interaktiven Abfragen von Shapes benötigt, um zu ermitteln, auf welche Shapes mit der Maus geklickt wurden.

Handler werden mit Hilfe einer Installierungs- und Deinstallierungsoperation im Kontext verwaltet. Handler können von Anwendungsprogrammen für jede Engine individuell festgelegt werden. Die Engines zur Bildsynthese mit konkreten Renderingsystemen, z.B. die OpenGL-Engine oder die RenderMan-Engine, konfigurieren sich zum Konstruktionszeitpunkt, indem sie alle Default-Handler, die für sie in ihrer Klasse (in einem statischen Klassenattribut) registriert wurden, in ihre Handler-Tabelle übernehmen. Insofern sind Handler in Anwendungsprogrammen meist unsichtbar, da die Default-Handler der Engines im Allgemeinen ausreichen. Darüber hinaus kann jedes Anwendungsprogramm die Default-Einstellungen durch Installation eigener Handler überschreiben, um anwendungsspezifische oder optimierte Varianten von Auswertungsalgorithmen festzulegen.

Handler werden meist implizit aufgerufen, wenn ein Attribut in den Kontext einer Engine abgelegt oder ein Shape von der Engine verarbeitet wird. Wird zum Beispiel ein Material-Attribut abgelegt, dann wird einerseits von der Engine der zugehörige Attributstack im Kontext aktualisiert und andererseits versucht, einen Attribut-Painter oder –Simplifier zu finden.

> Die Konfiguration der Handler-Tabelle einer Engine erfolgt zur Laufzeit und bietet somit eine hohe Flexibilität bezüglich der den Shapes und Attributen zugeordneten Auswertungsalgorithmen. Durch die Faktorisierung von Funktionalität (Handler) und Deklaration (Shapes und Attribute) lassen sich Shape- und Attributklassen allgemein entwerfen, ohne die Details einer Implementierung für ein spezielles Renderingsystem kennen zu müssen. Die Schnittstellen der Shape- und Attributklassen können sich auf deklarative Elemente konzentrieren, die für die Nutzbarkeit und Verständlichkeit aus Entwicklersicht entscheidend sind.

3.2.4 Kapselung von Auswertungsstrategien durch Technik-Objekte

Sprechen wir von „Auswertung“ im Kontext des generischen Renderingsystems, so ist damit nicht notwendig die Bildsynthese in Verbindung mit einer Projektion einer dreidimensionalen Szene auf eine zweidimensionale Zeichenfläche gemeint, obwohl diese Form der Auswertung eine der Hauptanwendungen einer Engine darstellt. Im folgenden sprechen wir allgemeiner von Auswertung und bezeichnen damit die Auswertung einer Folge von Shapes und Attributen durch eine Engine auf der Grundlage einer Auswertungsstrategie, die durch ein Technikobjekt repräsentiert wird. Der Begriff „Auswertungsstrategie“ ist hier weit gefasst: Darunter fallen Bildsyntheseverfahren, Bildanalyseverfahren und weitere, abstrakte Umsetzungen und Untersuchungen von graphisch-geometrischen Sachverhalten.

Technikklassen sind spezialisierte Attributklassen. Analog zu anderen Attributen, unterhält jede Engine in ihrem generischen Kontext einen Container für Technik-Objekte, konkret einen Stack. Das Top-Objekt dieses Stacks repräsentiert die aktiven Technik, die eine Engine zur Auswertung von Shapes und Attributen einsetzt.

Mit Hilfe von Technik-Objekten können von einer Engine unterschiedliche Auswertungsstrategien für Renderingobjekte angewendet werden. Jede Technik-Klasse arbeitet im Allgemeinen eng mit Handlern eines bestimmten Service zusammen. Engines zusammen mit Handlern und Techniken stellen somit die Kernkomponenten dar, mit denen das generische Renderingsystem implementiert ist.

Techniken des generischen Renderingsystems umfassen:

- *Image-Synthesis-Technik.* Diese Technik, die von Engines zur Bildsynthese eingesetzt wird, ermittelt den Shape-Painter zu einem auszuwertenden Shape bzw. den Attribut-Painter zu einem Attribut, das aktiviert wird. Dadurch wird die Abbildung von Shapes und Attributen an die Painter delegiert.
- *Ray-Request-Technik.* Die Ray-Request-Technik ermittelt Strahl-Objekt-Intersektionen, indem sie einen Handler vom Service *Ray-Intersector* zu einem auszuwertenden Shape ermittelt und die Ergebnisse, d.h. die Intersektionspunkte und Shape-Identifier, zusammenstellt.
- *Image-Analysis-Technik.* Eine Image-Analysis-Technik berechnet die Position, die Ausdehnung und die Fläche eines Shapes in einem Bild zu einer gegebenen Kameraeinstellung. Diese Technik arbeitet rein analytisch und erzeugt kein Bild.

Im Allgemeinen wendet jede Technik ein rekursives Verfahren zur Auswertung von Shapes an. Wird ein Shape an eine Engine gesendet, dann wird die aktive Technik versuchen, einen passenden Handler zu ermitteln. Wird kein passender Handler gefunden, wird die Technik im Allgemeinen versuchen, einen Simplifier für das Shape zu finden. Ist kein Simplifier für das Shape verfügbar, dann wird rekursiv die Auswertung mit der Elternklasse des Shapes als Target wiederholt. Dieser Prozess setzt sich fort, bis ein passender Handler gefunden wird oder die Wurzelklasse erreicht wird. In diesem Fall kann das Shape mit der gegebenen Engine nicht behandelt werden und wird ignoriert.

> Techniken und die mit ihnen kooperierenden Handlern sind kennzeichnend für die Software-Architektur des generischen Renderingsystems: Shape- und Attributklassen müssen nicht über Methoden, die in den Basisklassen für Shapes und Attribute festzulegen wären, verfügen, die eine Fülle von Grundfunktionen festzulegen hätten. Stattdessen sind die Basisklassen für Shapes und Attribute bezüglich ihrer Implementierung leichtgewichtig, da ein Großteil der Funktionalität in Handlern und Techniken enthalten ist. Zugleich steht es Anwendungsprogrammen und spezialisierten Auswertungsstrategien frei, diesen Funktionsumfang anwendungsspezifisch zu erweitern, ohne dabei die Software-Architektur des generischen Renderingsystem ändern zu müssen.

3.2.5 Assoziierung von Shapes und Attributen

Im vorgestellten Ansatz sind Shape-Klassen und Attribut-Klassen getrennt voneinander modelliert und erklären keine Assoziationen untereinander. Konkret bedeutet dies, dass zum Beispiel ein Shape-Objekt nicht die Attribut-Objekte kennt, die auf seine Umsetzung im Rahmen einer Auswertung einwirken. Stattdessen werden Shape-Objekte und Attribut-Objekte während der Auswertung indirekt mit Hilfe des generischen Kontexts assoziiert.

Der Zugriff auf den Kontext und damit auf die Attribut-Objekte geschieht durch die Handler, die Zugriff auf den generischen Kontext besitzen. Wird ein Shape-Objekt durch die Engine evaluiert, dann entscheidet der aufgerufene Handler, welche der Attribute für ihn relevant sind

und folglich mit in die Auswertungen eingehen. Ein Polygon-Painter kann zum Beispiel alle optischen Attribute berücksichtigen; ein Kugel-Painter könnte zusätzlich das Detaillierungs-Attribut auswerten, das den Grad der Tessellation der Kugel vorgibt.

Durch die Trennung von Shapes und Attributen wird erreicht, dass Shape-Objekte „leichtgewichtig“ [14] werden – sie speichern keinerlei Werte für Optik- und Transformationsattribute. Sie besitzen ausschließlich solche Attribute, die ihre Geometrie bestimmen. Das leichtgewichtige Design stellt sicher, dass Objekte so speichereffizient wie möglich sind, sodass sie in großer Zahl von Anwendungen eingesetzt und effizient implementiert werden können. Des Weiteren ist in diesem Design die Menge der Attribute, die ein Shape besitzt, nicht festgelegt, sodass etwa bei der Integration neuer Renderingverfahren und der damit verbundenen Einführung neuer Attributklassen das Design der Shape-Klassen unberührt bleibt.

Sowohl Attribute als auch Shapes werden durch Objektreferenzen an die Engine und den generischen Kontext übermittelt. Die Attribut-Container des Kontexts sind insofern Speicherplatz-unkritisch, da Attribut-Objekte durch eine Objektreferenz abgelegt werden; es wird also zu keinem Zeitpunkt der Inhalt eines Attribut-Objekts kopiert. Dabei wird unterstellt, dass alle Objekte im generischen Renderingsystem grundsätzlich mehrfach referenzierbar sind und die Speicherverwaltung automatisch erfolgt. Konzepte hierfür sind in allen objektorientierten Programmiersprachen vorhanden, in C++ in Form von Smart-Pointers [65] und in Java durch die dort vorhandene automatische Garbage Collection.

> Shapes und Attribute werden zur Laufzeit über den generischen Kontext assoziiert, wobei die Auswahl relevanter Attribute für die Umsetzung eines konkreten Shapes den Handlern, die das Shape auswerten, obliegt. Dadurch ist die Assoziation von Attributen mit Shapes frei gestaltbar und erweiterbar. Die Trennung von Shapes und Attributen in der Modellierung erlaubt eine leichtgewichtige Implementierung der Shape-Klassen.

3.2.6 Steueranweisungen der Engines

Die Schnittstelle einer Engine stellt Operationen zur Verfügung, die den generischen Kontext verwalten und die Auswertung von Shapes ermöglichen. Die tatsächliche Anzahl der Methoden ist auf Grund der Methoden-Überladung und der Objekt- und Argument-Polymorphie erstaunlich gering. Die Schnittstelle ist im folgenden skizziert:

```
class REngine {

public:
  void push(RMonoAttribute attr);
  void pop(RMonoAttribute attr);

  void add(RPolyAttribute attr);
  void remove(RPolyAttribute attr);

  void install(RHandler hdl);
  void deinstall(RHandler hdl);

  void eval(RShape shape);

  ...
};
```

Für Monoattribute, die in Stacks verwaltet werden, existieren *push*- und *pop*-Operationen; für Polyattribute, die in Listen verwaltet werden, existieren *add*- und *remove*-Operationen (siehe Abschnitt 3.4.1). Für Handler existieren *install*- und *deinstall*-Operationen. Zur Auswertung von Shapes legt die Engine-Klasse die *eval*-Operation fest.

Für geometrische Transformationen, die als spezialisierte Attributklassen modelliert werden, wurde eine Reihe von Methoden zur direkten Manipulation von Transformationsstacks eingeführt. Die Methoden lassen sich für einen der vier Matrixstacks anwenden, die analog zur OpenGL-Architektur gewählt wurden; der Model-View-Matrixstack ist dabei der Default-Stack. Daneben existiert ein Stack für die Projektionsmatrix, ein Stack für die Farbraum-Matrix und ein Stack für die Textur-Matrix. Die zugehörige Schnittstelle besitzt folgenden Aufbau:

```
class REngine {
public:

  enum MatrixStack { ModelView, Projection, Texture, Color };

  void pushMatrix(MatrixStack stack = ModelView);
  void popMatrix(MatrixStack stack = ModelView);

  void loadMatrix(RMatrix m, MatrixStack stack = ModelView);
  void multMatrix(RMatrix m, MatrixStack stack = ModelView);
  void scaleMatrix(RVector scaling, MatrixStack stack = ModelView);
  void translateMatrix(RVector translation, MatrixStack stack = ModelView);

  ...
};
```

Insbesondere für Echtzeit-Graphiksysteme besteht hier die Möglichkeit, in spezialisierten Engine-Klassen eine optimierte Implementierung durch Methodenüberladung bereitzustellen. Im Fall von OpenGL geschieht dies im Virtuellen Renderingsystem so, dass alle geometrischen Transformationen direkt an OpenGL und damit ggf. an eine unterstützende Hardware weitergeleitet werden. Geometrische Transformationen haben eine für das Echtzeit-Rendering herausgehobene Stellung, da jede Form der Transformationsberechnung in Software signifikant die Renderingeffizienz beeinträchtigen würde. Das Virtuelle Renderingsystem erhebt hier den Anspruch, nicht nur zu einer Abstrahierung des Renderingprozesses beizutragen, sondern auch effiziente, in der Praxis nutzbare Implementierungskonzepte bereitzustellen. Die Anwendung der Steuerkommandos soll im Folgenden anhand einer Reihe von Beispielen illustriert werden.

3.2.6.1 Rendering attributierter Shape-Objekte

Die Grundaufgabe des generischen Renderingsystems besteht in der Auswertung attributierter Shapes. In diesem Beispiel werden zwei Attribut-Objekte, ein Materialobjekt und ein Rotationsobjekt, auf den generischen Kontext einer Engine gelegt. Anschließend wird ein Shape-Objekt, eine Superquadrik, evaluiert. Zur Evaluation wird intern ein Handler herangezogen, der auf der Basis der Optik- und Transformationsattribute die Superquadrik in Form geometrischer Primitive rendert. Im Beispiel werden zunächst die zwei Attribut-Objekte und das Shape-Objekt erzeugt, bevor diese Objekte durch die Engine ausgewertet werden.

```
proc Example (REngine e) {
  RMaterial mtl = new RMaterial(0.2,0.8,0.0);  // Reflexionskoeffizienten
  RRotation rot = new RRotation(30, RVector(0,1,0)); // Winkel und Achse
  RSuperQuad sq = new RSuperQuad(0.9,0.9); // Rundungsparameter

  e.push(mtl);
  e.pushMatrix();
  e.multMatrix(rot.getMatrix());
  e.eval(sq);
  e.popMatrix();
  e.pop(mtl);
}
```

Die gezeigte Folge der Engine-Steuerkommandos ist generisch, d.h. sie kann für jede Art von Engine und damit für jedes adaptierte, konkrete Renderingsystem in dieser Form unmittelbar genutzt werden, ohne auf die speziellen Schnittstellenaspekte einzelner Renderingsysteme eingehen zu müssen.

In Abhängigkeit von der aktiven Technik der Engine könnte diese Folge von Steuerkommandos die Bildsynthese, eine geometrische Berechnung oder eine analytische Berechnung ausführen. Die Wahl der Handler ist Aufgabe der aktiven Technik – die Technik entscheidet, was „Auswertung“ in der jeweiligen Situation bedeutet.

Im Fall einer Bildsynthese-Technik für eine OpenGL-Engine würden in diesem Beispiel folgende Arbeitsschritte ausgeführt: Das Material-Objekt würde durch einen Material-Handler zur Aktualisierung der OpenGL-Materialparameter führen. Die Transformation würde die OpenGL Model-View-Matrix sichern und mit der gegebenen Transformationsmatrix multiplizieren. Die *eval*-Methode würde zunächst einen Simplifier aufrufen, der die Superquadrik in ein polygonales Netz zerlegt und an die Engine zurückgibt, die dann wiederum rekursiv versucht, dieses Netz zu evaluieren. Die Technik würde dann den OpenGL-spezifischen Netz-Painter verwenden, um das polygonale Netz mit Hilfe von OpenGL-Kommandos zu rendern. Abschließend würden die ursprüngliche Model-View-Matrix und Materialparameter wieder hergestellt.

3.2.6.2 Installation von Handlern

Änderungen in der Handler-Tabelle sind möglich, d.h. die Default-Handler für Shapes und Attribute können dynamisch ausgetauscht werden. Eine Motivation hierfür kann der Wunsch nach einer optimierten Implementierung eines Auswertungsalgorithmus sein. In dem folgenden Beispiel stellt die Anwendung einen spezialisierten Shape-Painter für Superquadriken bereit, der diese optimal für OpenGL umsetzt. Durch diesen Shape-Painter wird die Vereinfachung einer Superquadrik in polygonale Netze umgangen, denn die Bildsynthesetechnik versucht zunächst immer einen Painter zu finden und nur, falls kein Painter gefunden wird, initiiert sie die Simplifikation. Durch die Installation wird der Handler in die Handler-Tabelle mit dem Service „Shape-Painter“ und dem Target „Superquad“ eingetragen. Wird nun die unten aufgeführte Folge von Steueranweisungen für eine Engine ausgeführt, dann ermittelt die Bildsynthesetechnik den Shape-Painter und prüft, ob der Shape-Painter für die konkrete Engine anwendbar ist. Handelt es in diesem Beispiel nicht um eine OpenGL-Engine, dann wird das Shape simplifiziert.

```
proc Example (REngine e) {
  RSuperQuad sq = new RSuperQuad();
  RSuperQuadPainterGL sqpainter = new RSuperQuadPainterGL();

  e.install(sqpainter);
  e.eval(sq);
  e.deinstall(sqpainter);
}
```

Auch die hier gezeigte Folge von Steuerkommandos ist generisch, obwohl ein Renderingsystem-spezifischer Painter verwendet wird. Jeder Handler kann in jeder Engine installiert werden, aber nicht jeder Handler ist von jeder Engine benutzbar. Ist ein Handler für eine bestimmte Engine nicht anwendbar, dann zeigt die Installation (bzw. Deinstallation) keine Wirkung; der Handler wird ignoriert.

3.2.6.3 Kontextabhängige Attribut-Berechnung

Attribute im generischen Renderingsystem können so definiert sein, dass sie ihre Werte erst zum Traversierungszeitpunkt auf der Grundlage des momentanen Zustandes des generischen Kontexts berechnen; die Ausdruckskraft von Attributen wird dadurch erhöht. Insbesondere wird es möglich, Attribute zu definieren, deren Wertberechnung von weiteren Attributen abhängt. Insofern sind Attribute des generischen Renderingsystems mehr als "Sammlungen von Parameter-
werten".

In dem folgenden Beispiel wird eine geometrische Transformation verwendet, deren Matrixkoeffizienten erst zum Traversierungszeitpunkt berechnet werden können. Die geometrische Transformation transformiert das aktuelle lokale Koordinatensystem derart, dass eine ausgewählte Achse (z.B. die z-Achse) stets orthogonal zur Kameraebene steht; diese Form der automatischen Orientierung zur Kamera ist unter dem Namen *Billboarding* bekannt. Das Billboarding wird unter anderem zur automatischen Ausrichtung von Texten und Symbolen in Richtung des Betrachters eingesetzt. Ein Billboarding-Objekt berechnet in der *getMatrix*-Methode in Abhängigkeit von der momentanen Kameraposition, die über den generischen Kontext der Engine verfügbar ist, die entsprechende Rotationsmatrix.

```
proc Example (REngine e) {
  RBillboard bb = new RBillboard();
  RSuperQuad sq = new RSuperQuad();

  e.pushMatrix();
  e.multMatrix(bb.getMatrix(e));
  e.eval(sq);
  e.popMatrix();
}
```

Kontextabhängige Attribute lassen sich einheitlich in das Attributmanagement des generischen Renderingsystems eingliedern. Insbesondere erlaubt dieser Ansatz, Attribut-Objekte mit „Intelligenz“ zu versehen, die sich darin äußern kann, dass Attributwerte erst zum Zeitpunkt des Einsatzes eines Attributs berechnet werden. Komplexe Berechnung können so unter einer einheitlichen und einfachen Schnittstelle zur Verfügung gestellt werden. In dieser Hinsicht unterscheidet sich der hier gewählte Ansatz zur Modellierung von Attributen von der Software-Architektur anderer computergraphischer Systeme, die unter Attributen im Allgemeinen eine Sammlung von Datenfeldern verstehen.

> Engines verfügen über wenige, polymorphe Methoden zur Manipulation des generischen Kontexts. Die Zahl der Methoden ist aufgrund der objektbasierten Repräsentation aller Renderingobjekte gering. Aus Anwendungssicht sind Folgen von Steueranweisungen im Allgemeinen unabhängig von der tatsächlich zum Einsatz gelangenden Auswertungsstrategie formulierbar, da alle Implementierungsaspekte indirekt durch Handler-Objekte und Technik-Objekte definiert und kontrolliert werden.

3.2.7 Mikroprogrammierung mit Renderingkomponenten

Die Renderingobjekte des generischen Renderingsystems stellen keine Modellierungsstrukturen für Szenen bereit; die Renderingobjekte sind als elementare, zueinander orthogonale Komponenten entworfen. Es lassen sich jedoch leicht auf ihrer Basis Modellierungsstrukturen entwerfen, deren Aufgabe es sein kann, Strukturen für die Organisation von Szeneninhalten und damit für die Assoziation von Shapes und Attributen bereitzustellen. Eine allgemeine Diskussion von verschiedenen Szenenrepräsentationsformen findet sich bei Beier [9].

In folgendem Zusammenhang sprechen wir von *Mikroprogrammierung*: die Steueranweisungen der Engine werden im Allgemeinen nicht direkt von einer Anwendung spezifiziert, sondern werden zur Implementierung abstrakterer Komponenten, z.B. Szenenmodellen, eingesetzt; eine Folge von Steueranweisungen entspricht einem *Mikroprogramm* des generischen Renderingsystems. Unterschiedliche Assoziationskonzepte können auf diese Art gleichermaßen effizient und in Abhängigkeit von den Anforderungen einer Anwendung implementiert werden.

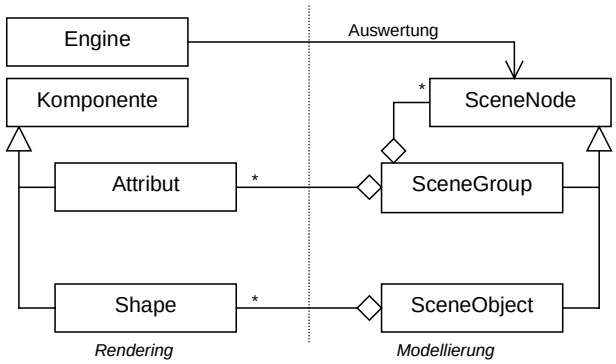

Abbildung 16. Klassenmodell einer hierarchischen Szenenrepräsentation.

3.2.7.1 Hierarchische Szenenmodellierung

Für eine *hierarchische Szenenmodellierung* wird gemeinhin eine Szenengraph-Darstellung verwendet (z.B. in OpenInventor). Eine Szenengraph-Darstellung kann objektbasiert durch Szenengraphknoten implementiert werden, die Container für Renderingobjekte darstellen. Konkret können innere Knoten eines Szenengraphen Attribut-Objekte enthalten, während Blattknoten Shape-Objekte aufnehmen. Bei der Traversierung werden die in einem Knoten enthaltenen Renderingobjekte durch Steueranweisungen ausgewertet. Das Klassendiagramm einer solchen hierarchischen Szenenrepräsentation findet sich in Abbildung 16.

Das folgende Code-Fragment skizziert eine Szenengraphen-Implementierung auf der Basis des Virtuellen Renderingsystems; sie ist weder vollständig noch effizient, sondern dient der Illustration der Mikroprogrammierung. Die Basisklasse für Szenengraphknoten *SceneNode* definiert eine Traversierungsmethode *traverse*, die als Argument eine Engine erwartet.

```
class SceneNode {
public:
  void traverse(REngine e) = 0;
};
```

Innere Knoten eines Szenengraphen, die Gruppenknoten vom Typ *SceneGroup*, enthalten eine Menge von Attribut-Objekten und können weitere Kindknoten besitzen. Die Attribut-Objekte wirken sich auf die Subgraphen aus. Bei der Traversierung werden zunächst alle Attribut-Objekte (im Beispiel eingeschränkt auf Monoattribute) gepusht, dann die Subgraphen rekursiv evaluiert und schließlich die Attribut-Objekte wieder gepoppt[*].

```
class SceneGroup : public SceneNode {
  array<RMonoAttribute> attr;
  array<SceneNode> children;
public:
  void traverse(REngine e) {
    int i;
    for (i=0; i<attr.size(); i++) e.push(attr[i]);
    for (i=0; i<children.size(); i++) children[i].traverse(e);
    for (i=attr.size()-1; i>=0; i--) e.pop(attr[i]);
  }
};
```

[*] Angemerkt sei, dass die *push*-Methode im Fall von Transformationsattribut-Objekten per Default-Argument den Model-View-Matrixstack sichert und die Transformationsmatrix mit dem gegenwärtigen Inhalt der Model-View-Matrix multipliziert; die *pop*-Methode restauriert analog den Inhalt des Model-View-Matrixstacks. Außerdem sind für die objektinternen Arrays entsprechende Zugriffsmethoden zu definieren, die hier weggelassen wurden, um das Beispiel kompakt zu halten.

Blattknoten vom Typ *SceneObject* enthalten Shape-Objekte; sie repräsentieren die Szenengeometrie. Ihre Traversierungsmethode sendet alle enthaltenen Shape-Objekte an die Engine zur Auswertung.

```
class SceneObject : public SceneNode {
  array<RShape> shapes;
public:
  void traverse(REngine e) {
    for (int i=0; i<shapes.size(); i++) e.eval(shapes[i]);
  }
};
```

Im letzten Beispiel wird ein Szenengraph konstruiert, der ein Shape-Objekt (Superquad) mit zwei Attribut-Objekten (Material und Rotationstransformation) verknüpft.

```
SceneObject leafNode = new SceneObject();
leafNode->append(new RSuperQuad());

SceneGroup rootNode = new SceneGroup();
rootNode->append(new RMaterial(0.2,0.8,0.0));
rootNode->append(new RRotation(30, RVector(0,1,0)));
rootNode->append(leafNode);
```

Durch die hierarchische Modellierung wird insbesondere die Kohärenz in der Attributierung darstellbar und die Zahl der Kontext-Änderungen kann minimiert werden. Dadurch lassen sich Echtzeit-Renderingsysteme wie OpenGL effizient ansteuern, da in ihnen Kontextänderungen generell teure Operationen darstellen. Der Nachteil dieser Modellierung liegt darin, dass nur während der Traversierung die tatsächlich auf ein Shape einwirkenden Attribute ermittelt werden können.

3.2.7.2 Nichthierarchische Szenenmodellierung

Eine *nichthierarchische Szenenmodellierung* ordnet einem oder mehreren Shape-Objekten eine Menge von Attribut-Objekten direkt zu. Eine Szene kann in dieser Form als Liste von derart mit Attribut-Objekten assoziierten Shape-Objekten repräsentiert werden. Zur Auswertung eines Szenenobjektes werden zunächst alle Attribut-Objekte dem generischen Kontext mitgeteilt. Die Geometrie des Szenenobjektes wird anschließend evaluiert. Zuletzt werden alle Attribut-Objekte wieder vom generischen Kontext entfernt. Das Klassendiagramm einer solchen Szenenrepräsentation findet sich in Abbildung 17.

```
class SceneObject {
  array<RMonoAttribute> attr;
  array<RShape> shapes;
  SceneObject nextObj;

public:
  void evaluate(REngine e) {
    int i;
    for (i=0; i<attr.size(); i++) e.push(attr[i]);
    for (i=0; i<shapes.size(); i++) e.eval(shapes[i]);
    for (i=attr.size()-1; i>=0; i--) e.pop(attr[i]);

    if(nextObj!=NULL) nextObj.evaluate(e);
  }
};
```

Der Vorteil dieser Szenenmodellierung liegt darin, dass zu jedem Szenenobjekt alle auf ihn einwirkenden Attribut-Objekte direkt zugreifbar sind; dies vereinfacht insbesondere die Editierung der Attribut-Objekte in einer Szenenbeschreibung. Der Nachteil liegt in der fehlenden Sichtbarkeit gemeinsamer Attribute, d.h. Kohärenzinformation für Attribute ist nicht vorhanden. Eine Optimierung der Attributauswertung könnte durch Attribut-Painter erfolgen, indem diese vor der Auswertung eines Attribut-Objekts prüfen, ob dieses Attribut-Objekt bereits das

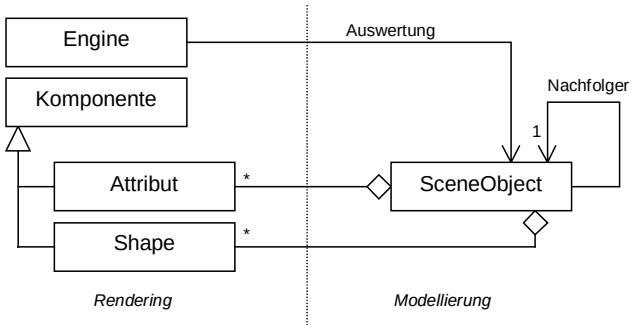

Abbildung 17. Klassenmodell einer nichthierarchischen Szenenrepräsentation.

aktive Attribut-Objekt im generischen Kontext darstellt und ob sich zwischenzeitlich der Zustand des Attribut-Objekts geändert hat. Attribut-Objekte müssen dazu eine Transaktionsnummer besitzen, die Aufschluss darüber gibt, wann das Objekt das letzte Mal geändert wurde. Die Transaktionsnummer ist ein systemweit eindeutiger Identifier, auf dem eine zeitliche Ordnung definiert ist.

Das Konzept der Mikroprogrammierung auf der Grundlage der Engine-Steuerkommandos führt bei der Szenenmodellierung zu einer strikten Trennung von Struktur und Inhalt: Die Strukturkomponenten (z.B. Knoten) enthalten Inhaltskomponenten, die Renderingobjekte. Die Unterscheidung von Shapes, Attributen und Handlern bei den Renderingobjekten ermöglicht eine minimale „Renderingsprache“, die mit wenigen polymorphen Steueranweisungen der Engine-Klasse implementiert wird. Dabei ist die Ausdruckskraft dieser Renderingsprache hoch und favorisiert nicht eine bestimmte Art der Szenenmodellierung. So kann anwendungsabhängig eine optimale Szenenmodellierung realisiert werden.

3.3 Entwurf der Shape-Klassen

In den meisten objektorientierten Renderingsystemen hängt der Entwurf der Shape-Klassen von dem verwendeten Renderingverfahren ab. Ein Renderingsystem auf der Basis von Ray-Tracing kann zum Beispiel implizit definierte Flächen als geometrische Primitive bereitstellen, während ein Echtzeit-Renderingsystem meist nur polygonale Objekte unterstützen wird. Im generischen Renderingsystem muss bei dem Entwurf der Shape-Klassen darauf geachtet werden, dass keine Annahmen über das Auswertungsverfahren den Entwurf mitbestimmen. Insbesondere steht die Nutzbarkeit im Vordergrund, so dass, vergleichbar zu RenderMan, versucht wird, eine möglichst umfangreiche Sammlung allgemein gültiger Shape-Klassen bereitzustellen. Wesentliche Kriterien, die bei dem Entwurf der Shape-Klassen des generischen Renderingsystems berücksichtigt wurden, sind:

- *Implementierungsunabhängige Vererbung*. Shape-Klassen sollen in Vererbungsbeziehung stehen, wenn sie eine gemeinsame Schnittstelle und ein gemeinsames Verhalten besitzen. Eine Ähnlichkeit in der Implementierung ist nicht maßgeblich. Würde diese Ähnlichkeit als Kriterium genutzt, dann könnte es zu Semantik-Inkonsistenzen kommen, die im Allgemeinen in der abgeleiteten Klasse durch Methoden-Überschreibung entschärft werden müssten [84]. Würde zum Beispiel eine Klasse „Quadrat“ von einer Klasse „Rechteck“ abgeleitet, dann könnten die unabhängigen Operationen der Rechteckklasse zum Setzen der Breite und Höhe bei Quadrat-Objekten zu einem Widerspruch führen.

- *Renderingunabhängige Darstellung*. Shape-Klassen sollten nicht auf Grund ähnlicher Renderingalgorithmen umgesetzt werden, da diese Ähnlichkeit meist an ein bestimmtes Renderingverfahren geknüpft ist. Würde zum Beispiel eine Klasse „Kugel“ von der Klasse „Mesh“ abgeleitet, weil unterstellt wird, dass Kugeln mit Hilfe von Polygonen dargestellt werden, dann träfe dies nicht für ein auf Ray-Tracing basierendes Renderingsystem zu; dort könnten Kugeln als implizit definierte Flächen dargestellt sein.
- *Erweiterbarkeit der Shape-Klassen*. Das generische Renderingsystem muss gewährleisten, dass anwendungsspezifische Shape-Klassen nahtlos und effizient integriert werden können. Wenn anwendungsspezifische Shape-Klassen nicht integriert werden könnten, dann müssten derartige Shapes durch eine Zerlegung in vorhandene Shapes konvertiert werden. Eine Anwendung, die zum Beispiel 3D-Pfeile in großer Menge benutzt, müsste jeden Pfeil durch einen Zylinder und einen Kegel darstellen. Zwei Probleme treten auf: 1) Die Anwendung speichert redundante Information, pro Pfeil einen Kegel und einen Zylinder, und selbst bei Mehrfachreferenzierung eines Standardpfeils müsste eine Transformationsmatrix vorgehalten werden. 2) Die Pfeilsemantik wird durch die Zerlegung nicht erhalten, die jedoch im Fall von Interaktionen benötigt oder zur Optimierung des Renderings eingesetzt werden könnte. Im Idealfall sollte die Shape-Klassenhierarchie um eine neue Klasse „Pfeil“ ergänzt werden können; durch Implementierung entsprechender Shape-Painter können optimale Renderingalgorithmen für konkrete Renderingsysteme bereitgestellt werden und mit Hilfe eines Shape-Simplifiers kann die Zerlegung in elementare Shapes für solche konkreten Renderingsysteme erfolgen, für die kein Painter zur Verfügung steht.

Wir können allerdings nicht erwarten, dass eine allgemein gehaltene Shape-Klassenhierarchie den Anforderungen der Implementierung der einzelnen Renderingsysteme optimal entgegenkommt. Jedoch steht für die Software-Architektur des generischen Renderingsystems die Verwendbarkeit der Shape-Klassen im Vordergrund.

3.3.1 Separation der Datenrepräsentation in Shapes

Shapes benötigen zur Spezifikation ihrer Geometrie geometrische und graphische Daten. Im Fall polygonaler Shapes sind dies insbesondere Eckkoordinaten und weitere Eckinformationen wie z.B. Eckfarbe, Ecktexturkoordinaten und Ecknormale. Shapes können ihre Daten intern speichern, zum Beispiel in Form von Arrays. Die interne Speicherung von Shape-Daten hat jedoch den Nachteil, dass eine Anwendung diese Datenstruktur nicht selbst bestimmen kann. Daten müssen in diesem Fall von einer Anwendungsdatenstruktur konvertiert und im Shape explizit abgelegt werden.

Im generischen Renderingsystem wird ein anderer Weg bestritten: Die Speicherung von Shape-Daten wird von der Shape-Klasse separiert, um sicherzustellen, dass Anwendungsprogramme Shapes mit den für sie optimalen Datenstrukturen ausstatten können. Das Klassenmodell des im folgenden diskutierten Iterator-Konzepts ist in Abbildung 18 dargestellt.

3.3.1.1 Iteratoren zum Zugriff auf Daten

Shape-Klassen im generischen Renderingsystem beziehen alle für ihre Definition notwendigen Daten durch Iteratoren. Ein *Iterator* ist ein Objekt, das einen sequentiellen Zugriff auf Datenelemente zur Verfügung stellt [34]. Iteratoren verstecken die interne Repräsentation von Daten durch eine einheitliche *Zugriffsschnittstelle*. Iteratoren können auf Datenstrukturen wie z.B. Array, Liste und Baum operieren oder prozedural Daten berechnen. Da Shapes und Iteratoren über eine einheitliche Schnittstelle verbunden sind, können Shapes ihre Daten in unterschiedlichen Datenstrukturen ablegen, ohne dabei die Schnittstelle der Shapes zu komplizieren. Auf

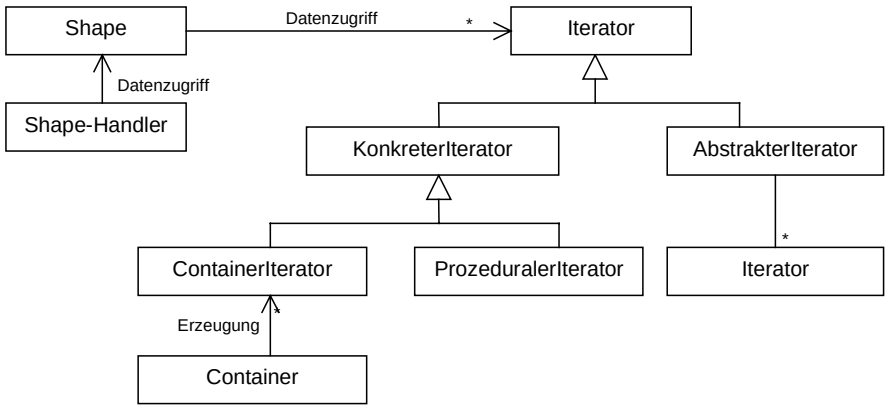

Abbildung 18. Klassenmodell der Shape-Iteratoren.

diese Weise können z.B. direkt anwendungsspezifische Datenstrukturen als Datenquellen für Shapes verwendet werden, vorausgesetzt, es wird ein spezieller Iterator für diese Datenstrukturen implementiert, d.h. die Renderingverfahren operieren so indirekt auf den anwendungsspezifischen Datenstrukturen. Iteratoren vermeiden in diesem Zusammenhang insbesondere Probleme der Datenintegrität. Als weitere, unmittelbare Anwendung dieses Konzepts kann eine Datenquelle mehrere unabhängige Iteratoren bereitstellen und dadurch mehrere Shapes an die gleiche Datenquelle simultan anschließen. Der Iterator-Ansatz wurde durch die C++-Standardbibliothek STL [80] motiviert.

Die Iterator-Schnittstelle definiert Methoden für das Zurücksetzen des virtuellen Cursors, das Voranschreiten um ein Datenelement, das Prüfen auf weitere Datenelemente und den Zugriff auf das aktuelle Datenelement:

```
template<class T> class RIterator {
public:
  void restart();
  void next();
  bool valid();
  T current() const;
};
```

Alle grundlegenden Datencontainertypen (z.B. Array, Liste und Warteschlange) verfügen über spezialisierte Iteratoren, die direkt von ihnen instantiiert werden können. Zum Beispiel stellt die Array-Klasse einen Array-Iterator bereit:

```
template<class T> class RArray {
public:
    RArray(int size);
    T operator[](int index);
    RIterator<T>* newIterator() { return RArrayIterator<T>(this); };
};

template<class T> class RArrayIterator : public RIterator<T> {
public:
    RArrayIterator(RArray<T> source);
    void restart() { index = 0; }
    void next() { index++; }
    bool valid() const { return index < src.size();}
    T current() const { return src[index]; }
private:
    RArray<T> src;
    int index;
};
```

Ein Beispiel für eine Kopplung von Shape und Iterator bietet das Linien-Shape, das eine geordnete Menge von Ecken durch Linienzüge verbindet. Das Shape erhält die Eckeninformation

durch einen Eckkoordinaten-Iterator, Ecknormalen-Iterator, Eckfarben-Iterator und Ecktexturkoordinaten-Iterator. Ein Linien-Shape hat keine Kenntnis, woher seine Iteratoren die Daten beziehen. Wird ein Linien-Shape durch eine Engine ausgewertet, dann fordert der zuständige Painter oder Simplifier die konkreten Daten mit Hilfe der Iterator-Schnittstelle an.

```
class RLines : public RShape {
public:
  RLines (
    RIterator<RVector> vertexCoords,
    RIterator<RVector> vertexNormals,
    RIterator<RVector> vertexTexCoords,
    RIterator<RColor> vertexColors
  );
...
private:
  RIterator<RVector> vItr;
  RIterator<RVector> vnItr;
  RIterator<RVector> vtItr;
  RIterator<RColor> vcItr;
};
```

3.3.1.2 Abstrakte Iteratoren

Auf der Basis des Iterator-Konzepts zur Verbindung von Shapes mit Datenquellen können abstrakte Iterator-Klassen definiert werden, die es Anwendungen ermöglichen, Datenfolgen mit spezieller Charakteristik auszudrücken. Zu diesen abstrakten Iterator-Klassen gehören:

- C*onstant-Iterator*. Ein konstanter Iterator repräsentiert eine Folge eines konstanten Wertes. Dieser Iterator wird genutzt, um einen bestimmten Wert zu replizieren, und er hilft somit den Speicherplatzbedarf zu vermindern.
- *Replicate-Iterator*. Ein Replicate-Iterator wiederholt Werte, die aus einem anderen Iterator stammen, und bildet daraus eine neue Folge. Zum Beispiel kann aus der Ursprungsfolge ‚abcd' mit dem Wiederholungsfaktor 3 die Sequenz ‚aaabbbcccddd' generiert werden.
- *Repeat-Iterator*. Ein Repeat-Iterator wiederholt eine Sequenz, die durch einen anderen Iterator festgelegt ist, als ganzes. Zusätzlich kann eine Startposition und die Länge des zu wiederholenden Abschnitts festgelegt werden. Zum Beispiel generiert die Sequenz ‚abcde' mit der Startposition 1 (das erste Element besitzt den Index 0), der Länge 3 und dem Wiederholfaktor 4 die Folge ‚bcdbcdbcdbcd'.
- *Skip-Iterator*. Ein Skip-Iterator wiederholt eine Folge, die durch einen anderen Iterator spezifiziert ist, und lässt dabei Werte in bestimmten Intervallen aus. Zum Beispiel generiert die Sequenz ‚abcdefghij' mit einem Auslassungsintervall von 3 die Sequenz ‚abdeghj'.
- C*omposite-Iterator*. Ein Composite-Iterator schließt zwei oder mehr Iteratoren zu einem neuen Iterator zusammen.
- *Indexed-Iterator*. Ein Indexed-Iterator greift indiziert auf Elemente einer Sequenz zu und bildet daraus eine neue Folge. Ein solcher Iterator wird durch zwei Iteratoren definiert, einen Iterator, der die Indizes liefert, und einen weiteren Iterator, der die Datenfolge, auf die sich die Indizes beziehen, bereitstellt. Zum Beispiel generiert die Index-Folge ‚1-3-7-8' aus der Datenfolge ‚abcdefghijklmno' die Ergebnissequenz ‚acgh'.

Im folgenden Beispiel wird ein Polygon durch eine Sequenz von *N* Ecken spezifiziert, die alternierend gefärbt werden sollen. Für alle Ecken soll die identische Eckennormale bereitgestellt werden. Der Konstruktor eines Polygons erwartet bis zu vier Iteratoren, je einen für Eckkoordinaten, Normalen, Farben und Texturkoordinaten.

Der Iterator für die Eckkoordinaten ist ein Container-Iterator, der von der Array-Klasse bereitgestellt wird. Die Eckfarben werden durch einen Repeat-Iterator modelliert, der auf einen Array-Iterator zurückgreift und dessen Werte *N*/2+1 mal wiederholt. Das Array enthält die beiden abwechselnd anzuwendenden Farben. Ein Constant-Iterator repräsentiert die Ecknormalen.

```
RArray<RVector> vertices = new RArray<RVector>(N); // create vertex array
vertices[0] = RVector (x1, y1, z1); // assign coordinates
...
vertices[N-1] = RVector (xN, yN, zN);

RArray<RColor> colors = new RArray<RColor>(2); // create color array
colors[0] = blue;
colors[1] = white;

RPolygon p = new RPolygon(
    new RArrayIterator<RVector>(vertices),
    new RConstantIterator<RVector>(RVector(0,1,0), N),
    new RRepeatIterator<RColor>(
        new RArrayIterator<RColor>(colors), N/2+1
    )
);
```

Das Iterator-Konzept hat für Echtzeit-Renderingsysteme nur geringe Auswirkung auf das Leistungsverhalten im Vergleich zu einer direkten Speicherung der Daten in den Shapes. In beiden Varianten müssen die Daten an das Renderingsystem, z.B. OpenGL, mit Hilfe von Funktionsaufrufen übermittelt werden. Ein zeitkritischer Shape-Painter kann außerdem zur Laufzeit den Typ des Iterators erfragen. Stellt der Shape-Painter fest, dass es sich um einen Array-Iterator handelt, kann dieser direkt auf die Array-Repräsentation zurückgreifen und dadurch die Iterator-Traversierung vermeiden. Jedoch sind die damit erzielbaren Einsparung insgesamt gering; der Flaschenhals liegt in der Übermittlung der Daten an die Graphik-Hardware. So werden z.B. im Fall von OpenGL-Primitiven die Eckdaten pro Ecke mittels einzelner Funktionsaufrufe übertragen. Eine Ausnahme hierzu bilden die in OpenGL 1.1 eingeführten Vertex-Arrays: Hier kann ein polygonales Primitiv mit Hilfe von Indizes durch eine Eckentabelle spezifiziert werden; die Zahl der OpenGL-Funktionsaufrufe wird dadurch vermindert, da die Indizes als ganzes übermittelt werden. Diese Variante ist implementierbar, falls durch eine Laufzeitabfrage des Iterator-Typs ermittelt wird, dass es sich bei der Datenquelle um ein Array handelt.

> Shapes werden über Iteratoren, die eine Schnittstelle für den sequentiellen Datenzugriff definieren, an Datenquellen angeschlossen. Iteratoren ermöglichen die Trennung der Datenhaltung von den Shapes und damit die Integration anwendungseigener Datenstrukturen in Shapes. Auf der Grundlage des Iterator-Konzepts können prozedural spezifizierte Datenfolgen modelliert werden. Datenfolgen mit speziellen Charakteristiken lassen sich mit abstrakten Iteratoren ausdrücken.

3.3.2 Implementierung von Shape-Typen

Im Verlauf der langfristigen Entwicklung eines computergraphischen Anwendungsprogramms entsteht häufig der Wunsch, neue Shape-Typen zu integrieren, um damit eine bessere Rendering-Geschwindigkeit zu erzielen oder neue Shape-Modellierungstechniken zur Verfügung zu stellen. Anwendungsprogramme können neue Shape-Typen gleichwertig zu den im generischen Renderingsystem eingebauten Shape-Typen integrieren.

Ein *Shape-Typ* besteht konzeptionell aus einer Shape-Klasse, die alle zur Definition notwendigen Daten eines Shapes festlegt, und einer oder mehreren Handler-Klassen, die Services für Shapes des neuen Typs bereitstellen (siehe Abbildung 19). Im einfachsten Fall wird zu einem neuen Shape-Typ ein Shape-Simplifier bereitgestellt; alle anderen Services der Handler-

Tabelle können durch Shape-Simplifikation und rekursive Auswertung auf Services bereits eingebauter Shape-Typen delegiert werden. Jeder Shape-Simplifier registriert sich dazu in der Basisklasse aller Engines. Durch diese Registrierung wird der Simplifier automatisch in die Handler-Tabelle jeder Engine eingetragen.

Weiter können Shape-Painter für konkrete Renderingsysteme bereitgestellt werden. Ist ein Painter definiert, dann kann ein Shape bei der Evaluation durch eine Engine direkt auf ein konkretes Renderingsystem abgebildet werden. Jeder Shape-Painter registriert sich in der zugehörigen Engine und wird dadurch automatisch in deren Handler-Tabelle eingetragen.

Abbildung 20 zeigt das Klassenmodell des Shape-Typs *Volumen*. Ein Volumen *RVolume* repräsentiert einen 3D-Array von Datenwerten. Objekte der Klasse *RVolumeSimplifier* extrahieren aus dem Datenvolumen eine Menge transparenter, texturierter Polygone, die entsprechend der Kameraeinstellung „ineinander gesteckt“ werden, um den Eindruck einer Volumenvisualisierung zu erzeugen. Objekte der Klasse *RVolumePainterGL* hingegen rendern ein Datenvolumen direkt mit Hilfe von 3D-Texturen in OpenGL.

Im generischen Renderingsystem wird ein Shape-Typ durch eine Shape-Klasse, eine Shape-Simplifier-Klasse und spezielle Shape-Painter-Klassen definiert. Shape-Painter können für jedes konkrete Renderingsystem bereitgestellt werden. Der Shape-Simplifier bewirkt, dass automatisch alle Services auch für den neuen Shape-Typ verfügbar sind. Das Konzept der Shape-Typen stellt somit einen geordneten Weg zur Integration anwendungsspezifischer Shapes und technisch die gleichen Mittel wie bei eingebauten Shape-Typen bereit.

3.3.3 Shape-Klassenhierarchie

Das generische Renderingsystem definiert eine Sammlung von Shape-Klassen sowie die zugehörigen Shape-Simplifier und Shape-Painter. Diese Shape-Typen lassen sich für alle adaptierten, konkreten Renderingsysteme einsetzen. Der Entwurf richtet sich nach Ähnlichkeiten in den Objekteigenschaften und unterscheidet folgende Kategorien:

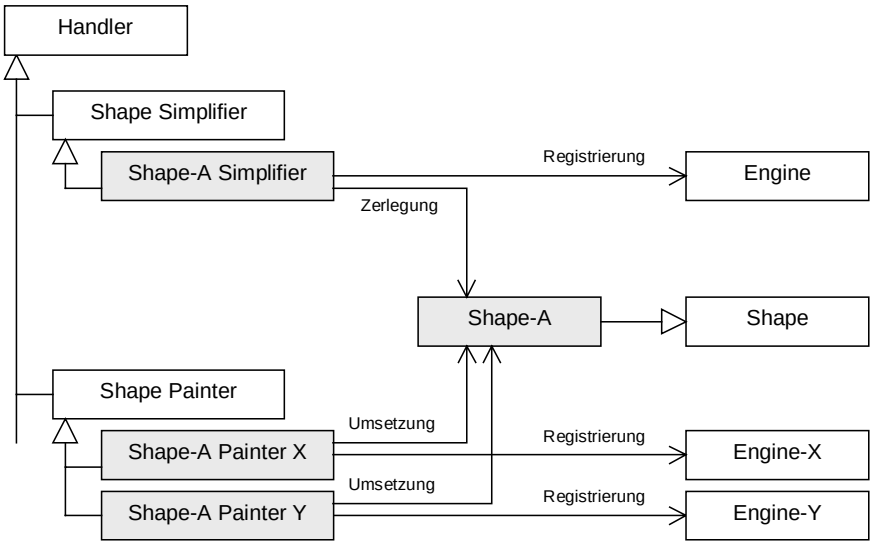

Abbildung 19. Klassenmodell eines Shape-Typs.

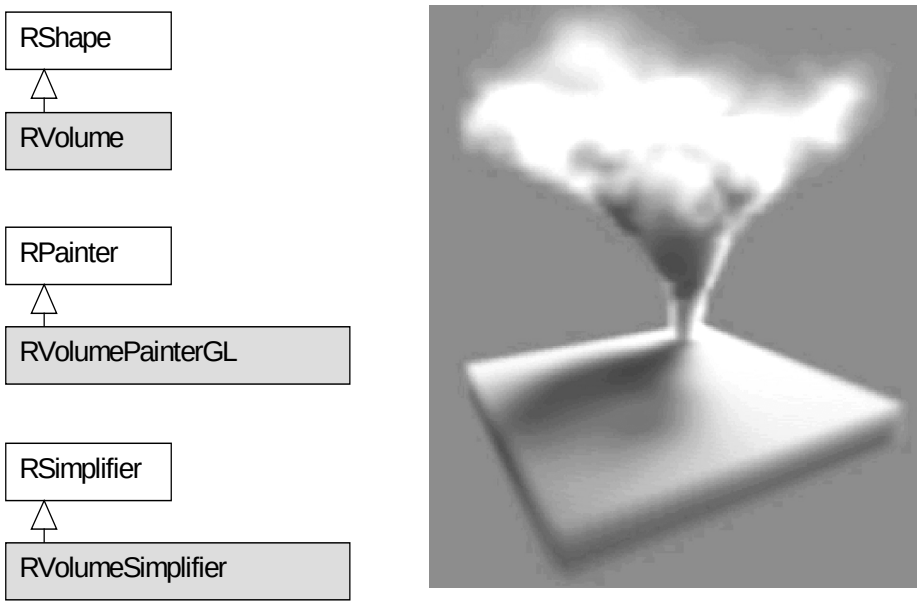

Abbildung 20. Klassenmodell des Shape-Typs *Volume* (links). Beispiel eines gerenderten Volumens (links).

- *Polygonale Shapes.* Polygonale Shapes umfassen alle Shapes, deren Geometrie durch Eckeninformation beschrieben werden kann. Grundsätzlich lassen sich Iteratoren für Eckkoordinaten, Ecknormalen, Eckfarben und Ecktexturkoordinaten festlegen. Konkrete polygonale Shape-Typen sind Punkte, Linien, konvexe Polygone (PolygonSet), allgemeine Polygone (Facet), Polygonnetz (Mesh) und Extrusionskörper (Extrusion). Die genannten polygonalen Shapes sind mit 2D-Geometriedaten und 3D-Geometriedaten einsetzbar. Für OpenGL existieren Shape-Painter, die diese Shapes optimal umsetzen und direkt an die Hardware weiterleiten.
- *Analytische Shapes*. Analytische Shapes umfassen alle Shapes, deren Oberfläche implizit beschrieben werden kann. Die Shapes werden durch Parameterwerte definiert. Shapes dieser Kategorie umfassen Box, Kugel, Torus, Zylinder, Kegel, Paraboloid, Hyberboloid, Ebene, Scheibe und Superquadrik.
- *Kurven.* Shapes dieser Kategorie werden durch Kontrollpunkte, Gewichte und weitere Parameter (z.B. Grad) definiert. Vertreter dieser Kategorie sind z.B. Beziér-Kurven und B-Spline-Kurven.
- *Freiformflächen*. Shapes dieser Kategorie werden durch ein zweidimensionales Array von Kontrollpunkten und Gewichten repräsentiert. Zu ihnen gehören quadratische, bilineare Patches und Non-Uniform Rational B-Splines (NURBS).
- *Text-Shapes* Shapes dieser Kategorie repräsentieren einzelne Glyphen, z.B. Buchstaben, eines Fonts. Die Glyphen können als Texturen (2D) oder als geometrische Modelle (3D) generiert werden.

Diese Shape-Klassenhierarchie erhebt nicht den Anspruch auf Vollständigkeit; sie deckt nahezu alle Shape-Typen ab, die in konkreten Renderingsystemen zu finden sind. In einigen Erweiterungen des Virtuellen Renderingsystems werden spezielle Shape-Klassen angeboten, so zum Beispiel durch die adaptierte Tubing & Extrusion-Bibliothek [107] oder durch einen Solid-Modellierer auf der Basis von Halbkantenstrukturen [59].

3.4 Entwurf der Attribut-Klassen

Die Analyse computergraphischer Systeme hat gezeigt, dass sich Renderingsysteme insbesondere hinsichtlich der graphischen Attribute unterscheiden, da diese meist eng mit dem zugrundeliegenden Renderingverfahren verknüpft sind. Im generischen Renderingsystem muss deshalb eine möglichst allgemeine Definition und ein technisch leistungsfähiges Implementierungskonzept gefunden werden, um diesem Umstand gerecht zu werden.

Attribute im generischen Renderingsystem definieren konkret sowohl die Erscheinung und Transformation von Szenenobjekten als auch Parameter, die sich auf die Arbeitsweise von Auswertungsverfahren auswirken; Attribute sind Modifikatoren. Trotz dieser allgemeinen Definition ist es nicht möglich, die Attribute verschiedener Renderingsysteme durch eine Sammlung allgemeiner Attributklassen auszudrücken – zu groß sind die Unterschiede in den einzelnen Renderingsystemen. Stattdessen wird eine relative geringe Zahl von Kernattributen definiert, die durch spezifische Attribute für die jeweiligen Renderingsysteme ergänzt werden. Der Ansatz zur Modellierung von Attributklassen lässt sich dabei wie folgt charakterisieren:

- *Objektbasierte Attributdarstellung*. Die Darstellung von Attributen erfolgt durch Objekte, so dass Attribute über eine explizite Schnittstelle und einen gekapselten Zustand verfügen. Diese Herangehensweise bietet insbesondere aus Nutzersicht den Vorteil, dass die Handhabung zusammengehörender Parameter konsistent und sicher erfolgt – verglichen mit einem Ansatz, in dem Attribute als Parametersammlungen dargestellt sind. So werden zum Beispiel im Texturattribut alle wesentlichen Texturparameter (z.B. Filterung, Randbehandlung, Blendmodus) gebündelt und konsistent verwaltet.
- *Shape-unabhängige Attributdarstellung*. Attribute sind unabhängig von der Shape-Klassenhierarchie modelliert. Eine Engine verwaltet Attribute mit Hilfe des generischen Kontexts. Attribute sind nicht mit Shapes assoziiert; die Assoziation legen die Handler fest, die frei in ihrer Entscheidung sind, welche der Attribute sie in ihrem Renderingalgorithmus berücksichtigen. Dadurch können im generischen Renderingsystem neue Attribute definiert werden, ohne die Shape-Klassenhierarchie zu tangieren.
- *Bereitstellung von Kernattributen.* Das generische Renderingsystem stellt eine Sammlung von Kernattributen bereit, die so gewählt wurden, dass sie für alle derzeit adaptierten Renderingsysteme sinnvoll umgesetzt werden können. Zu den Kernattributen zählen insbesondere geometrische Transformationen und Lichtquellen. Der Grad, zu dem diese Attribute ausgewertet werden können, hängt von den Leistungsmerkmalen des jeweiligen Zielrenderingsystems ab. Dabei lassen sich hauptsächlich zwei Gruppen von Renderingsystemen unterscheiden – diejenigen mit einem lokalen und diejenigen mit einem globalen Beleuchtungsmodell – die jeweils stark ähnliche Leistungsmerkmale besitzen. Die Bereitstellung von Kernattributen verfolgt das Ziel, es Anwendungen zu ermöglichen, das konkrete Renderingsystem auszutauschen und dabei ohne zusätzlichen Kodierungsaufwand brauchbare, wenngleich selten optimale Ausgaben mit unterschiedlichen Renderingsystemen zu erzielen.
- *Einheitliche Attributverwaltung.* Die einheitliche Verwaltung aller Attribute mit Hilfe des generischen Kontexts vereinfacht die Handhabung der Attribute, insbesondere aber die der renderingsystemspezifischen und anwendungsspezifischen Attribute.
- *Spezialisierbarkeit der Attribute.* Für jedes adaptierte Renderingsystem werden die nur dort vorhandenen Attribute durch Attribute des generischen Renderingsystems modelliert, um sicherzustellen, dass die spezifischen Fähigkeiten und Gestaltungsmöglichkeiten eines Renderingsystems in generischen Renderingsystem zugreifbar und nutzbar bleiben. Für das photorealistische Renderingsystem RenderMan werden zum Beispiel die dort vorhandenen

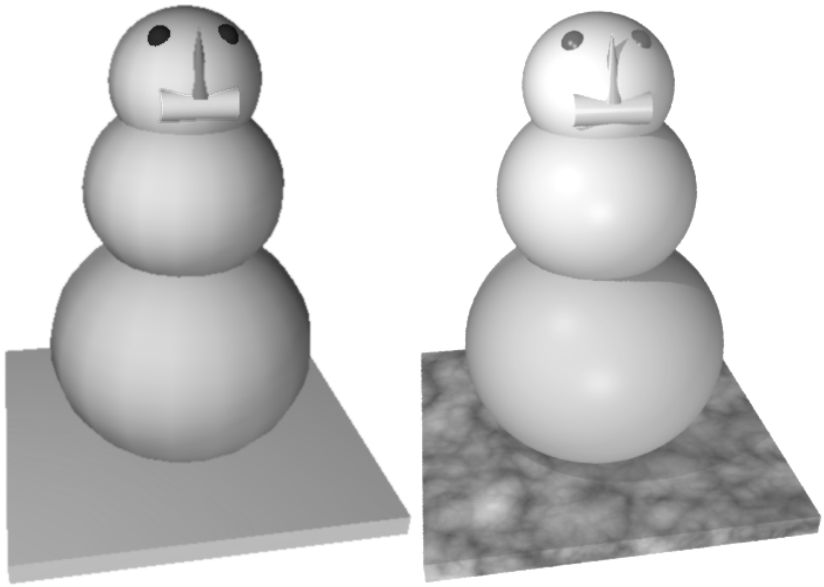

Abbildung 21. OpenGL-Variante einer Szene (links) und RenderMan-Variante (rechts).

Shader als spezielle Attribute zur Verfügung gestellt. Nur RenderMan-spezifische Handler werten diese Attribute aus; alle anderen Handler werden dadurch nicht beeinflusst. Eine Festschreibung oder gar Vereinheitlichung der Attribute würde allen adaptierten Renderingsystemen das Wesentliche nehmen, ihre spezifischen graphischen Fähigkeiten, die sie auszeichnen.

Abbildung 21 zeigt eine Szene, die sowohl mit einer OpenGL-Engine als auch mit einer RenderMan-Engine ausgewertet wurde. Die Szene nutzt spezielle RenderMan-Shader: Im Fall der Bodenfläche wird ein Marmor-Shader verwendet, im Fall des Schneemanns ein Plastik-Shader. Die Farbe der Bodenfläche im OpenGL-generierten Bild leitet sich von den Material-Koeffizienten im Marmor-Shader ab.

3.4.1 Monoattribute und Polyattribute

Das generische Renderingsystem unterteilt Attribute in Mono- und Polyattribute. Die Unterteilung erfolgt danach, ob sich nur ein Attribut-Objekt oder mehrere Attribut-Objekte einer bestimmten Attributkategorie gleichzeitig auf das Rendering auswirken können. Ein Attribut wird als *aktiv* bezeichnet, wenn es im generischen Kontext einem Mikroprogramm zur Verfügung steht.

- *Monoattribute.* Ein Monoattributkategorie ist eine Attributkategorie, in der nur jeweils ein Attribut-Objekt zu einem Zeitpunkt im generischen Kontext aktiv sein kann. Eine Monoattributkategorie wird durch einen Stack im generischen Kontext repräsentiert. Das oberste Element des Stacks repräsentiert das aktive Attribut-Objekt der jeweiligen Attributkategorie.
- *Polyattribute.* Ein Polyattributkategorie ist eine Attributkategorie, in der mehr als ein Attribut-Objekt zu einem Zeitpunkt im generischen Kontext aktiv sein kann; alle aktiven Attribut-Objekte können grundsätzlich während der Auswertung betrachtet werden und sich gemeinsam auf die Auswertung von Renderingkomponenten auswirken. Eine Polyattributkategorie wird durch eine Liste im generischen Kontext repräsentiert; jedes Attribut-Objekt einer solchen kann über den Listenindex referenziert werden.

Zur Verdeutlichung betrachten wir jeweils einen typischen Vertreter dieser beiden Attributkategorien. Das Farbattribut wird im generischen Kontext als Monoattribut dargestellt: Bei der Auswertung eines Shapes wird das jeweils aktive Farbattribut betrachtet, das die Oberflächen-

farbe des Shapes festlegt. Lichtquellenattribute hingegen sind als Polyattribute dargestellt. Beim Rendering einer Szenen sind im Allgemeinen mehrere Lichtquellen gleichzeitig aktiv.

Die Stack-Datenstruktur für Monoattribute und Listen-Datenstruktur für Polyattribute im generischen Kontext geben die zur Verfügung gestellten Methoden zur Manipulation der Attribute vor. Für Stacks stehen eine *push*-, *pop*- und *top*-Methode bereit; für Listen stehen eine *add*-, *remove*- und *get*-Methode zur Verfügung. Die *Aktivierung* eines Attribut-Objekts erfolgt durch die *push*- bzw. *add*-Methode; die *Deaktivierung* erfolgt mittels der *pop*- bzw. *remove*-Methode.

3.4.2 Auswertung von Attributen

Attribute können von einer Auswertungsstrategie auf verschiedene Arten ausgewertet werden.

- *Direkte Auswertung von Attributen bei ihrer Aktivierung.* In dieser Variante existiert ein Attribut-Painter, der aufgerufen wird, wenn ein Attribut-Objekt auf den generischen Kontext gelegt wird bzw. von dort entfernt wird. Der Painter hat im Allgemeinen die Aufgabe, den Zustand im graphischen Kontext des zugrundeliegenden Renderingsystems entsprechend zu aktualisieren (z.B. im Fall von OpenGL).
- *Indirekte Auswertung von Attributen durch Shape-Handler.* In der zweiten Variante bewirkt das Eintragen bzw. Entfernen von Attribut-Objekten in den bzw. von dem generischen Kontext keine Veränderung im Zustand des zugrundeliegenden Renderingsystems – allein die Shape-Handler, z.B. die Shape-Painter, sind verantwortlich für die Abfrage und Interpretation der aktiven Attribut-Objekte des generischen Kontexts.
- *Auswertung von Attributen durch Simplifikation.* Komplexe Attribute, die durch Aggregation weniger komplexer Attribute dargestellt werden können, lassen sich mit Hilfe von Attribut-Simplifiern auswerten. Ein Attribut-Simplifier zerlegt ein komplexes Attribut-Objekt in eine Menge von weniger komplexen Attribut-Objekten und wertet diese Attribut-Objekte rekursiv aus. Beispielsweise könnte ein Attribut „Hervorhebung“ in die Attribute „Linienstil“ und „Farbe“ zerlegt werden. Attribut-Simplifier gelangen auch dann zum Einsatz, wenn ein renderingsystemspezifisches Attribut sinnvoll, aber nur approximativ auf ein anderes Renderingsystem abgebildet werden kann. Zum Beispiel können im Fall eines RenderMan-Shader-Attributes, das für OpenGL umgesetzt werden soll, die Materialkoeffizienten für ambientes, diffuses und spekulares Licht ermittelt werden, um dann als OpenGL-Material-Attribut ausgewertet zu werden.

Die erste Variante ist überwiegend dort im Einsatz, wo Attribute direkt dem Attributkontext des konkreten Renderingsystems zugeordnet werden können. Die zweite Variante wird hingegen dann eingesetzt, wenn die Attribute eher die Arbeitsweise eines Renderingalgorithmus beeinflussen, also nicht direkt auf Zielkonstrukte des konkreten Renderingsystems abgebildet werden können. Die dritte Variante eignet sich für anwendungsspezifische, im Allgemeinen abstrakte Attribute, die sich auf bereits vorhandene abstützen.

3.4.3 Attribut-Klassenhierarchie

Die im generischen Renderingsystem enthaltenen Kernattribute sind im folgenden kurz skizziert. Es sei angemerkt, dass diese Sammlung weder vollständig noch optimal ist; sie kann als Anhaltspunkt für eine weitere Diskussion und Entwicklung dienen. Durch die Offenheit der Attributmodellierung, insbesondere durch die einfache Integration neuer Attribute in das generische Renderingsystem, ergeben sich beim Einsatz dieses Systems keine Beschränkungen.

- *Oberflächen-Attribute.* Diese Attribute umfassen Farbe, Material, Textur und Tessellation. Sie legen einheitlich die graphischen Grundvariablen der Geometrie-Darstellung fest, können jedoch von renderingsystemspezifischen Attributen übersteuert werden. Diese Attribute sind als Monoattribute modelliert.
- *Punkt- und Linienattribute.* Diese Attribute umfassen Zeichenstile und Zeichenparameter für Punktprimitive und Linienprimitive. Diese Attribute sind ebenfalls als Monoattribute modelliert.
- *Lichtquellen-Attribute.* Diese Attribute umfassen gerichtete Lichtquellen, punktförmige Lichtquellen, Spot-Lichtquellen und ambiente Lichtquellen. Sie sind als Polyattribute modelliert.
- *Geometrische Transformationen.* Diese Attribute enthalten 4x4-Transformationsmatrizen. Konkrete geometrische Transformationen umfassen Skalierung, Translation, Rotation, perspektivische und orthogonale Projektion, sowie Kamera-Orientierungstransformationen. Transformationen beziehen sich im Allgemeinen auf die Model-View-Matrix des generischen Kontexts. Transformationen können auch explizit auf Textur-, Farbraum- und Projektionsmatrizen des generischen Kontexts angewendet werden. Geometrische Transformationen sind als Monoattribute modelliert.
- *Filter-Attribute.* Diese Attribute repräsentieren Identifier, die insbesondere Shapes beigeordnet werden können. Während des Renderings können diese Identifier zur Auswahl bestimmter Szenenobjekte oder zur Identifikation von Szenenobjekten eingesetzt werden. Diese Attribute sind als Polyattribute modelliert.

Das Virtuelle Renderingsystem stellt ferner für jedes konkrete Renderingsystem eine Sammlung von spezifischen Attributen, die die Eigenheiten des jeweiligen Systems ansteuern, bereit. Zur Integration eines anwendungsspezifischen Attributs muss eine Attributklasse modelliert werden. Je nach Art der Auswertung ist zudem noch ein Attribut-Painter oder Attribut-Simplifier erforderlich.

> Das generische Renderingsystem verwaltet Attribute über den generischen Kontext. Die Verwaltung unterscheidet zwischen Monoattributen und Polyattributen. Die Attribut-Auswertung erfolgt über Handler-Objekte. Abgesehen von Kernattributen, die auf allen konkreten Renderingsystemen anwendbar sind, werden für konkrete Renderingsysteme ihren graphischen Leistungsmerkmalen entsprechende Attribute modelliert. Dieser Ansatz stellt sicher, dass die Uneinheitlichkeit in den Attributen verschiedener Systeme im generischen Renderingsystem modelliert werden kann.

3.5 Integration von Auswertungsstrategien

Die explizite, objektorientierte Modellierung von Auswertungsalgorithmen ist ein wesentliches Kennzeichen des generischen Renderingsystems und zugleich das Schlüsselkonzept, mit dem über die reine Bildsynthese hinausgehende Auswertungsstrategien von graphisch-geometrischen Sachverhalten in das generische Renderingsystem integriert werden können. Diese Loslösung von einer festgelegten Rendering-Pipeline durch den Einsatz von austauschbaren Technik-Objekten, die die Auswertungsstrategie für Renderingobjekte definieren, erlaubt es, beliebige, nicht notwendig visuelle Verfahren auf Szenenbeschreibungen anzuwenden.

Renderingverfahren, insbesondere im Bereich des Echtzeit-Renderings, erfordern es zuweilen, dass die Beschreibung der Szenenobjekte mehrfach ausgewertet wird; wir sprechen hier

von *Multi-Pass-Rendering*. In jedem der Durchläufe werden die Szenenobjekte unterschiedlich, z.B. für unterschiedliche Framebuffer-Komponenten, ausgewertet. Techniken müssen hierzu insbesondere die Kontrolle über die Traversierung der Szenenbeschreibung übernehmen können.

3.5.1 Technik-Objekte

Eine *Technik* repräsentiert im generischen Renderingsystem die Implementierung einer Auswertungsstrategie für Renderingkomponenten. Insbesondere werden darunter Verfahren, die eine Szenenbeschreibung auswerten, verstanden. Die Form der Ergebnisse einer solchen Auswertung ist dabei nicht festgelegt.

Ein *Technik-Objekt*, das ein spezialisiertes Monoattribut-Objekt repräsentiert, implementiert eine Strategie zur Auswertung einer Folge von Engine-Steuerkommandos. Ein Technik-Objekt legt fest, wieviele Durchläufe durch eine Szenenbeschreibung, die in Form von Engine-Steuerkommandos kodiert ist, benötigt werden. Die Schnittstelle der Technik-Objekte ist wie folgt definiert:

```
class RTechnique : public RMonoAttribute {
public:
  void start(REngine e);
  void stop(REngine e);
  void preparePass(REngine e);
  void finishPass(REngine e);
  void nextPass(REngine e)
  bool needsPass(REngine e);
  ...
};
```

Ein Technik-Objekt ist für die Auswertung von Attribut- und Shape-Objekten zuständig. Die Methoden zur Auswertung dieser Objekte, die die Engine bereitstellt, delegieren ihre Arbeit an das in der Engine aktive Technik-Objekt. Wird zum Beispiel ein Attribut-Objekt auf den generischen Kontext gelegt, dann wird dieser Aufruf an das aktive Technik-Objekt weitergeleitet, das entscheidet, ob das Attribut ignoriert, auf den generischen Kontext abgelegt oder anderweitig ausgewertet wird. Ein Technik-Objekt zur Bildsynthese wird in einem solchen Fall versuchen, einen Painter oder Simplifier für das Attribut-Objekt zu finden und deren Auswertungsfunktionalität auszuführen. Analog erfolgt die Delegation im Fall eines Shape-Objektes.

Innerhalb einer hierarchischen Szenenbeschreibung muss dafür Sorge getragen werden, dass ein in einem Knoten befindliches Technik-Objekt *lokal* die Traversierungkontrolle erhält, insbesondere dann, wenn die Technik mehrere Durchläufe benötigt. Die Kontrolle über die Traversierung übernimmt das Technik-Objekt; dazu definiert die Schnittstelle der Technik-Objekte insbesondere die Methode *needsPass*, die mitteilt, ob ein weiterer Renderingdurchlauf notwendig ist. Eine Traversierung eines Szenengraphen kann unter Anwendung eines Technik-Objektes wie folgt kodiert werden (die Klasse *RootNode* sei die Knotenklasse, deren Instanzen den obersten Knoten eines Szenengraphen bilden und durch die die Traversierung initiiert wird):

```
void RootNode::doTraversal(REngine e) {
  RTechnique t = e.getActiveTechnique();
  t.start(e);
  while(t.needsPass(e)) {
    t.preparePass(e);
    this.traverse(e);
    t.finishPass(e);
  }
  t.stop(e);
}
```

Die Implementierung einer Technik-Klasse umfasst auch die Methoden, die von einer Engine zur Manipulation des generischen Kontexts und zur Auswertung von Shapes definiert werden. Dadurch wird es einer einzelnen Technik möglich, die Manipulation des generischen Kontexts zu kontrollieren. Die Schnittstelle umfasst folgende weitere Methoden:

```
class RTechnique : public RMonoAttribute {
public:
  ...
  void forwardEval(REngine e, RShape shape);

  void forwardPush(REngine e, RMonoAttribute attr);
  void forwardPop(REngine e, RMonoAttribute attr);
  void forwardAdd(REngine e, RPolyAttribute attr);
  void forwardRemove(REngine e, RPolyAttribute attr);
  void forwardInstall(REngine e, RHandler hdl);
  void forwardDeinstall(REngine e, RHandler hdl);
};
```

Die Methoden zur Auswertung von Renderingobjekten wurden in die Schnittstelle der Technik-Objekte aufgenommen, um dort eine Möglichkeit zu schaffen, effizient mit Attributen und Shapes umzugehen. Konkret kann dies bedeuten, dass zum Beispiel eine Technik, die ausschließlich analytisch-geometrische Berechnungen durchführt, alle nicht relevanten Attribute (z.B. graphische Attribute) ignoriert, d.h. für sie keine Handler aufruft. Ein weiteres Beispiel wäre eine Technik, die Schnittpunkte eines Strahls mit einem bestimmten Szenenobjekt berechnet. Wird diesem Szenenobjekt ein Filter-Attribut beigeordnet, dann kann die Intersektionstechnik alle *forwardEval*-Aufrufe für solche Shape-Objekte ignorieren, für die dieses Attribut nicht vorhanden ist.

3.5.2 Bildsynthese-Techniken

Eine *Bildsynthese-Technik* rendert alle evaluierten Shape-Objekte unter Berücksichtigung der aktiven Attribut-Objekte mit dem Ziel, ein Bild zur gegebenen Szenenbeschreibung zu erzeugen.

Im Allgemeinen implementiert die *forwardEval*-Methode die rekursive Auswertungsstrategie, die für Shapes beschrieben wurde, d.h. es wird zunächst versucht, einen Painter oder einen Simplifier zum gegebenen Shape-Objekt zu finden. Schlägt die Suche mit der Shape-Klasse als Target fehl, wird in der Handler-Tabelle nach Handlern für die nächsthöhere Basisklasse des Shapes rekursiv weitergesucht. Die Methoden zur Verwaltung der Monoattribute und Polyattribute suchen analog nach geeigneten Handlern in der Handler-Tabelle.

Eine Bildsynthese-Technik benötigt im Fall einer direkten Abbildung in eine Szenenbeschreibung im Format des Zielrenderingsystems einen einzigen Durchlauf. Im Fall von OpenGL sind mindestens zwei Durchläufe erforderlich: Im ersten Durchlauf werden nur Shape-Objekte, die nicht transparent sind, gezeichnet. Im zweiten Durchlauf werden ausschließlich Shape-Objekte gezeichnet, die transparent sind, wobei das Schreiben im Tiefenbuffer blockiert wird. In Varianten des Szenenrenderings, in denen Antialiasing durch Multi-Pass-Rendering erzielt wird, sind entsprechend mehr Durchläufe notwendig.

3.5.3 Bildanalyse-Techniken

Eine *Bildanalyse-Technik* berechnet oder extrahiert Informationen über Szenen oder einzelne Szenenobjekte, ohne dabei ein Bild zu synthetisieren. Im Allgemeinen sind Bildanalyse-Techniken nicht direkt an ein konkretes Renderingsystem gebunden und benötigen keine

spezielle computergraphische Hardware. Beispiele für Analyseverfahren, die auf Szenenbeschreibungen angewendet werden können, sind:

- *Strahl-Objekt-Intersektionsberechnung:* Ein solches Technik-Objekt evaluiert Shape-Objekte, indem geprüft wird, ob ihre Geometrie von einem Strahl geschnitten wird. DieSchnittpunkte werden im Technik-Objekt abgelegt und können von dort der Anwendung zur Verfügung gestellt werden. Diese Bildanalysetechnik liefert objektpräzise Ergebnisse und kann zum Beispiel zur Messung von Abständen, zur Unterstützung von Benutzer-Szenen-Interaktionen oder zur Ermittlung von Verdeckung eingesetzt werden. Das Technik-Objekt implementiert die *forwardEval*-Methode dergestalt, dass Intersektions-Handler für Shape-Objekte gesucht werden; falls keine Handler gefunden werden, wird durch Simplifikation der Shape-Objekte versucht, geeignete Handler zu finden.
- *Attributanalyse:* Ein solches Technik-Objekt verzeichnet für ein gegebenes Shape-Objekt alle für die Shape-Auswertung relevanten Attribute. Mit dieser Technik lassen sich alle tatsächlich verwendeten Attribute, die auf ein Shape bei der Bildsynthese einwirken, ermitteln. Die ermittelten Attribut-Objekte werden im Technik-Objekt abgelegt und stehen dort der Anwendung zur Verfügung.
- *Bildbereichsanalyse:* Ein solches Technik-Objekt berechnet für ein oder mehrere gegebene Shapes deren Lage (nach der Projektion) im Bildraum. Berechnet werden die Bounding-Box in Bildraumkoordinaten. Weiter wird die für das jeweilige Shape-Objekt verwendete Projektions- und Model-View-Matrix verzeichnet.

3.5.4 Verwendung von Technik-Objekten als Attribute

Technik-Objekte besitzen eine komplexe Implementierung, die sowohl für das Multi-Pass-Rendering als auch für die Delegation von Engine-Steuerkommandos verantwortlich zeichnet. Aus der Sicht einer Anwendung, d.h. aus der Sicht der Szenenbeschreibung, unterscheiden sich Technik-Objekte nicht von anderen Attribut-Objekten. Bei der Auswertung eines Mikroprogramms des generischen Renderingsystems muss ausschließlich dafür Sorge getragen werden, dass die aktiven Technik-Objekte den Kontrollfluss übernehmen. Im nächsten Abschnitt, der einen Entwurf einer generischen Szenenrepräsentation vorstellt, wird auf die speziellen Aspekte der Verwendung von Technik-Objekten innerhalb einer hierarchischen Szenenbeschreibung näher eingegangen.

Im generischen Renderingsystem werden Auswertungsstrategien durch Technik-Objekte gekapselt. Sie definieren eine Schnittstelle für Multi-Pass-Rendering und für die (über Engines delegierte) Auswertung von Shapes und Attributen. Technik-Objekte kontrollieren somit die Wahl der Auswertungsalgorithmen als auch den Kontrollfluss der Auswertung. Insbesondere lassen sich beliebige Bildsynthese- und Bildanalyse-Techniken mit dem hier vorgestellten Ansatz vollständig in das Gesamtkonzept des generischen Renderingsystems integrieren. Damit wird eine Auswertungsmethodik für graphisch-geometrische Sachverhalte geschaffen, die keine Annahmen über die Natur des Renderingverfahrens aufstellt. Der Begriff „Renderingverfahren“ bezeichnet deswegen im generischen Renderingsystem allgemein die Interpretation graphisch-geometrischer Sachverhalte, die durch Renderingkomponenten in der Szenenrepräsentation spezifiziert sind.

4 ENTWURF EINES GENERISCHEN SZENENGRAPHEN

Der Begriff „Szene“ wird im folgenden als Bezeichnung für einen graphisch-geometrischen Sachverhalt verwendet. Ein solcher Sachverhalt besteht aus Szenenobjekten, die durch eine hierarchische Datenstruktur, der Szenenrepräsentation, beschrieben werden. In der Computergraphik wird unter einem *Szenengraph* ein gerichteter, azyklischer Graph verstanden, dessen Knoten Bausteine der Szenenspezifikation enthalten. Eine ideale Implementierung einer Szenengraph-Datenstruktur sollte die Erstellung computergraphischer Anwendungsprogramme erleichtern und das Rendering einer Szene durch Analyse des Szenengraphen optimieren [99]. Die *Szenenmodellierung* bezeichnet den Vorgang der Konstruktion einer Szenenbeschreibung.

Szenengraphen sind in den meisten Graphiksystemen und computergraphischen Bibliotheken als Modellierungsstruktur allgegenwärtig. Der Szenengraph fand eine weite Verbreitung insbesondere durch die computergraphische Bibliothek OpenInventor [101], deren Szenengraphkonzept in abgewandelter Form in die Virtual Reality Modeling Language (VRML) übernommen wurde. Die Gründe für die Popularität des Szenengraphen sind vielfältig: Der Szenengraph ermöglicht die kompakte Spezifikation von Szenen, erlaubt die mehrfache Instanziierung von Szenenobjekten durch mehrfache Referenzierung von Subgraphen und schließlich eignet sie die Graphenstruktur unmittelbar zur Editierung durch eine graphische Benutzerschnittstelle.

In diesem Kapitel wird in Form des generischen Szenengraphen eine Fortentwicklung dieses Konzeptes vorgestellt. Besondere Kennzeichen dieses Szenengraphen sind die Unterstützung verschiedener Renderingsysteme, die Integration von Multi-Pass-Renderingverfahren und die Erhöhung der Abstraktion in der Szenenspezifikation durch deklarative Elemente.

4.1 Motivation für einen generischen Szenengraphen

Unterstützung der Szenenmodellierung für das generische Renderingsystem

Für die im generischen Renderingsystem definierten Renderingkomponenten soll dieser Ansatz abstraktere Modellierungskomponenten bereitstellen, die die Komplexität der Programmierung des generischen Renderingsystems deutlich verringern. Der generische Szenengraph ist deshalb eng an das generische Renderingsystem geknüpft, jedoch grundsätzlich auf andere Rendering-

systeme übertragbar. Das Klassenmodell, auf dem sich der generische Szenengraph gründet, unterscheidet drei Hauptkategorien von Objekten:

- *Renderingobjekte*: Zu ihnen zählen die im Kapitel 3 vorgestellten Shapes, Attribute, Handler und Techniken; diese Objekte repräsentieren die geometrischen, graphischen und algorithmischen Elemente, die zur Konstruktion einer Szene eingesetzt werden.
- *Szenengraphknoten:* Sie organisieren hierarchisch Renderingobjekte in einem gerichteten, azyklischen Graphen. Sie können gleichermaßen Renderingobjekte enthalten, generieren, modifizieren oder beschränken.
- *Engines:* Sie traversieren einen Szenengraphen und interpretieren dessen Inhalt, die Renderingobjekte. Während der Szenengraphauswertung werden mit Hilfe des generischen Kontexts Attribute mit Shapes assoziiert sowie installierte Handler dazu genutzt, Renderingobjekte in Konstrukte oder Befehle eines zugrundeliegenden Renderingsystems abzubilden.

Unterstützung unterschiedlicher Renderingsysteme

Der hier vorgestellte Ansatz eines generischen Szenengraphen will beliebige Renderingsysteme und Renderingverfahren unterstützen. Zu diesem Zweck muss eine Trennung der Szenenmodellierung von der Szenenauswertung vorgenommen werden. Ein generischer Szenengraph kann so zum Beispiel mit Hilfe eines Echtzeit-Renderingsystems oder skizzenhaft mit Hilfe eines nichtphotorealistischen Renderingsystems ausgegeben werden. Zugleich könnte ein Sound-Renderingsystem gemeinsam mit einem Echtzeit-Renderingsystem auf einem generischen Szenengraphen zur audio-visuellen Ausgabe operieren.

Zur technischen Umsetzung der Trennung von Szenenmodellierung und Szenenauswertung dienen Auswertungsstrategien in Form von Techniken, Auswertungsalgorithmen in Form von Handlern und Auswertungsmaschinen in Form von Engines.

Unterstützung von Multi-Pass-Renderingverfahren

Die Arbeit an dem generischen Szenengraphen war nicht zuletzt durch die steigende Zahl von Echtzeit-Renderingverfahren motiviert, die Per-Pixel-Beleuchtungsverfahren und Mehrfachtexturierungsverfahren einsetzen und damit eine wesentliche Qualitätssteigerung bei interaktiver Computergraphik im Sinne des Pseudorealismus ermöglichen. Diese Verfahren lassen sich derzeit nicht in bestehende Szenengraph-Systeme einfügen, da deren Entwurf kein Multi-Pass-Rendering vorsieht. Mit den Techniken des generischen Renderingsystems, die sich als Attribute in einen generischen Szenengraphen einfügen lassen, können jedoch einfach und effizient Multi-Pass-Renderingverfahren implementiert werden. Sie ermöglichen es, Subgraphen eines generischen Szenengraphen mehrfach zu traversieren.

Unterstützung von deklarativer Szenenmodellierung

Überdies kann der generische Szenengraph deklarative Szenenmodellierung unterstützen. *Deklarative Szenenmodellierung* bedeutet in diesem Zusammenhang, dass die Szenengraphkomponenten sich in ihrer Anordnung weniger an der Ordnung, in der sie einem zugrundeliegenden Renderingsystem mitgeteilt werden müssen, orientieren, sondern in ihrer Anordnung rein konzeptionellen Ordnungskriterien folgen können. Bei der Spezifikation von Lichtquellen fordern klassische Szenengraphen zum Beispiel im Allgemeinen, dass die Lichtquellen, die auf alle Szenenobjekte wirken sollen, im Wurzelbereich des Szenengraphen abgelegt werden müssen, d.h. dass während der Traversierung die Lichtquellen aktiviert werden, bevor in die einzelnen Subgraphen, die die Szenenobjekte beschreiben, verzweigt wird. Bei OpenGL ist diese Forderung z.B. dadurch begründet, dass jede Szenengeometrie unmittelbar gerendert und somit

auch beleuchtet wird, d.h. alle Lichtquellen müssen zum Zeitpunkt der Auswertung der einzelnen Shapes aktiviert sein. Eine deklarative Spezifikation hingegen würde es erlauben, dass Lichtquellen beliebig in Subgraphen enthalten sein können (und damit z.B. in einem lokalen Koordinatensystem). Beim generischen Szenengraphen kann eine stärker deklarative (wenngleich auch nicht vollständig deklarative) Szenenmodellierung dadurch ermöglicht werden, dass ein Szenengraph vor seiner eigentlichen Auswertung im Rahmen einer Vorauswertung analysiert wird. Die Unterstützung deklarativer Szenenmodellierung verbessert sowohl die Benutzbarkeit als auch die Kompaktheit der Szenenspezifikation. Hinzu kommt, dass der Aufwand für die Vorauswertung eines Szenengraphen (pro Frame) durch Optimierung so minimiert werden kann, dass kein signifikanter Geschwindigkeitsverlust eintritt.

4.2 Szenenrepräsentation in computergraphischen Systemen

Dieser Abschnitt stellt vor, wie heutige computergraphische Systeme Szenen repräsentieren. Eine Analyse und Evaluation der Software-Architektur dieser Systeme wurde bereits in Kapitel 2 vorgestellt.

OpenInventor, Java3D und VRML

Mit OpenInventor [101] wurde die klassische Form eines Szenengraphen eingeführt, die von vielen nachfolgenden Systemen (z.B. Java3D [100] und VRML [51]) übernommen wurde. Ein wesentliches Merkmal des OpenInventor-Szenengraphen ist es, dass sich die Rendering-Pipeline in der Anordnung der graphischen Primitive zu einem gewissen Grad widerspiegelt. Konkret bedeutet dies, dass die Elemente der Szenenbeschreibung so angeordnet werden müssen, dass sie bei einem depth-first-Traversal des Szenengraphen unmittelbar von der Rendering-Pipeline verarbeitet werden können (z.B. erfolgt die Definition der Kameraparameter im Wurzelknoten). Diese Graphiksysteme sind für das Echtzeit-Rendering ausgelegt und müssen folglich die Rendering-Pipeline im Aufbau einer Szenenbeschreibung entsprechend stark berücksichtigen. Andere Renderingverfahren, zum Beispiel photorealistische oder nichtphotorealistische, werden von diesen Systemen nicht unterstützt, da deren Implementierung auf der Grundlage dieser Szenenrepräsentation nicht direkt erfolgen kann.

Ein weiteres Merkmal, das bei der Bewertung der Szenenrepräsentation herangezogen werden kann, ist die Zugriffsmöglichkeit auf die Funktionalität des zugrundeliegenden Renderingsystems. Die OpenInventor-Bibliothek beschränkt den Zugriff auf das Renderingsystem (OpenGL) nicht, d.h. Anwendungen können zum Beispiel zusätzliche OpenInventor-Knoten implementieren, die direkt OpenGL-Funktionalität verwenden. Im Gegensatz dazu stellen Java3D und VRML keinerlei Zugriffsmöglichkeiten auf das Renderingsystem bereit; die Erweiterbarkeit dieser Systeme ist von daher fundamental eingeschränkt.

BOOGA, PREMO und Generic-3D

BOOGA [2], PREMO [22] und Generic-3D [8] sind graphische Bibliotheken und Frameworks, die von einer objektorientierten Software-Architektur geprägt sind. Sie basieren auf OpenGL für das Echtzeit-Rendering und implementieren daneben meist zusätzliche Renderingverfahren mit einem globalen Beleuchtungsmodell. Der Schwerpunkt dieser Systeme liegt jedoch weder auf der Verallgemeinerung der Szenenrepräsentation noch auf der Nutzung möglichst verschiedener Renderingsysteme – sie zeigen, wie durch eine objekt- und komponentenorientierte Software-Architektur Graphiksysteme implementiert werden können. PREMO stellt Szenen in Analogie zum OpenInventor-Szenengraphen dar [109], wohingegen Generic-3D verschiedene low-level Datenstrukturen anbietet, die zur Implementierung von Szenenrepräsentationen ver-

wendet werden können; die Implementierung einer Szenenrepräsentation ist somit Aufgabe der Anwendungsprogramme oder geschieht in den von Generic-3D abgeleiteten Graphiksystemen. Die Szenenrepräsentation selbst wurde im Vergleich zu OpenInventor jedoch nicht weiterentwickelt.

GRAMS

Das GRAMS-Graphiksystem [29] findet sich als eines der wenigen, das Renderingalgorithmen von den Shapes separiert. Der Schwerpunkt seiner Systemarchitektur liegt auf der dynamischen Auswahl von Shape-Renderingalgorithmen anhand von Effizienz-Kriterien [30]. Dadurch ist es möglich, verschiedene Renderingsysteme optimal anzusteuern. Die Szenenrepräsentation erfolgt jedoch in Analogie zu OpenInventor, ohne dass dabei Algorithmen zur Umsetzung von Attributen möglich sind und auch keine Renderingverfahren im Graphen eingefügt werden können.

Vision

Die Software-Architektur des Vision-Systems [92], das den Schwerpunkt auf die physikalisch-basierte Simulation von Licht legt, trennt strikt geometrische Primitive von graphischen und geometrischen Attributen. Die Software-Architektur ist von einem durchgängigen objektorientierten Ansatz geprägt [93]. Die Szenendarstellung erfolgt mit Hilfe eines OpenInventor-ähnlichen Szenengraphen, in dem jedoch Seiteneffekte auf Geschwisterknoten, wie sie im OpenInventor-Szenengraph möglich sind, ausgeschlossen werden. Im Vergleich dazu konzentriert sich der hier vorgestellte generische Szenengraph auf die Unterstützung von beliebigen Renderingsystemen und Multi-Pass-Renderingverfahren.

Interaktion mit Szenenobjekten und deren Animation werden als komplementäre Aspekte der Szenenrepräsentation hier nicht betrachtet, da sie nicht Gegenstand dieser Arbeit sind. Abstraktere Software-Komponenten, die Interaktions- und Animationsfunktionalität bereitstellen, können allerdings auf der Grundlage des generischen Szenengraphen erstellt werden.

4.3 Konzepte des generischen Szenengraphen

Der generische Szenengraph führt eine strikte *Trennung von Struktur und Inhalt in Szenenbeschreibungen* ein: Die Struktur wird über einen gerichteten, azyklischen Graphen beschrieben; die Knoten eines Graphen werden durch Knotenobjekte implementiert, und die Kanten des Graphen werden durch Eltern-Kind-Beziehungen zwischen den Knotenobjekten modelliert. Den Inhalt eines Knotens bilden die Renderingobjekte.

Ein generischer Szenengraph ist hinsichtlich seiner Auswertung durch Auswertungsmaschinen, den Engines, parametrisiert, d.h. die Interpretation des Graphen obliegt nicht den Knoten, sondern den Engines, die den Graphen traversieren. Die Knotenobjekte übergeben die in ihnen enthaltenen Renderingobjekte der Engine während der Traversierung zur Auswertung. Aufgrund der Fähigkeit des generischen Renderingsystems, verschiedene Auswertungsstrategien zu modellieren, kann der generische Szenengraph so mit unterschiedlichen Auswertungsstrategien, z.B. unterschiedlichen Renderingverfahren, ausgewertet werden. Dabei ermöglicht es das generische Renderingsystem, dass innerhalb eines Szenengraphen Attribute enthalten sein dürfen, die nur von bestimmten Auswertungsstrategien berücksichtigt werden.

Die Komponenten eines generischen Szenengraphen und dessen Auswertung durch sind in Abbildung 22 illustriert. Der hier vorgestellte Ansatz zur Szenenmodellierung betont die

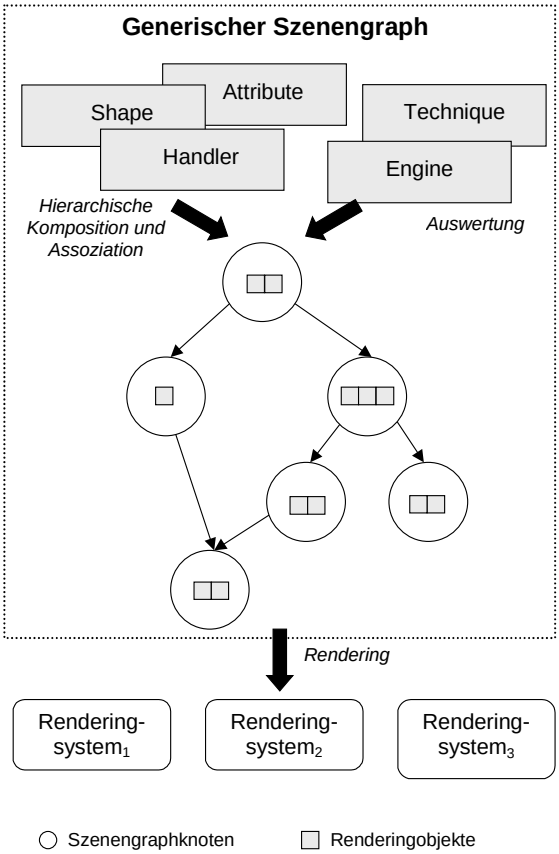

Abbildung 22. Aufbau und Auswertung generischer Szenengraphen.

abstrakte Spezifikation von graphisch-geometrischen Sachverhalten – die Implementierung von Auswertungsstrategien ist nicht seine Aufgabe, sondern deren Nutzung.

4.3.1 Renderingobjekte

Zum Aufbau eines generischen Szenengraphen stehen alle Renderingkomponenten des generischen Renderingsystems zur Verfügung. Aus der Sicht der Szenenmodellierung werden die Renderingkomponenten wie folgt unterschieden:

- *Shapes*
- *Attribute*
- *Monoattribute*
- *Polyattribute*
- *Handler*
- *Techniken*

4.3.2 Szenengraphknoten

Szenengraphknoten dienen als Behälter für Renderingobjekte und Subgraphen. Ein Subgraph wird dadurch in einen Elternknoten eingefügt, indem der Wurzelknoten des Subgraphen in den Elternknoten eingefügt wird.

Szenengraphknoten können direkt von Seite des Anwendungsprogramms mit Renderingobjekten befüllt werden. Dazu definiert der generische Szenengraph einen universellen

Szenengraphknoten *SceneThing*. Ein SceneThing kann gemischt sowohl Renderingobjekte als auch Kindknoten enthalten. Die Komponenten werden intern in einer inhomogenen Liste verwaltet. Die Darstellung eines universellen Szenengraphknotens kann inhaltsbezogen (s. Abbildung 23 links) oder strukturbezogen (s. Abbildung 23 Mitte) erfolgen; beide Darstellungsformen sind äquivalent. Bei der Auswertung entscheidet der Typ der Komponente, wie mit der Komponente verfahren wird: Renderingobjekte werden der Engine zur Auswertung übergeben, wohingegen für Kindknoten die Auswertung rekursiv fortgesetzt wird. In dem vorgestellten universellen Szenengraphknoten werden Renderingobjekte und Szenengraphknoten gleichwertig als Inhaltskomponenten behandelt, d.h. Knotenobjekte und Renderingobjekte können beliebig gemischt in der Inhaltsliste eines Knotenobjekts stehen. Der Grund für dieses Herangehen ist pragmatischer Natur: Die Szenenmodellierung wird vereinfacht, da Attributierung und Subgraphenbildung mit Hilfe eines einzigen Typs von Szenengraphknoten vorgenommen werden können. Somit muss auch nur eine einzige Knotenklasse bereitgestellt werden, um generische Szenengraphen konstruieren zu können.

Neben dem universellen Szenengraphknoten können spezialisierte Szenengraphknoten ihren Inhalt auch selbst festlegen. Diese Art von Szenengraphknoten verfolgen im Allgemeinen das Ziel, die Szenenmodellierung zu automatisieren. Ein Beispiel hierfür ist der Justierungsknoten, der die in ihm enthaltenen Szenenobjekte derart transformiert, dass sie exakt in eine vorgegebene Bounding-Box passen. Bei jeder Auswertung des Knotens durch eine Engine ermittelt der Knoten die aktuelle Bounding-Box seiner Szenenobjekte, berechnet eine geometrische Transformation, übermittelt diese Transformation der Engine und wertet erst dann seinen Inhalt aus.

Szenengraphknoten verfügen über Methoden zur Verwaltung ihres Inhalts, der Renderingobjekte und ihrer hierarchischen Struktur, d.h. ihrer Eltern-Kind-Knotenbeziehungen. Szenengraphknoten stellen außerdem Traversierungsmethoden bereit. Zwei Traversierungstypen werden hier unterschieden, Auswertung und Inspektion, die beide auf der Grundlage des Besucher-Entwurfsmusters (Visitor-Entwurfsmuster [34]) implementiert wurden.

4.3.3 Szenengraphauswertung

Die *Auswertung* eines Szenengraphen verfolgt das Ziel, den Inhalt in seiner hierarchischen Anordnung als Beschreibung eines geometrisch-graphischen Sachverhalts zu interpretieren und mit Hilfe einer Engine in ein Zielmedium umzusetzen. Der Szenengraph wird dazu mit einer Engine als Parameter in depth-first-Order traversiert, wobei ein Knoten die Traversierung lokal einer vorhandenen Technik übertragen kann. Die Ordnung der Komponenten in der Inhaltsliste eines Knoten ist entscheidend: Attribute, Handler und Techniken wirken nur auf die ihnen in der Inhaltsliste nachfolgenden Komponenten.

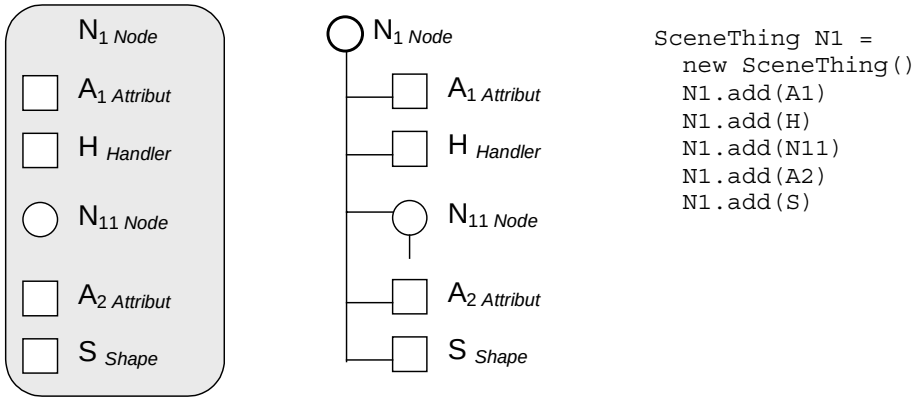

Abbildung 23. Inhaltsbezogene Darstellung eines universellen Szenengraphknotens (links). Strukturbezogene Darstellung des Knotens (Mitte). Konstruktion des Knotens (rechts).

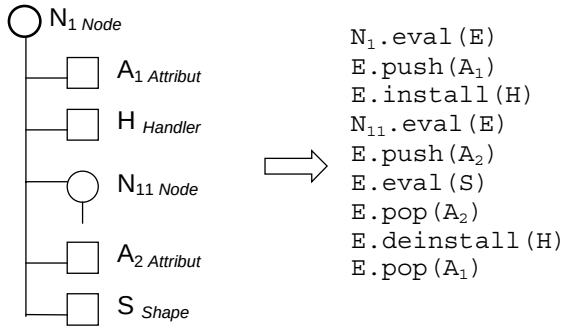

Abbildung 24. Auswertung eines Szenengraphknoten durch Mikroprogrammierung.

Bei der Auswertung eines Szenengraphen obliegt es den Szenengraphknoten, die in ihnen enthaltenen Renderingobjekte mit Hilfe geeigneter Steueranweisungen durch eine Engine auswerten zu lassen. Szenengraphknoten formulieren dazu ein Mikroprogramm mit den Engine-Steueranweisungen. Ein Beispiel eines Szenengraphknoten und dessen Auswertung findet sich in Abbildung 24. Die Implementierung einer allgemeinen Auswertungsmethode soll kurz an einer Implementierungsskizze veranschaulicht werden. Wir definieren dazu eine Klasse *Node*, die ihren Inhalt in einer Liste hält:

```
class Node {
private: List<Object> children;
public:

 void eval(Engine e) {
  unapply(apply(children,e),e);
 }

};
```

Die Methode *eval* der Knotenklasse besitzt als Parameter eine Engine. Die Implementierung besteht aus zwei Funktionen, eine zum Anwenden der Renderingobjekte auf die Engine und den generischen Kontext (*apply*) und eine zum Restaurieren des Kontexts (*unapply*). Monoattribute werden dadurch auf den ihrer Attributkategorie entsprechenden Stack gelegt bzw. wieder entfernt. Polyattribute werden in die ihrer Attributkategorie entsprechenden Liste eingefügt bzw. entfernt. Handler werden in der Handler-Tabelle des Kontexts installiert bzw. deinstalliert. Die *push*-, *pop*-, *add*- und *remove*-Methodenaufrufe induzieren eine Suche nach einem geeigneten Handler für die Attribute, ebenso der *eval*-Methodenaufruf der Engine. An diesen Stellen „passiert“ nun die eigentliche Renderingarbeit: Die so indirekt aufgerufenen Handler
übernehmen die Umsetzung der Attribute und Shapes für das gegebene Zielrenderingsystem.

```
proc apply(List list, Engine e ) : Stack {
 Stack s = {}

 iterate c through list {
  if(c is a shape) e.eval(c)
  else if(c is node) c.eval(e) // rekursive Auswertung der Kindknoten
  else {
   s.push(c)
   if(c is a mono attribute) e.push(c)
   else if(c is a poly attribute) e.add(c)
   else if(c is a handler) e.install(c)
  }
 }

 return s
}
```

```
proc unapply(Stack s, Engine e) {
 while s not empty {
  Object c ← s.pop()
  if(c is a mono attribute) e.pop(c)
  else if(c is a poly attribute) e.remove(c)
  else if(c is a handler) e.deinstall(c)
 }
}
```

Durch die Konstruktion der Auswertungsmethode wird sichergestellt, dass Attribute und Handler sich niemals auf Geschwisterknoten auswirken: Der Aufruf von *apply* modifiziert den generischen Kontext und der Aufruf von *unapply* restauriert den Zustand vollständig. Attribute und Handler eines Knoten wirken allenfalls „in die Tiefe“ des Graphen. Anzumerken ist noch, dass in der obigen Darstellung die Behandlung der Technik ausgeklammert wurde; sie wird im Abschnitt 4.6 beschrieben.

4.3.4 Szenengraphinspektion

Im Gegensatz zur Auswertung wird bei einer *Inspektion* der Szenengraph in depth-first-Order traversiert, ohne dass dabei der Szenengraphinhalt die Traversierung beeinflussen kann. Die Inspektion dient der Exploration der Struktur und des Inhalts eines Szenengraphen. Die Inspektion findet zum Beispiel Anwendung, wenn ein Szenengraph in einem Fenster der Benutzerschnittstelle editiert werden soll. Zur Editierung muss der Szenengraph nicht interpretiert, sondern in seiner Struktur ausgelesen und ggf. modifiziert werden können. Die Implementierung einer allgemeinen Inspektionsmethode wird im folgenden skizziert.

```
class Node {
private: List<Object> children;
public:

 void inspect(Visitor v) {
  iterate c through children {
   v.explore(c)  // inspect single component
   if(c is a node) {
    c.inspect(v) // inspect subgraph
   }
  }
 }

};
```

Das Visitor-Objekt kapselt die Inspektionsfunktionalität. Jeder Knoten, der mit einem Visitor-Objekt besucht wird, teilt dem Visitor-Objekt seinen Inhalt Objekt für Objekt mit. Handelt es sich um ein Knotenobjekt, dann wird die Inspektion rekursiv mit dem Kindknoten fortgesetzt.

4.3.5 Entwurfsprinzipien des generischen Szenengraphen

Beim Entwurf des generischen Szenengraphen wurde versucht, einerseits die Vorzüge und die Natur des OpenInventor-Szenengraphen beizubehalten, andererseits seine Beschränkungen weitestgehend durch den Einsatz eines grundverschiedenen Objektmodells aufzuheben. Die Kriterien, die zum generischen Szenengraphen geführt haben, sind:

- *Strikte Separation von Shapes und Attributen* in Analogie zur Szenendarstellung im Vision-System. Erweitert wurde dieser Ansatz noch dadurch, dass bezüglich der Attributkategorien keinerlei Beschränkungen bestehen. Insbesondere können renderingsystemspezifische und anwendungsspezifische Attribute dadurch integriert werden.
- *Strikte Separation der Deklaration von Szenen und der Implementierung der Renderingverfahren und -algorithmen.* Renderingalgorithmen und Renderingverfahren für Rendering-

objekte werden durch Handler und Techniken gekapselt. Daher können Shape- und Attributklassen „leichtgewichtig“ entworfen werden, und Algorithmen können partiell innerhalb der Szenenbeschreibung durch die Installation neuer Handler ausgetauscht werden.

- *Separation von Struktur und Inhalt in der Szenenbeschreibung.* Szenengraphen stellen durch die Szenengraphknoten hierarchische Strukturen bereit, die den Inhalt (d.h. die Renderingobjekte) für die Auswertung anordnen. Dadurch können sowohl die Strukturkomponenten als auch Inhaltskomponenten getrennt voneinander weiterentwickelt werden.
- *Szenengraphen sind bezüglich des Auswertungsverfahrens parametrisiert.* Ein Szenengraph kann nur in Verbindung mit einer Engine, ihren Handlern und Techniken ausgewertet werden; die Engine parametrisiert den Szenengraph. Dadurch kann ein und derselbe Szenengraph für vollkommen unterschiedliche Renderingverfahren genutzt wurden; der Szenengraph enthält eine abstrakte Beschreibung eines graphisch-geometrischen Sachverhalts.

Die strikte Unterscheidung von Shapes und Attributen, Struktur und Inhalt sowie von Deklaration und Implementierung führt zu einer feinkörnigen Sammlung zueinander orthogonaler Komponenten für die Szenenmodellierung. Die unterschiedlichen Auswertungsstrategien werden durch eine Parametrisierung des generischen Szenengraphen mit der Hilfe von Engines realisiert.

4.4 Nutzung von Renderingsystemen

Um ein konkretes Renderingsystem effizient und vollständig mit einem generischen Szenengraphen anzusteuern, bedarf es spezifischer Handler und spezifischer Attribute.

4.4.1 Code-Integration durch native Handler

Damit eine effiziente Umsetzung von Shapes und Attributen auf ein konkretes Renderingsystem gewährleistet wird, ist die Entwicklung von *nativen*[*] *Handlern* für möglichst viele der eingebauten Shapes und Attribute notwendig. Ein Anwendungsprogramm kann daneben weitere native Handler für Shapes und Attribute bereitstellen, die das Rendering unter von der Anwendung festgelegten Rahmenbedingungen optimieren. Zum Beispiel wäre es denkbar, dass eine Anwendung spezielle OpenGL-Handler für ausgewählte Shapes und Attribute bereitstellt, die die neu hinzugekommenen Funktionen von OpenGL 1.2 im Vergleich zu Standard-OpenGL-Handler, die auf der Version 1.1 basieren, nutzen. Die Handler müssen hier insbesondere zur Laufzeit prüfen, welche OpenGL-Extensions konkret bereitgestellt werden.

Das Handler-Konzept eignet sich auch zur effizienten Integration von bereits existierenden Graphikcodefragmenten. Durch spezialisierte, anwendungsspezifische Handler lassen sich diese Fragmente direkt in die Szenenrepräsentation einbinden und damit Code effektiv wiederverwenden.

Zum Beispiel könnte eine Anwendung einen Parser für Wavefront-3D-Objekt-Dateien wiederverwenden, der solche Modelle aus Dateien einliest und sie mit Hilfe von OpenGL-Kommandos rendert. In diesem Fall muss die Anwendung folgende Klassen implementieren:

[*] „Nativ“ bedeutet in diesem Zusammenhang, dass die „einheimischen“ Funktionen des zugrundeliegenden Renderingsystems direkt genutzt werden; die gleiche Sprachregelung findet sich bei nativen Implementierungen von Java-Methoden.

- *Shape-Klasse:* Objekte dieser Klasse speichern den Dateinamen und übernehmen das Parsing des Dateiinhalts. Sie enthalten zusätzlich die interne Datenstruktur, die der Parser erzeugt.
- *OpenGL-spezifische Shape-Painter-Klasse:* Diese Klasse „ummantelt" das Renderingcodefragment, das direkt auf der geparsten Darstellung eines Wavefront-3D-Objekts operiert, und führt das Codefragment aus, wenn der Handler von der Engine aktiviert wird.
- *Shape-Simplifier-Klasse:* Diese Klasse konvertiert die geparste Darstellung eines Wavefront-3D-Objekts in eine Menge von eingebauten Shape- und Attribut-Objekten; sie sendet diese Objekte zur Engine zur rekursiven Auswertung.

Wird ein Simplifier für die neue Shape-Klasse bereitgestellt, dann können sofort alle anderen Engines die Shape-Objekte des neuen Typs interpretieren. Ansonsten wäre die Interpretation auf das Rendering mit OpenGL-Engines beschränkt. Das Beispiel zeigt zudem die Mächtigkeit der Mikroprogrammierung: Ein Wavefront-Objekt besteht konzeptionell aus Gruppen, für die individuelle Materialeigenschaften festgelegt sein können. Beim Aufruf des Shape-Simplifiers zerlegt dieser ein solches Shape-Objekt in eine Folge von Material-Attribut-Objekten und Polygon-Shape-Objekten, die rekursiv von einer Engine auswertet werden. Die Simplifikation eines Shapes in eine inhomogene Menge von Renderingobjekten ist möglich.

Das Beispiel zeigt, wie der generische Szenengraph existierenden, möglicherweise effizienten und getesteten Code integriert kann. Abbildung 25 zeigt einen generischen Szenengraphen, der eine Szene mit einem Wavefront-Objekt modelliert. Das gezeigte Objekt wird durch das im zweiten Subgraphen befindliche Shape-Objekt vom Typ *WavefrontObject* repräsentiert. Ein Renderingalgorithmus für OpenGL (*WavefrontPainterGL*) und ein allgemein einsetzbarer Simplifikationsalgorithmus (*WavefrontSimplifier*) sind als Handler eingefügt, d.h. sie werden während der Traversierung des Subgraphen in der jeweiligen Engine installiert. Die OpenGL-Engine wird auf den OpenGL-Painter und die RenderMan-Engine auf den Simplifier zurückgreifen.

4.4.2 Renderingsystemabhängige Attributierung

Da graphische und weitere gestalterische Attribute in hohem Maße vom jeweiligen Renderingverfahren abhängen, richten sich die Attribute, die in einem generischen Szenengraphen enthalten sind, tendenziell stark an den Renderingsystemen aus, die zur Auswertung des Szenengraphen verwendet werden. Im generischen Szenengraphen können Attribut-Objekte für unterschiedliche Renderingsysteme nebeneinander abgelegt werden. Attribute werden von solchen Engines ignoriert, die keine passenden Handler für sie besitzen.

Das generische Renderingsystem definiert – wie im letzten Kapitel erläutert – eine Reihe von Standardattributen, die sich grundsätzlich auf alle Renderingsysteme abbilden lassen. Dazu zählen insbesondere die geometrischen Transformationen, die sich in keinem der derzeit unterstützten Renderingsysteme unterscheiden. Für OpenGL steht eine Sammlung spezialisierter Attribute bereit, die vollständig den Funktionsumfang von OpenGL im generischen Szenengraphen zur Verfügung stellt. Insbesondere wurden Attribute implementiert, die die Texturierungsfunktionalität von OpenGL mit Hilfe von Attributen modellieren. Für RenderMan wurden Attribute entworfen, die die dort vorhandene Shader-Language unterstützen und insbesondere kompilierte Shader-Dateien zu interpretieren in der Lage sind. Der Szenengraph in Abbildung 25 enthält RenderMan-spezifische Attribute, die *RenderManShader*-Objekte.

Die spezifischen Attribute für einzelne Renderingsysteme versuchen, die Produktion von hochqualitativen Bildern und Filmen zu vereinfachen. Sind diese speziellen Attribute einmal in einem generischen Szenengraphen enthalten, dann kann dieser Szenengraph mit unterschiedli-

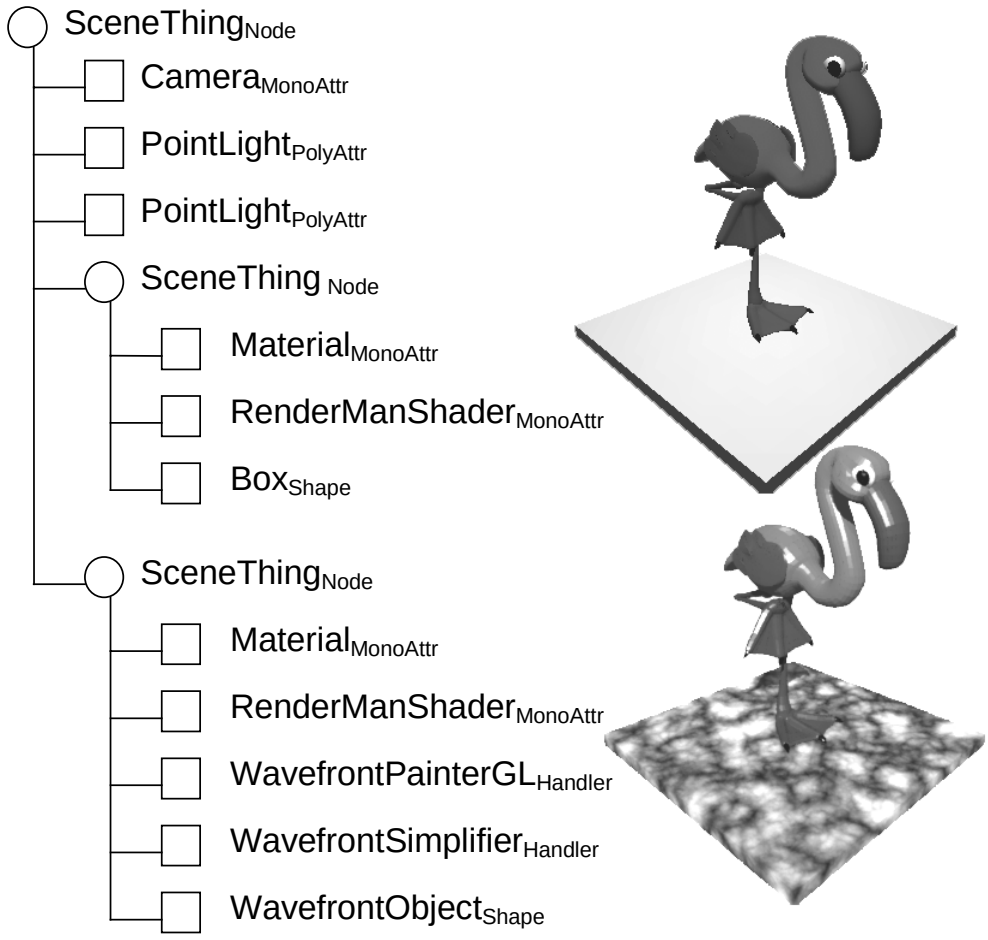

Abbildung 25. Beispiel eines generischen Szenengraphen mit nativen Handlern und Renderingsystem-spezifischen Attributen (links). Bildsynthese durch die OpenGL-Engine und RenderMan-Engine (rechts).

chen Engines wieder und wieder ausgewertet werden, ohne dass dabei manuell die Ausgabe nachzubearbeiten ist. Speziell kann damit die Planung von Animationen in Echtzeit mit OpenGL einfach realisiert werden, und die spätere Ausgabe automatisiert mit einem photorealistischen Renderingsystem erfolgen. Somit stehen alle unterstützten Renderingsysteme unter einer einheitlichen Anwendungsprogrammierschnittstelle (API) dem Anwendungsentwickler zur Verfügung.

Die Unterstützung unterschiedlicher Renderingsysteme erfolgt durch Parametrisierung des generischen Szenengraphen mit Engines. Native Handler können in Form neuer Shape-Typen direkt existierenden Graphik-Programmcode in einem generischen Szenengraph integrieren. Feinheiten, Besonderheiten und Stärken einzelner Renderingsysteme können in generischen Szenengraphen mit Hilfe Renderingsystem-spezifischer Attribute modelliert werden.

4.5 Integration von Multi-Pass-Rendering

Das Multi-Pass-Rendering ist ein charakteristisches Element von Hardware-beschleunigten Schattierungs-, Beleuchtungs- und Modellierungstechniken. Diese Verfahren implementieren zum Beispiel komplexe Beleuchtungsmodelle [46], Bump-Mapping für Oberflächen [52] oder

Constructive-Solid-Geometry-Modellierung im Bildraum [114]. In generischen Szenengraphen lassen sich diese Verfahren mit Hilfe der im generischen Renderingsystem vorhandenen Technik-Objekte integrieren.

Technik-Objekte können innerhalb eines Szenengraphen geschachtelt auftreten, wobei sie wie andere Attribute eingefügt werden. Jedes Technik-Objekt kann eine mehrfache Traversierung des Subgraphen, der durch seinen Behälterknoten definiert ist, bewirken. An das Technik-Objekt werden während dieser Traversierung alle Auswertungskommandos der Engine delegiert.

Die Auswertungsmethode eines Szenengraphknoten muss folglich die Kontrolle einem angetroffenen Technik-Objekt übergeben. Im folgenden ist eine modifizierte Fassung der oben vorgestellten Auswertungsmethode für die Beispielimplementierung einer Szenengraphknoten-Klasse skizziert.

```
proc apply(List C, Engine E) : Stack {
 List P = C    // objects to be processed
 Stack S = {}  // objects to be unapplied
 while P not empty do {
  c ← first element of P
  P.remove(c)
  if(c is a shape) E.eval(c)
  else if(c is a node) c.eval(E)
  else {
   S.push(c)
   if(c is a technique ) {
    E.push(c)
    if(c is a technique applicable to E) {
     c.start(E);
     while(c.needsPass(E)) {
      unapply(apply(P,E),E)
     }
     c.stop(E)
     P ← ∅
    }
   }
   else if(c is a mono attribute) E.push(c)
   else if(c is a poly attribute) E.add(c)
   else if(c is a handler) E.install(c)
  }
 }
 return S
}
```

Jedes Technik-Objekt ist in der Lage zu klären, ob es für eine gegebene Engine anwendbar ist oder nicht. Ist das Technik-Objekt nicht anwendbar, dann stellt es sich funktionslos und hat keine Wirkung auf die Auswertung.

> In generischen Szenengraphen sind Multi-Pass-Renderingverfahren durch Technik-Objekte integrierbar. Den Technik-Objekten wird bei der Traversierung des Szenengraphen lokal die Steuerung des Kontrollflusses übertragen. Ihre Handhabung bei der Szenenmodellierung ist einfach, denn sie unterscheidet sich nicht von der der Attribute.

4.6 Deklarative Szenenmodellierung

Elemente einer deklarativen Szenenmodellierung erleichtern die Spezifikation von Szenen insbesondere im Bereich des stark Pipeline-orientierten Echtzeit-Renderings. Klassische Szenengraphen werten die Szenengraphknoten in depth-first-Order aus, so dass bei der Traversierung

keine globale Information über die Szene bereitsteht. Für das Echtzeit-Rendering bedeutet dies konkret zum Beispiel, dass alle Lichtquellen in der Nähe des Wurzelknotens stehen müssen, andernfalls wären die Lichtquellen nur für diejenigen Szenenobjekte wirksam, die im Subgraphen des Knotens enthalten sind, der die Lichtquelle enthält. Die Kameraeinstellung muss im Wurzelknoten vorliegen, denn sonst kann die Verarbeitung der Szenenobjekte in OpenGL nicht erfolgen, da die Projektionstransformation beim Start des Frame-Renderings festgelegt werden muss. Ein klassischer Szenengraph ist von daher stets so aufgebaut, dass sein Inhalt mit der Rendering-Pipeline konsistent ist.

Im generischen Szenengraph wurde hier eine Abhilfe geschaffen: Durch eine Vorauswertung (Pre-Traversal) des Szenengraphen besteht die Möglichkeit, globale Informationen zu sammeln. Insbesondere werden dabei berücksichtig:

- *Lichtquellen*: Während der Vorauswertung werden alle angetroffenen Lichtquellen unter Berücksichtigung ihres lokalen Koordinatensystems im Renderingsystem angemeldet und aktiviert. Während der Haupttraversierung sind sie somit global aktiv. Ferner lassen sich im Szenengraph Lichtschalter-Attribute einfügen, die die Aktivierung und Deaktivierung einer angemeldeten Lichtquelle steuern. Dadurch kann eine Lichtquelle für einen Subgraphen einzeln an- und ausgeschaltet werden.
- *Kamera*: Die Kameraeinstellung wird im generischen Renderingsystem als Attribut modelliert, das den Kamerastandpunkt, die Kamerarichtung und weitere optische Parameter enthält. In der Vorauswertung wird nach diesem Attribut gefahndet, um die Position und Richtung der Kamera in Weltkoordinaten zu erhalten. Auf diesem Weg kann die Kamera als Szenenobjekt modelliert werden, das in einem lokalen Koordinatensystem eingebettet sein darf und auf das geometrische Transformationen wirken können (z.B. Bewegung durch Translation).

Eine weitere Untersuchung des Einsatzes deklarativer Elemente ist noch notwendig. Besonderes Interesse gilt der Optimierung einer kombinierten Vor- und Hauptauswertung: Geometrische Transformationen können lokal im Szenengraphen vorberechnet werden, so dass in der Hauptauswertungsphase diese Berechnungen nicht wiederholt werden müssen. Weiter kann während der Vorauswertung auf die Auswertung der graphischen Attribute fast vollständig verzichtet werden, denn sie sind nicht für die Lichtquellen- und Kamerakonfiguration notwendig.

> In generischen Szenengraphen wird die deklarative Szenenmodellierung dadurch ermöglicht, dass durch eine Vorauswertung des Szenengraphen globale Szeneninformationen gesammelt werden können, die in die Hauptauswertung eingehen. Die Kosten für eine zweifache Traversierung eines Szenengraphen bei einer Auswertung lassen sich durch Optimierung der Szenengraphen-Implementierung gering halten.

4.7 Graphisch-geometrische Sachverhalte

Ein generischer Szenengraph kann zur Kodierung von *graphisch-geometrischen Sachverhalten* verwendet werden. Der graphisch-geometrische Sachverhalt bedeutet im Vergleich zu klassischen Szenengraph-basierten Systemen insofern eine Erweiterung, als er die Art der Umsetzung in ein Zielmedium nicht festlegt, sondern durch die Engines parametrisiert.

Im graphisch-geometrischen Sachverhalt können darüber hinaus simultan mehrere Auswertungsalgorithmen und Auswertungsverfahren in Form von Engine-spezifischen Handlern und Techniken abgelegt werden. Erst in Verbindung mit einer geeigneten Engine tragen sie zur

Interpretation des Inhalts des Sachverhalts bei. Ebenso kann spezifisch für eine Engine attributiert werden.

Der generische Szenengraph besitzt somit insbesondere die Voraussetzung zur Modellierung multimedialer Dokumente in Verbindung mit Echtzeit-Rendering-Engines und Sound-Rendering-Engines.

5 FALLSTUDIE: MULTI-PASS-RENDERING

Im Bereich des Echtzeit-Renderings vollzieht sich derzeit ein grundlegender Wandel im konzeptionellen Herangehen an das Szenen-Rendering. Die Optimierung geometrischer Modelle war bislang eines der zentralen Themen, denn nur durch eine angemessene Reduktion der geometrischen Komplexität der darzustellenden Szenengeometrie konnte eine interaktive Bildwiederholrate erreicht werden. Wege dazu sind Verfahren zur Multiresolutionsmodellierung von geometrischen Modellen, Kompression von geometrischen Modellen und Verfahren des Image-Based-Renderings.

Mittlerweile ist – auch durch den Fortschritt auf der Seite der Computergraphik-Hardware – eine für interaktive Anwendungen arbeitsfähige Geschwindigkeit bei geometrisch komplexen Szenen mit heutiger „low-cost“ Hardware erreichbar. Um so mehr drängt sich die Frage in den Vordergrund, wie die Bildqualität durch den Einsatz neuer, Hardware-beschleunigbarer Verfahren gesteigert werden kann und wie durch einen kreativen Einsatz der Hardware-Beschleunigung neue Renderingverfahren realisiert werden können.

Das Multi-Pass-Rendering repräsentiert ein Merkmal einer umfangreichen Gruppe von Echtzeit-Renderingverfahren, die realistische Beleuchtungs- und Schattierungstechniken bereitstellen; eine Einführung findet sich hierzu bei Diefenbach [19]. Beispiele für derartige Verfahren sind bildbasierte Verfahren zur Simulation realistischer Kameralinsen [47], Beleuchtungsberechnungen auf der Grundlage physikalisch-basierter Reflexionsmodelle [45], Bump-Mapping zur Erhöhung des visuellen Details ebener Geometrien [52], bildpräzise Berechnung von Schatten und Spiegelung [53] sowie nicht-diffuse, globale Illumination mit Hilfe von Environment-Maps [46].

Eine weitere Gruppe von Renderingverfahren, die auf der Basis von Multi-Pass-Rendering verwirklicht werden können, stellen echtzeitfähige nichtphotorealistische Verfahren dar. Insbesondere werden hier mehrfache Renderingdurchläufe eingesetzt, um im Framebuffer Teilbilder zu erstellen und zu kombinieren (z.B. zur Konstruktion von G-Buffern nach Saito und Takahashi [86]). Beispiele hierfür sind die Berechnung der bildpräzisen Silhouetten und der verdeckten Linien [52].

In dieser Fallstudie wird gezeigt, wie Multi-Pass-Renderingverfahren in das generische Renderingsystem und den generischen Szenengraph integriert werden können. Illustriert wird

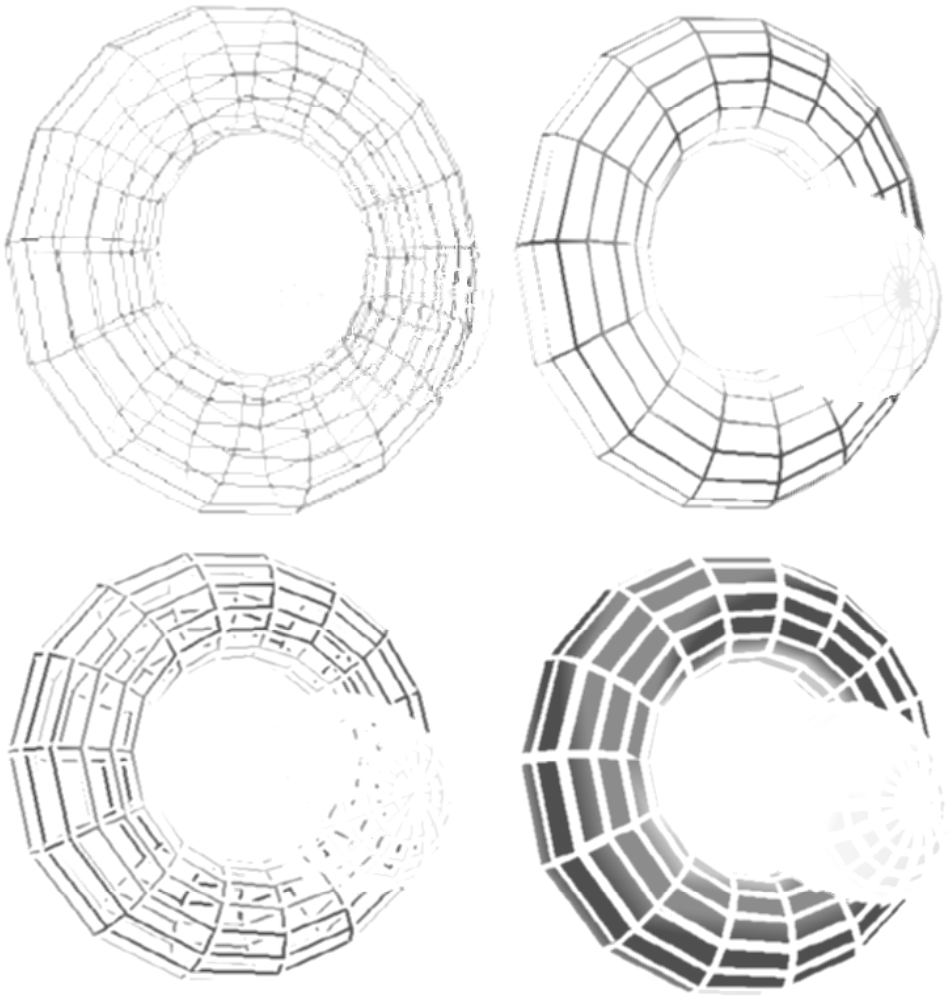

Abbildung 26. Linienstile durch Multi-Pass-Rendering mit OpenGL. Drahtgitterdarstellung (oben links). Drahtgitterdarstellung der sichtbaren Linien (oben rechts). Haloe-Darstellung des Drahtgittermodells (unten links). Fragmentierte Darstellung der Oberflächen (unten rechts).

das Vorgehen an den Beispielen einer Technik-Klasse für Linienzeichnung und einer Technik-Klasse für Reflexion, Schattenwurf und Bump-Mapping.

5.1 Multi-Pass-Linienzeichnungen

Linienzeichnungen repräsentieren Abstraktionen komplexer geometrischer Objekte. Sie ermöglichen es, die Darstellung zu vereinfachen, indem der Detaillierungsgrad der Darstellung reduziert wird. Eine ausführliche Darstellung von Verfahren zum Linienzeichnen und ihr Einsatz als Visualisierungstechnik findet sich bei Strothotte [96].

Die hier vorgestellten Verfahren zum Zeichnen von Linien arbeiten pixelpräzise und erfordern kein Preprocessing der Szenenobjekte. Ziel ist es, die Kanten eines Objektes (im Bildraum) zu klassifizieren und diese auf der Grundlage der Klassifikation in besonderer Weise zu zeichnen. Das Hauptproblem der linienbasierten Darstellung von Objekten mit OpenGL liegt darin, dass OpenGL unterschiedliche Rasterisierungsalgorithmen für Linien und Polygone verwendet. Die Randpixel eines gefüllt gezeichneten Polygons weichen von den Pixeln ab, die

entstehen, wenn der Rand des Polygons mit Hilfe von Linien gezeichnet wird. Visuelle Artefakte sind die Folge [48].

Im folgenden sei die Oberfläche eines Szenenobjekts durch eine Menge von Polygonen beschrieben. Die Polygone bilden ein Oberflächennetz. Bezüglich der Geschlossenheit, des Selbstschnitts oder des Genus sind keine Einschränkungen gegeben.

5.1.1 Multi-Pass-Renderingalgorithmen für Liniendarstellungen

Die hier vorgestellten Renderingalgorithmen für Liniendarstellungen zählen zu den sog. *Advanced Graphics Programming Techniques* [62] für das Renderingsystem OpenGL. Abbildung 26 zeigt eine Szene mit zwei Objekten, die mit diesen Verfahren gerendert wurden.

5.1.1.1 Hidden-Line-Stil

Dieser Stil wird für Drahtgitter-Darstellungen von Szenenobjekten benötigt und klassifiziert die Linien danach, ob sie verdeckt oder sichtbar sind. Die verdeckten Linien können z.B. vollständig entfernt oder graphisch unterschiedlich (im Vergleich zu den sichtbaren Linien) gezeichnet werden. Abbildung 26 (oben rechts) zeigt ein Objekt, dessen verdeckte Linien entfernt wurden.

Der Renderingalgorithmus benötigt drei Durchläufe. Im ersten Durchlauf werden die verdeckten Linien gezeichnet, im zweiten Durchlauf wird der Tiefenbuffer mit dem gefüllt gezeichneten Objekt aktualisiert und im dritten Durchlauf werden die sichtbaren Linien bzw. Linienfragmente gezeichnet:

1. Durchlauf:
   - Freigabe des Color-Buffers (*glColorMask*).
   - Freigabe des Tiefen-Buffers und Aktivierung des Tiefentests (*glDepthBuffer*, *glDepthTest*).
   - Setzen der graphischen Parameter für verdeckte Linien (z.B. *glColor*, *glLineWidth*).
   - Zeichnen des Objektes in Liniendarstellung (*glPolygonMode*).

2. Durchlauf:
   - Sperrung des Color-Buffers (*glColorMask*).
   - Zeichnen des Objektes mit gefüllten Polygonen (*glPolygonMode*).

3. Durchlauf:
   - Freigabe des Color-Buffers.
   - Setzen der graphischen Parameter für sichtbare Linien.
   - Zeichnen des Objektes in Liniendarstellung (*glPolygonMode*).

Das Objekt wird von diesem Algorithmus dreimal gezeichnet, zweimal als Linienmodell und einmal als Solidmodell, d.h. mit gefüllten Polygonen. Der Tiefenbuffer wird im 2. Durchlauf so aktualisiert, dass das Objekt korrekt bezüglich seiner Tiefenwirkung in die vorhandene Szene integriert wird. Verdeckte Linien werden dadurch ermittelt, dass sie im 3. Durchlauf, quasi durch z-Buffer „geschützt“, nicht überzeichnet werden können.

Die Darstellung der Linienmodelle muss dabei die Artefakte, die aus den unterschiedlichen Rasterisierungsverfahren entstehen, vermeiden. Dazu kann in OpenGL der sog. Polygon-Offset (*glPolygonOffset*) oder eine minimale Veränderung des Wertebereichs des Tiefenbuffers (*glDepthRange*) angewendet werden [108].

5.1.1.2 Haloed-Line-Stil

Der Haloed-Line-Stil vereinfacht die Wahrnehmung von Drahtgittermodellen dadurch, dass Linien, die hinter einer anderen Linie verlaufen, kurz vor und nach dieser Linie nicht gezeichnet werden, d.h. eine Lücke besitzen. Somit wird die Unterscheidung zwischen verdeckten und

sichtbaren Linien vereinfacht. Abbildung 26 (unten links) zeigt ein Objekt, das in diesem Stil gezeichnet wurde.

Der Renderingalgorithmus benötigt zwei Durchläufe:

1. Durchlauf:
- Sperrung des Color-Buffers.
- Freigabe des Tiefen-Buffers und Aktivierung des Tiefentests.
- Setzen der Linienstärke auf eine große Breite.
- Zeichnen des Objektes in Liniendarstellung.

2. Durchlauf:
- Rücksetzen der Linienstärke auf kleine Breite.
- Freigabe des Color-Buffers.
- Setzen des Tiefentests auf Gleichheit.
- Zeichnen des Objektes in Liniendarstellung.

Im ersten Durchlauf wird ein Tiefenbild des Objekts mit breiten Linien erzeugt. Jede der im zweiten Durchlauf gezeichneten sichtbaren Linien wird dadurch an den Berührstellen mit unsichtbaren Linien von einem breiten Rand umgeben, denn „um“ die sichtbare Linie herum ist im Tiefen-Buffer der Tiefenwert der sichtbaren Linienpixel. Somit können unsichtbare Linien nur kurz vor und nach einer sichtbaren Linie gezeichnet werden.

Diese Methode ist in Modellen dort problematisch, wo mehrere sichtbare Linien zusammentreffen, denn sie blockieren sich hinsichtlich ihrer Tiefenwerte gegenseitig. Durch die Diskretisierung des Tiefen-Buffers sind auch Grenzen bei der Linienstärke gegeben [62].

5.1.1.3 Rendering von Silhouetten

Der Umriss, d.h. die Silhouette, ist ein charakteristisches Merkmal der Geometrie eines Szenenobjektes in seiner bildhaften Darstellung [96]. Insbesondere im Gebiet des nichtphotorealistischen Renderings wurden hierzu Algorithmen entwickelt, die sich grob in objektpräzise und bildpräzise Algorithmen einteilen lassen; bei Hertzmann [49] findet sich eine Übersicht der verschiedenen Ansätze. Für ein trianguliertes Modell kann eine *Silhouettenkante* definiert werden als eine Kante, die ein Front-Facing- und ein Back-Facing-Dreieck verbindet [71]. Diese Bedingung hängt von der Kameraeinstellung und dem Zustand des Modells ab.

Der hier vorgestellte Algorithmus arbeitet rein bildpräzise und berechnet die Randpixel der sichtbaren Fläche eines Szenenobjekts. Insofern ist dieser Algorithmus kein echter Silhouetten-Renderingalgorithmus, sondern detektiert die Kontur eines Szenenobjekts im Bild. Der Renderingalgorithmus benötigt fünf Durchläufe. In den ersten vier Durchläufen wird der Stencil-Buffer aufgebaut, um im fünften Durchlauf auf der Basis der erstellten Markierung die Silhouetten-Pixel zu identifizieren [62].

1. Durchlauf:
   - Setze Stencil-Operation auf Inkrementieren (*GL_INCREMENT*).
   - Setze Stencil-Funktion auf Erhalten des Stencil-Wertes (*GL_KEEP*)
   - Sperrung des Color-Buffers.
   - Sperrung des Tiefen-Buffers.
   - Verschiebe Darstellung um +1 Pixel in x-Richtung.
   - Zeichne Objekt
2. Durchlauf:
   - Verschiebe Darstellung um -1 Pixel in x-Richtung.
   - Zeichne Objekt
3. Durchlauf:
   - Verschiebe Darstellung um +1 Pixel in y-Richtung.
   - Zeichne Objekt
4. Durchlauf:

- Verschiebe Darstellung um -1 Pixel in y-Richtung.
- Zeichne Objekt

5. Durchlauf:
   - Freigabe des Color-Buffers.
   - Setze Stencil-Operation auf No-Operation.
   - Setze Stencil-Funktion auf Test auf Gleichheit mit 2 oder 3 als Referenzwert.
   - Zeichne die Bounding-Box des Objektes mit der gewünschten Silhouetten-Farbe.

Im Bild haben alle Pixel im Inneren des Objekts einen Stencil-Wert von 4, alle Pixel außerhalb des Objektes haben einen Stencil-Wert von 0 oder 1. Die Silhouetten-Pixel besitzen einen Stencil-Wert von 2 oder 3.

Dieser Algorithmus ermittelt die Silhouetten-Pixel und zeichnet diese Pixel mit Hilfe eines Hilfsobjekts (ein das Objekt im Bildraum vollständig überdeckendes Polygon), wobei der Sten-

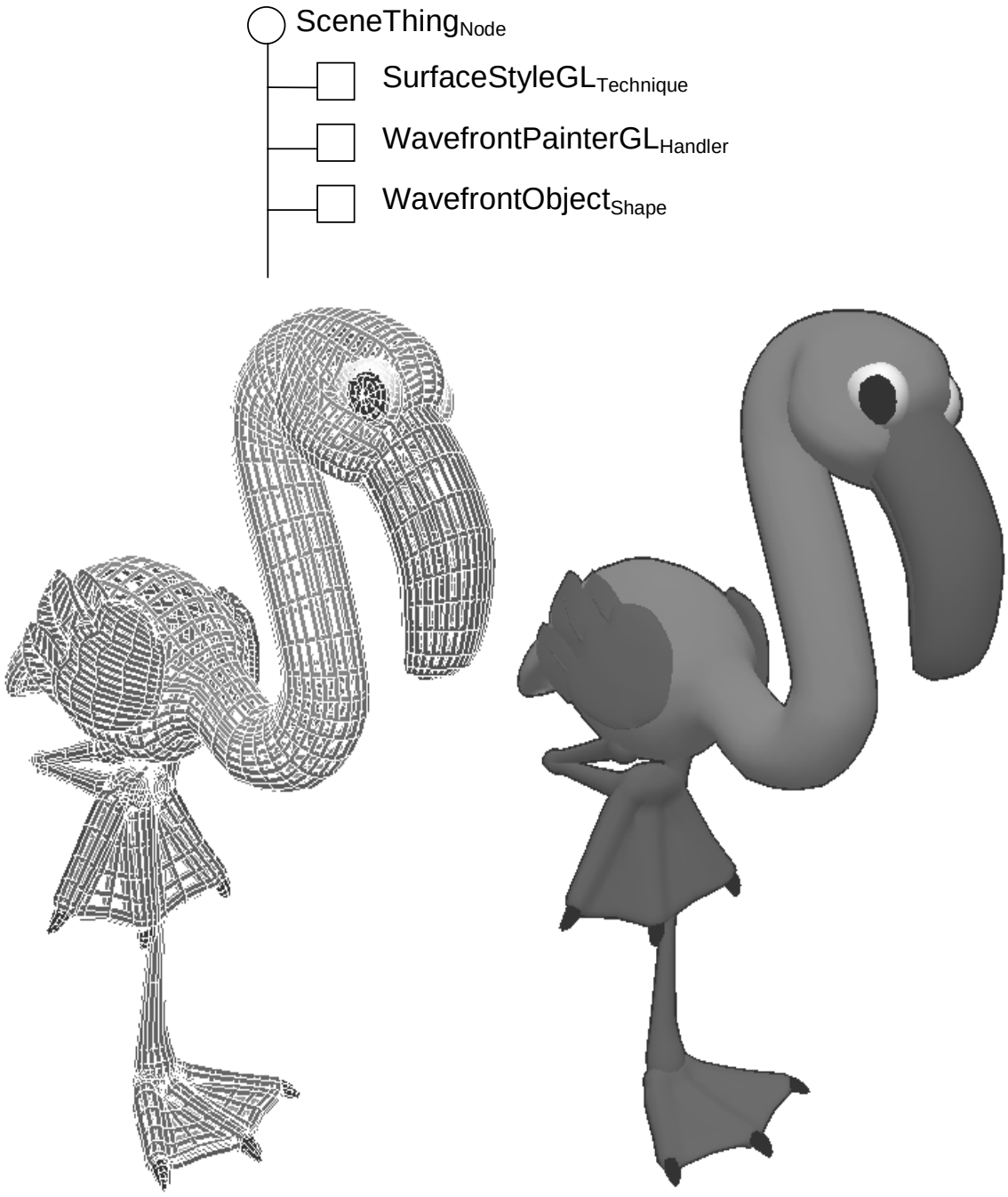

Abbildung 27. Szenengraph-Fragment zur Visualisierung eines Wavefront-Obj-Modells. Linienzeichnung im Haloe-Stil (links). Modell mit verstärkter Silhouette (rechts).

cil-Test zur Auswahl der Pixel verwendet wird; ausgewählt wird über eine Bitmaske, die das 2. Bit abprüft (für Werte von 2 oder 3). Das eigentliche Objekt kann zuvor regulär gezeichnet werden. Die Verschiebung des Fensterkoordinatensystems kann durch eine Änderung der OpenGL-Viewport-Einstellung erfolgen. Der Nachteil dieses Renderingalgorithmus ist die hohe Anzahl von Renderingdurchläufen. Der Vorteil liegt in der Pixel-präzisen Identifikation der Silhouettenpixel beliebiger Objekte.

Anzumerken ist, dass insbesondere in zukünftigen Erweiterungen des OpenGL-Standards weitere bildpräzise Algorithmen zu erwarten sind. Im Zusammenhang z.B. mit den *Register-Combiners* [72], die als Verallgemeinerung des Multitexturings eine konfigurierbare Berechnung von Fragmentfarben auf einer Per-Pixel-Basis ermöglichen, kann die Silhouette kurvenförmiger Oberflächen mit nur einem Renderingdurchlauf bestimmt werden [31]. Der Algorithmus hängt allerdings stark von der Qualität der (interpolierten) Oberflächennormalen ab und erzielt Ergebnisse hoher Qualität nur für kurvenförmige Flächen.

5.1.2 Integration der Multi-Pass-Renderingalgorithmen

Zur Integration der Multi-Pass-Renderingalgorithmen für Liniendarstellungen in das generische Renderingsystem wurde die Technik-Klasse *SurfaceStyleGL* entworfen. Die Implementierung beruht darauf, je nach gewähltem Stil die entsprechende Anzahl Renderingdurchläufe anzufordern; das Zeichnen des Objektes beinhaltet die Auswertung eines Subgraphen im generischen Szenengraphen, d.h. der Zeichenstil kann sich auf eine komplexe Teilszene erstrecken.

In Abbildung 27 ist beispielhaft ein Szenengraph dargestellt, der ein solches Technik-Objekt (*SurfaceStyleGL*) als Attribut der Szenenbeschreibung enthält. Dieser Linienstil ist OpenGL-spezifisch und kann nur von der OpenGL- Engine ausgewertet werden. Die Technik wirkt sich im Beispiel auf das im Wavefront-Format spezifizierte Shape (*WavefrontObject*) aus. Zuvor findet sich der zugehörige Painter (*WavefrontPainterGL*), der ein solches Shape mit OpenGL zeichnet. Die im Beispiel variierenden Materialeigenschaften sind in der Wavefront-Shape-Beschreibung enthalten.

5.2 Reflexion und Schatten

Im Echtzeit-Rendering stellen die Simulation von Reflexion und Schatten eine besondere Herausforderung dar: Eine exakte Berechnung z.B. auf der Grundlage physikalischer Modelle ist derzeit in Echtzeit nicht durchführbar, so dass nach geeigneten Vereinfachungen gesucht werden muss, die visuell überzeugende Ergebnisse liefern. Zur Simulation von Reflexion und Schatten haben sich eine Reihe von Verfahren herausgebildet, die durch Multi-Pass-Rendering und durch Nutzung des Stencil-Buffers in Echtzeit lauffähig sind. Zwei solche Verfahren, die ausführlich bei Kilgard [53] beschrieben sind, und ihre Integration in das generische Renderingsystem werden in diesem Abschnitt betrachtet.

Zur Simulation der Reflexion eines Szenenobjekts in der Szene wird zunächst in der Szenenbeschreibung festgelegt, welche der Szenenobjekte potentiell als Spiegel agieren. Diese Spiegel-Objekte werden in ebene Teilspiegel zerlegt, deren Ebenengleichung im weiteren Verlauf benötigt wird. Im ersten Durchlauf wird die Szene bis auf die Spiegel gezeichnet. Dann wird pro Spiegel-Objekt ein separater Durchlauf benötigt. Dabei werden jeweils die sichtbaren Pixel des Spiegels im Stencil-Buffer markiert und der Tiefenbuffer an diesen Stellen wieder initialisiert. Anschließend wird die Szene – geeignet transformiert – in die Spiegelfläche gezeichnet und zuletzt der Stencil-Buffer zurückgesetzt.

Der Renderingablauf gliedert sich wie folgt:

1. Durchlauf:
   - Lösche Stencil-Buffer und deaktiviere den Stencil-Test
   - Zeichne die Szene ohne Spiegelobjekte

2. Durchlauf (für jedes Spiegel-Objekt):

   Setzen der Stencil-Pixel für sichtbare Spiegel-Pixel
   - Aktiviere Stencil-Test: Setze Stencil-Pixel auf 1, falls Tiefentest erfolgreich.
   - Deaktiviere Color-Buffer.
   - Zeichne die Spiegel-Polygone.

   Löschen der Tiefenwerte für sichtbare Spiegel-Pixel
   - Setze Tiefenbereich auf [1,1] und Tiefentest auf ALWAYS PASS.
   - Setze Stencil-Funktion auf Gleichheitstest.
   - Zeichne die Spiegel-Polygone.
   - Setze Tiefenbereich zurück auf [0,1] und Tiefenfunktion auf GL_LESS.

   Zeichnen der gespiegelten Szene in den Spiegel
   - Aktiviere Color-Buffer.
   - Setze Clipping-Ebene zum Ausschluss der Geometrie auf der Spiegelrückseite.
   - Berechne und setze Matrix zur Spiegelung der Szene.
   - Zeichne die Szene ohne Spiegelobjekte.
   - Zeichne alle Spiegel-Objekte bis auf das aktuelle Spiegel-Objekt grau.

   Rücksetzen der Spiegel-Stencil-Pixel
   - Deaktiviere Color-Buffer.
   - Setze Stencil-Funktion auf "Löschen mit 0".
   - Zeichne die Spiegel-Polygone.
   - Aktiviere Color-Buffer.

Der 2. Durchlauf wird für jedes Spiegel-Objekt der Szene wiederholt. Auf der Basis dieses Algorithmus lassen sich mit Hilfe der Stencil-Operationen zum Inkrementieren und Dekrementierung des Stencil-Wertes und einer entsprechenden Stencil-Buffer-Bittiefe mehrfache Reflexionen modellieren; ein solches Verfahren beschreibt Diefenbach [19].

Zur Integration dieses Verfahrens wurden zwei Attribute definiert, die festlegen, welche Objekte sich spiegeln (*Reflectable*) und welche Objekte als Spiegelfläche aufzufassen sind (*Reflector*). Das Renderingverfahren wurde als Technik-Objekt (*ShadowReflection*) implementiert. Die Implementierung kombiniert das vorgestellte Verfahren zur Echtzeit-Reflexion mit einem Verfahren zur Echtzeit-Schattenberechnung. In Abbildung 28 findet sich ein Beispielszenengraph, der diese Technik einsetzt. Die Bilder zeigen die Szene ausschließlich mit Reflexion (oben) und mit Reflexion und Schatten (unten).

Das Verfahren zur Berechnung von Schatten basiert auf der von Crow [16] eingeführten Schatten-Volumen-Technik, die von Bergeron [12] auf offene und nichtebene Modelle erweitert wurde. Dazu werden in einem vorgeschalteten Durchlauf die Schattenvolumina schattenwerfender Szenenobjekte bzgl. einer Lichtquelle berechnet. Beim Rendering der Szenenobjekte wird anschließend mit Hilfe des Stencil-Buffers ermittelt, ob ein Szenenobjekt im Schattenvolumen liegt. Dazu werden im Allgemeinen zwei Durchläufe benötigt: Im ersten Durchlauf wird die Szene mit ambienter Beleuchtung gezeichnet; im zweiten Durchlauf (bzw. für jede Lichtquelle) wird die Szene mit diffuser und spekularer Beleuchtung dort gezeichnet, wo Szenenobjektfragmente nicht im Schatten liegen. Der Test, ob ein Fragment eines Szenenobjekts im Schatten liegt, erfolgt mit Hilfe des Stencil-Buffers; eine ausführliche Darstellung der Implementierung findet sich [53].

Zur Integration dieses Verfahrens wurden zwei Attribute definiert, die analog zur Reflexion festlegen, welche Objekte Schatten werfen (*ShadowCast*) und welche Objekte Schatten empfangen (*ShadowReceiver*). Da sowohl das Reflexions- als auch das Schattenverfahren eng auf der Implementierungsebene verknüpft sind, wurden sie in einer Technik-Klasse zusammengefasst. Insbesondere gilt es, die Nutzung des Stencil-Buffers abzustimmen, da beide Verfahren diesen Buffer benötigen. Die Arbeit von Diefenbach und Badler [20] skizziert, wie diese Verfahren rekursiv für Szenen mit mehreren Spiegeln angewendet werden könnten.

Die *ShadowReflection*-Technik ist zugleich ein Beispiel für die Kapselung eines komplexen Multi-Pass-Renderingverfahrens, das vollständig mit Hilfe eines Technik-Objekts implementiert und durch gezielte Attributierung der Szenenobjekte im Szenengraphen präzise eingesetzt werden kann. Die Kapselung eines komplexen Renderingverfahrens bringt langfristig den

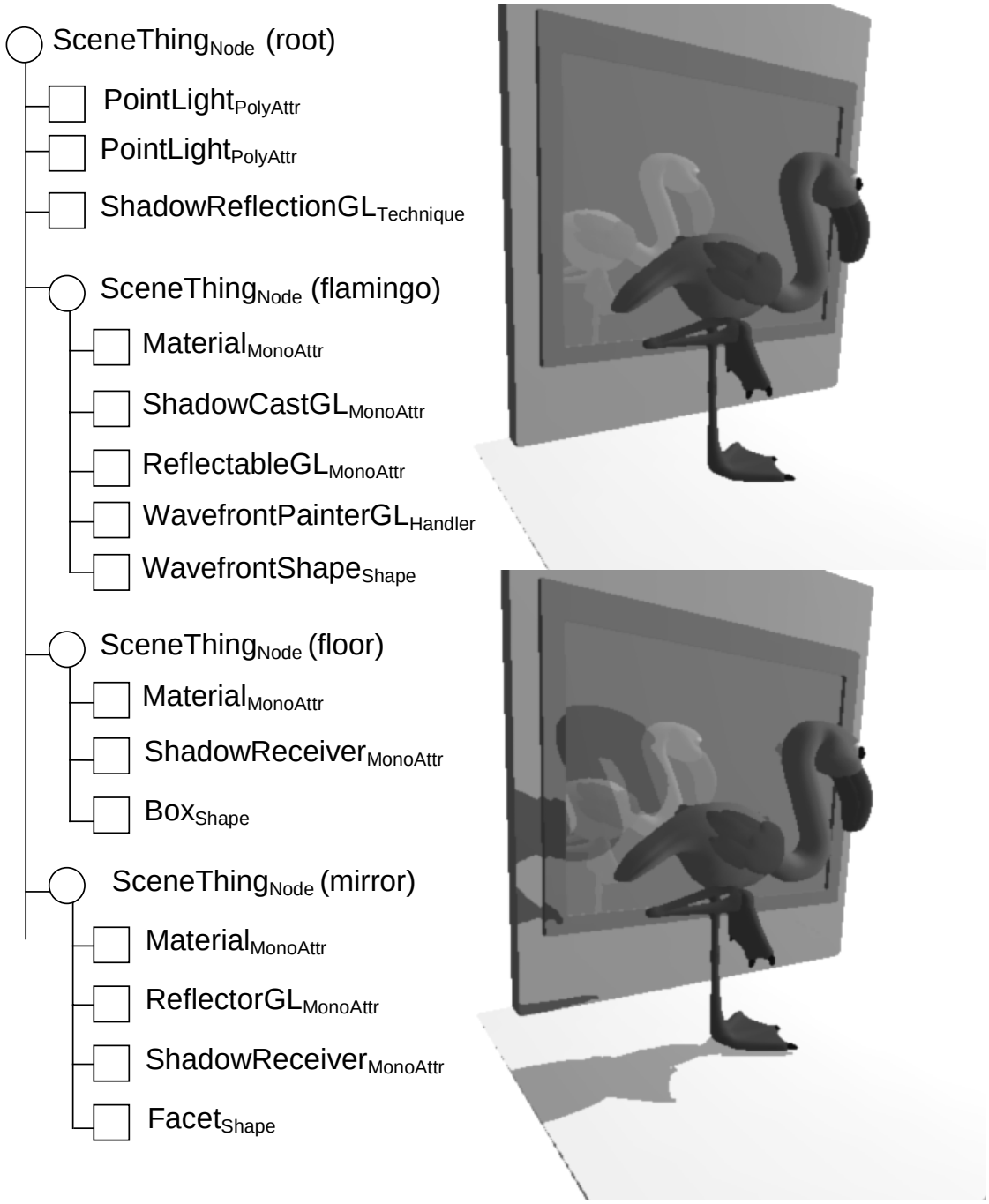

Abbildung 28. Beispiel des Reflexions- und Schattenverfahrens implementiert als Technik. Darstellung mit Reflexion (oben) und kombiniert mit Schatten (unten).

Vorteil, dass Entwicklungen im Hardware-Bereich unmittelbar Eingang finden können, ohne dass dabei die Schnittstelle und die deklarative Verwendung der Attribute im Szenengraph geändert werden müssten. So kann z.B. zur Implementierung des Schattenwurfs auch ein bildpräzises Verfahren (z.B. Shadow-Buffer [81]) eingesetzt werden, sofern die Hardware-Voraussetzungen gegeben sind.

5.3 Bewertung der Integration

Multi-Pass-Renderingverfahren können in das generische Renderingsystem als spezialisierte Technik-Klassen integriert werden. Spezialisierte Attribute zur Steuerung der Verfahren lassen sich ebenfalls leicht integrieren, so dass diese Verfahren vollständig in die Szenenbeschreibung auf der Grundlage des generischen Szenengraphen aufgenommen werden können.

Insbesondere ermöglicht es dieser Ansatz, die Fülle der Multi-Pass-Renderingverfahren, die Hardware-beschleunigt arbeiten, in einem Framework zu implementieren. So könnte z.B. bildbasiertes Constructive-Solid-Modeling (CSG) durch eine spezialisierte Technik in Verbindung mit spezialisierten CSG-fähigen Shapes realisiert werden; ein Algorithmus hierfür wird bei Wiegand [114] vorgestellt. Auch Bump-Mapping, das im Allgemeinen mehrere Rendering-Durchläufe benötigt [52], kann als Technik-Klasse gekapselt werden. Technik-Objekte und die sie steuernden Attribut-Objekte repräsentieren somit wiederverwendbare Grundbausteine für komplexe Renderingverfahren.

Sollen zwei oder mehr OpenGL-Renderingverfahren gleichzeitig verwendet werden, muss geprüft werden, inwieweit ein Ressourcen-Konflikt entsteht: Die meisten OpenGL-Renderingverfahren stützen ihre Implementierung auf die intensive Nutzung des OpenGL-Framebuffers. Benötigen zwei Techniken zum Beispiel beide den Stencil-Buffer, so ist zu klären, wie diese Techniken konfliktfrei diese Komponente des Framebuffers nutzen können. Die Sicherung des Framebuffer-Inhalts scheidet im Allgemeinen aus, da sonst das Rendering kaum in Echtzeit erfolgen kann. Derzeit gibt es für dieses Problem keine automatisierbare Lösung, denn OpenGL erlaubt keine explizite Kontrolle über den Framebuffer, d.h. Framebuffer-Inhalte können derzeit dort nicht temporär gesichert, sondern müssen im Anwendungsspeicher zwischengelagert werden.

6 FALLSTUDIE: NICHTPHOTOREALISTISCHES RENDERING

An diesem Fallbeispiel wird gezeigt, wie eine abstrakte Bildrepräsentation, die insbesondere für nichtphotorealistisches Rendering einsetzbar ist, in das generische Renderingsystem und in den generischen Szenengraph integriert werden kann. Eine abstrakte Bildrepräsentation [24] beschreibt analytisch das Bild einer Szene. Die Beschreibung enthält als Grundlage die Menge von sichtbaren, in Objektkoordinaten vorliegenden Polygonen, die mit ihnen assoziierten graphischen Attribute und schließlich die Renderingverfahren, die zur Umsetzung der Beschreibung in ein konkretes Bild benötigt werden. Zur Berechnung einer abstrakten Bildrepräsentation muss ein objektpräzises Verfahren zur Entfernung verdeckter Flächen eingesetzt werden; es bildet das Kernelement der Implementierung der abstrakten Bildrepräsentation. Eine Reihe von nichtphotorealistischen Renderingverfahren lässt sich auf der Basis dieser Bildrepräsentation entwickeln, die insbesondere von der analytischen Beschreibung und den darauf aufsetzenden Operationen profitieren. Diese Verfahren können darüber hinaus individuell für jedes einzelne Szenenobjekt zur Herstellung eines konkreten Bildes ausgewählt werden, da die Semantik der Szenenobjekte in der abstrakten Bildrepräsentation erhalten wird. Die abstrakte Bildrepräsentation kann nahtlos in das generische Renderingsystem integriert werden. Hierdurch eröffnen sich insbesondere Möglichkeiten zur Konstruktion hybrider Renderingsysteme, die nichtphotorealistisches Rendering und konventionelle Renderingverfahren zur Bildsynthese verwenden.

6.1 Einordnung des Nichtphotorealistischen Renderings

Nichtphotorealistische Computergraphik gewinnt als junge Disziplin innerhalb der Computergraphik immer mehr an Bedeutung aufgrund der weiten Anwendungsfelder, die von interaktiven Illustrationen bis zur Cartoon-Produktion reichen. *Nichtphotorealistisches Rendering* (NPR) beschreibt alle jene Verfahren, die zu computergraphischen Bildern führen, die im weitesten Sinne künstlerisch gestaltet und in ihrer Herstellung handgezeichnet erscheinen [56]. Hier zeigt sich gerade die Schwierigkeit des Begriffs, der solche Renderingverfahren beschreibt, die computergraphische Bilder erzeugen, die nicht als Fotographie einer virtuellen Wirklichkeit erscheinen sollen. Vielmehr sind diese Bilder durch den Einsatz von Zufälligkeit, Ungenauigkeit, Mehrdeutigkeit charakterisiert im Gegensatz zu realistischen Aufnahmen, die

geometrisch-präzise und im Bild klar sind. Wir könnten insofern wie Strothotte und Schlechtweg [97] auch von *nichtrealistischem Rendering* sprechen.

Eine Reihe von Algorithmen und Techniken wurden entwickelt, die sich in ihrem Aufbau grundlegend von denen aus den Bereichen des photorealistischen Renderings und Echtzeit-Renderings unterscheiden. Insofern stellt sich Frage, wie nichtphotorealistisches Rendering (NPR) in das generische Renderingsystem eingefügt werden kann, da es technisch-strukturell grundlegend verschieden zu den anderen Renderingverfahren ist.

Aus softwaretechnischer Sicht kann zunächst festgestellt werden, dass NPR-Verfahren eine andere Rendering-Pipeline besitzen als etwa Verfahren aus dem Echtzeit-Rendering: Die Berechnung nichtsichtbarer Flächen, um nur ein Beispiel zu nennen, muss bei einigen NPR-Verfahren objektpräzise erfolgen. Andere bildpräzise NPR-Verfahren müssen Bilder, die spezielle Informationen wie z.B. Tiefenwerte oder Oberflächennormalen enthalten, vorberechnen und kombinieren diese mit der eigentlichen Darstellung durch Bildoperationen zum Ergebnisbild. In dem hier vorgestellten Ansatz für eine abstrakte Bildrepräsentation bildet jedoch die objektpräzise Darstellung aller sichtbaren Polygone den Ausgangspunkt. Auf der Basis dieser Darstellung operieren in einem nachgeschalteten Schritt NPR-Verfahren, um ein konkretes Bild mit Hilfe von NPR-Techniken, wie z.B. Liniendarstellungen, zu synthetisieren.

Die Integration bildpräziser Verfahren des Nichtphotorealismus, insbesondere unter Berücksichtigung der Forderung nach einer Echtzeitdarstellung, stellt ein ebenfalls noch wenig bearbeitetes Forschungsgebiet dar. Es wird jedoch in dieser Arbeit nicht vertieft, da aus der Sicht der Software-Architektur computergraphischer Systeme die Implementierung dieser Verfahren sich auf die Konstrukte des Echtzeit-Renderings abstützt und – soweit vermutet werden kann – intensiv Multi-Pass-Rendering benötigt. Die Handhabung des Multi-Pass-Rendering wurde im vorangegangenen Kapitel dieser Arbeit beschrieben.

6.2 Konzepte der abstrakten Bildrepräsentation

Eine *abstrakte Bildrepräsentation* besteht aus

- einer Sammlung von analytisch spezifizierten *Image-Patches*, die in 2D-Sichtebenenkoordinaten und in 3D-Objektkoordinaten gegeben sind,
- einer Sammlung von mit ihnen assoziierten *Image-Patch-Attributen*, die graphische Parameter des nichtphotorealistischen Renderings repräsentieren, und
- einer Sammlung von mit den Image-Patches assoziierten *Image-Patch-Renderern*, die die nichtphotorealistisch-bildgebenden Verfahren implementieren, dabei auf den Image-Patches operieren und von den Image-Patch-Attributen gesteuert werden.

Ein *Image-Patch* repräsentiert ein in der Sichtebene liegendes beliebig geformtes Polygon, das einen Teil der sichtbaren Fläche eines Szenenobjektes wiedergibt. Die Image-Patches repräsentieren solche und nur solche Teile der projizierten Szenengeometrie, die tatsächlich sichtbar sind. Transparente Szenenobjekte bedürfen einer gesonderten Behandlung, da ihre Image-Patches im Gegensatz zu den Image-Patches der nichttransparenten Szenenobjekte sich mit anderen Image-Patches überlappen dürfen. Die Image-Patches aller transparenten Szenenobjekte repräsentieren eine separate Schicht in der abstrakten Bildrepräsentation.

Zur Berechnung der Image-Patches ist ein objektpräziser Hidden-Surface-Removal-Algorithmus [33] notwendig. In unserem Ansatz wird eine modifizierte Version des Weiler-Atherton-Algorithmus [111] eingesetzt. Der Algorithmus wurde so erweitert, dass er korrekt transparente Polygone berücksichtigt, 3D-Koordinaten der geclippten Polygone bereitstellt und

die Kanten der Image-Patches durch ein Postprocessing klassifiziert. Die Image-Patches verfügen über topologische Information (z.B. Nachbarschaftspolygone). Sie erhalten auch die Objektsemantik aus dem zugrundeliegenden Szenengraphen.

Nichtphotorealistische Renderingverfahren werden in Form von Image-Patch-Renderern als Operatoren auf Image-Patches implementiert. Zu jedem Image-Patch kann explizit festgelegt werden, von welchem Image-Patch-Renderer es bearbeitet werden soll, d.h. ein NPR-Verfahren kann auf Szenenobjektbasis erklärt werden. Einem Verfahren ist auch freigestellt, im Bildraum oder im Objektraum zu operieren; die notwendigen Informationen, die beide Varianten von Verfahren benötigen, werden von der abstrakten Bildrepräsentation bereitgestellt. Auf diese Weise lassen sich zu einer abstrakten Bildrepräsentation unterschiedliche konkrete Bilder synthetisieren. Mit unterschiedlichen NPR-Verfahren kann ohne Neuberechnung der abstrakten Bildrepräsentation z.B. interaktiv ein konkretes Bild entworfen werden.

Die abstrakte Bildrepräsentation kann vollständig in den Framework des generischen Renderingsystems integriert werden. Insbesondere lassen NPR-Attribute und NPR-Image-Patch-Renderer nahtlos in den generische Szenengraph aufnehmen. Auch das Rendering der Image-Patches kann mit Hilfe eines automatisiert hergestellten Szenengraphen verwirklicht werden.

6.3 Entwicklung von NPR-Verfahren

Die Idee zur abstrakten Bildrepräsentation ist eng verwandt mit Haeberlis Paint-by-Number-Ansatz [41], in dem ein Bild als geordnete Sammlung von Pinselstrichen kodiert wird. 3D-Szeneninformation kann darin auch verwendet werden, z.B. Oberflächennormalen, Oberflächenfarbe, Tiefe und Schattierung. Das Prinzip, ein konkretes Bild auf der Basis einer abstrakten Repräsentation durch Anwendung von NPR-Renderingverfahren zu synthetisieren, bildet auch die Grundlage für die hier vorgestellte abstrakte Bildrepräsentation. Im Unterschied zu Haeberlis Ansatz werden jedoch zunächst alle sichtbaren Oberflächen einer Szene explizit berechnet und klassifiziert (z.B. zur Unterstützung von Linienzeichnungen), um erst anschließend mit möglicherweise unterschiedlichen NPR-Verfahren ausgewertet zu werden. Außerdem wurde dieses Verfahren vollständig in eine Szenengraphen-Darstellung integriert.

Das automatisierte Rendering von Pen-and-Ink-Illustrationen von Winkenbach und Salesin [115] basiert auf einem BSP-Baum [33] als Szenendatenstruktur, die zum schnellen Clipping eingesetzt wird. Sichtbare Polygone in der Sichtebene dienen als Grundlage für das Zeichnen mit Pinselstrichen. Bedingt durch den BSP-Baum werden die sichtbaren Flächen nicht explizit berechnet; eine Ordnung ist definiert, die, wenn sie auf die Polygone angewendet wird, zu einer korrekten Darstellung der sichtbaren Flächen führt. Die Überlagerung der Flächen stellt jedoch ein Problem für einige NPR-Verfahren dar. Dieser Ansatz unterscheidet darüber hinaus nicht zwischen transparenten und nichttransparenten Polygonen.

Eine Reihe von NPR-Verfahren befasst sich mit Liniendarstellungen. Markosian et al. [60] verwenden Appells Hidden-Line-Algorithmus [4] und konzentrieren sich auf die Identifikation und das Rendering von Silhouetten im Rahmen eines Echtzeit-Renderingverfahrens. In Strothotte [96] wird ein Liniendarstellungsverfahren für illustrative Zwecke auf der Basis analytischen Entfernens verdeckter Linien vorgestellt. Die abstrakte Bildrepräsentation unterstützt und erleichtert die Implementierung von Liniendarstellungen mit Hilfe eines allgemein gehaltenen Modells der sichtbaren Flächen einer Szene.

Eine Reihe von NPR-Verfahren operiert direkt im Bildraum. Salisbury et al. [87] haben ein Konzept für Pen-and-Ink-Illustrationen auf der Grundlage von orientierten Texturen und Graustufenbildern entwickelt. Saito und Takahashi [86] entwickelten NPR-Verfahren, die mit Hilfe

von sog. G-Buffern arbeiten. Ein G-Buffer stellt eine konzeptionelle Erweiterung des Framebuffers dar, der beliebige Informationen, z.B. Oberflächennormalen, enthält. Bildpräzise NPR-Verfahren sind nicht Hauptgegenstand der abstrakten Bildrepräsentation, denn der Ausgangspunkt, die sichtbaren Flächen, werden hier objektpräzise berechnet.

Weitere NPR-Verfahren befassen sich mit physikalisch-basierter Simulation von Zeichenprozessen, z.B. Wasserfarben-Zeichnungen (Curtis et al. [17]). Gemeinsam ist diesen Verfahren, dass sie ein Bild nicht in Form von abstrakt spezifizierten Bestandteilen benötigen, sondern unmittelbar auf einen zuvor berechneten Bildmaterial operieren. Insbesondere existiert keine Verbindung zwischen Szenenrepräsentation und Bildrepräsentation. Anzumerken ist, dass ungeachtet der unterschiedlichen Voraussetzungen der einzelnen Verfahren diese theoretisch als spezielle Image-Patch-Renderer in die abstrakte Bildrepräsentation integriert werden könnten. Allerdings ist der Berechnungsaufwand im Vergleich zu direkt operierenden Verfahren zu hoch.

6.4 Aufbau der abstrakten Bildrepräsentation

Die Art und Weise, in der Renderingkomponenten in der abstrakten Bildrepräsentation ausgewertet werden, unterscheidet sich grundlegend zwischen Photorealismus-, Nichtphotorealismus- und Echtzeit-Renderingverfahren. Allgemein kann gesagt werden, dass die Rendering-Pipeline von Photorealismus- und Echtzeit-Verfahren sich an den Anforderungen des zugrundeliegenden Beleuchtungsmodells und des Sichtbarkeitsalgorithmus orientieren. Nichtphotorealistische Verfahren hingegen stellen im Allgemeinen den Prozess des Zeichnens in den Mittelpunkt ihrer Überlegungen; das Beleuchtungsmodell bildet einen von vielen Aspekten, die bei der Bildherstellung berücksichtigt werden können. Dieser Abschnitt erläutert die Rendering-Pipeline und die Renderingkomponenten der abstrakten Bildrepräsentation.

6.4.1 Auswertungsstufen

Die Rendering-Pipeline der abstrakten Bildrepräsentation (siehe Abbildung 29) unterscheidet drei Auswertungsstufen:

- *Preprocessing*: Eine hierarchische Szenenbeschreibung in Form eines generischen Szenengraphen wird zunächst ausgewertet, indem alle mehrfach referenzierten Knoten und Subgraphen zu einer flachen Sammlung von attributierten Szenenobjekten, gegeben im Weltkoordinatensystem, aufgelöst werden. Die so erhaltene Zwischenrepräsentation ist sichtunabhängig und kann partiell aktualisiert werden, falls sich die Szenenbeschreibung ändert. Das Preprocessing erfolgt, um einen direkten Zugriff auf alle Szenenobjekte und den mit ihnen assoziierten NPR-Attributen zu gewährleisten, ohne dabei den Szenengraphen traversieren zu müssen. Außerdem werden im Rahmen des Preprocessings alle nichtpolygonalen Szenenobjekte durch polygonale Darstellungen approximiert, damit sie direkt vom Hidden-Surface-Algorithmus bearbeitet werden können.
- *Sichtbarkeitsauswertung*. In dieser Stufe wird die vorberechnete Zwischenrepräsentation der Szene in eine abstrakte Bildrepräsentation umgewandelt. Diese Umwandlung umfasst das Clippen gegen das Sichtvolumen, Back-face Culling für solide, nichttransparente Objekte, Occlusion-Culling und letztlich die eigentliche Berechnung der sichtbaren Flächen. Hierzu wird eine modifizierte Version des Weiler-Atherton-Algorithmus zur objektpräzisen Berechnung von sichtbaren Flächen in 3D auf der Basis von Clipping-Operationen eingesetzt.

- *Nichtphotorealistisches Rendering*: Die Image-Patches, die während der Sichtbarkeitsauswertung ermittelt wurden, werden in dieser Stufe zur Synthetisierung konkreter Bilder herangezogen. Grundsätzlich können dabei 2D-Renderingverfahren genutzt werden, denn die Sichtbarkeitsberechnung wurde bereits durchgeführt. Darüber hinaus sind die Weltkoordinaten der sichtbaren Flächen verfügbar, so dass die Image-Patches z.B. in eine 3D-Szene integriert werden können. Auf die Image-Patches lassen sich sowohl objektpräzise als auch bildpräzise NPR-Verfahren anwenden.

6.4.2 Repräsentationsstufen der abstrakten Bildrepräsentation

Die Rendering-Pipeline der abstrakten Bildrepräsentation unterscheidet drei Repräsentationen der Szeneninformation.

- *Szenengraphrepräsentation*: Der generische Szenengraph besteht aus strukturellen Komponenten, den Szenengraphknoten, und Inhaltskomponenten, hauptsächlich den Shapes und

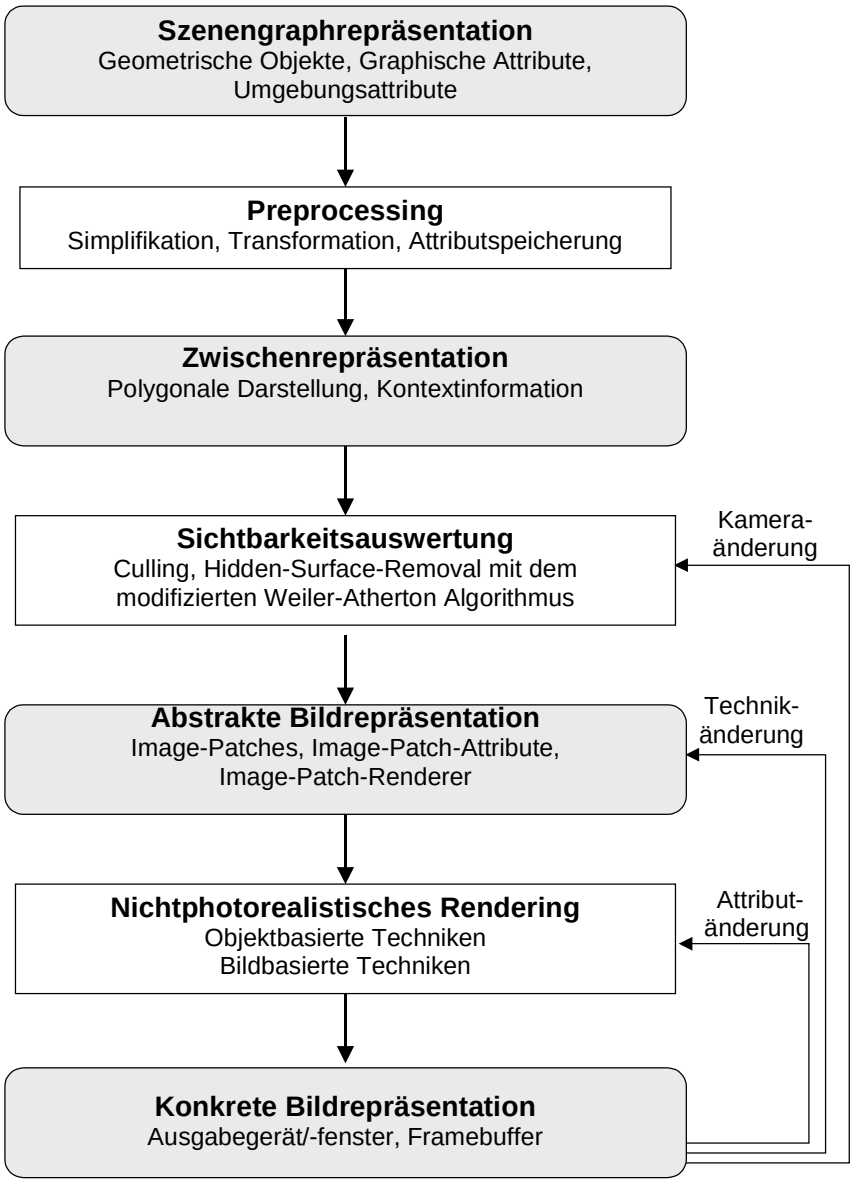

Abbildung 29. Rendering-Pipeline der abstrakten Bildrepräsentation.

Attributen. Die Attributierung, einschließlich der Spezifikation geometrischer Transformationen, erfolgt mittels hierarchischer Szenenmodellierung.

- *Zwischenrepräsentation:* Ein Zwischenobjekt wird für jedes Szenenobjekt generiert. Ein Zwischenobjekt repräsentiert das Szenenobjekt als Polygon bzw. als Polyeder. Ein Zwischenobjekt ist in Modellkoordinaten gegeben. Zusätzlich wird die Modelltransformationsmatrix gespeichert, die das Zwischenobjekt in Weltkoordinaten überführt. Außerdem werden für jedes Zwischenobjekt die aktuelle Projektions- und Orientierungsmatrix der Kamera sowie alle NPR-Attribute festgehalten. Ein Zwischenobjekt kann als Tupel von Polyeder und Kontextinformation aufgefasst werden. Um die Speicheranforderung dieser Repräsentation zu vermindern, werden sowohl Polyeder als auch Kontextinformationen als *shared objects* [98] modelliert, die mehrfach referenziert sein dürfen.
- *Abstrakte Bildrepräsentation:* Die abstrakte Bildrepräsentation ergibt sich durch die Projektion der Zwischenobjekte auf eine Sichtebene und der Berechnung aller sichtbaren Teilflächen. Die Repräsentation besteht aus Image-Patches, Image-Patch-Attributen und Image-Patch-Renderern. Das zugehörige Szenenobjekt wird als *Vorgänger* bezeichnet und kann über die Kontextinformation zu einem Zwischenobjekt erfragt werden. Das Rendering der Image-Patches wird von den mit ihnen assoziierten *Image-Patch-Attributen* (z.B. Linienstil, Linienstärke, Farbe) kontrolliert und durch *Image-Patch-Renderer* (z.B. Linienrendering) implementiert.

6.5 Komponenten der abstrakten Bildrepräsentation

Eine abstrakte Bildrepräsentation ist durch eine Sammlung von Image-Patches, Image-Patch-Renderern und Image-Patch-Attributen spezifiziert, die mit einer Szenenbeschreibung und der aus ihr abgeleiteten Zwischenrepräsentation assoziiert sind. Ein *konkretes Bild* kann durch Anwendung der Image-Patch-Renderer auf die Image-Patches erzeugt werden (s. Abbildung 30).

6.5.1 Image-Patches

Ein Image-Patch ist ein in der Sichtebene liegendes Polygon, das nicht notwendig konvex ist. Es entspricht einem Teil oder der Gesamtheit der sichtbaren Oberfläche eines Szenenobjektes. Ein Image-Patch kann bezüglich der Sichtebene als zweidimensionales geometrisches Objekt aufgefasst werden. Ein Image-Patch einer nichttransparenten Oberfläche besitzt keine Überlappung mit anderen Image-Patches einer Szene; ein Image-Patch resultiert aus dem Objekt der Zwischenrepräsentation nach dem vollständigen Clippen gegen alle anderen Polygone der Szene. Ein Image-Patch einer transparenten Oberfläche kann beliebig andere Image-Patches von transparenten und nichttransparenten Oberflächen überlappen. Die Image-Patches der nichttransparenten und transparenten Oberflächen bilden zwei Schichten von Image-Patches.

Ein Image-Patch ist ein konvexes oder konkaves Polygon, das im Inneren keine Löcher aufweist; dies wird durch den Weiler-Atherton-Algorithmus, der zur Berechnung eingesetzt wird, sichergestellt. Die Eckkoordinaten sind in 3D-Weltkoordinaten des Szenenobjekts spezifiziert. Durch die Transformation der Eckkoordinaten in Kamerakoordinaten erhalten wir die Polygone in normalisierten Fensterkoordinaten. Die Transformationsmatrizen sind in der jedem Image-Patch zugeordneten Kontextinformation erhalten; die Kontextinformation ist über die Zwischenrepräsentation zugreifbar.

6.5.1.1 Nachbar- und Geschwister-Image-Patches

Ein Image-Patch einer abstrakten Bildrepräsentation verfügt über Informationen über die angrenzenden Image-Patches, den *Nachbar-Image-Patches*. Ein Nachbar-Image-Patch kann demselben oder einem anderen Szenenobjekt zugehörig sein. Die Nachbarschaft bezieht sich stets auf die Sichtebene und wird in einem Postprocessing-Schritt nach der Sichtbarkeitsermittlung berechnet.

Ein Image-Patch einer abstrakten Bildrepräsentation kennt vollständig die Liste der Image-Patches, die sein Szenenobjekt in der Sichtebene wiedergeben. Alle Image-Patches eines Szenenobjektes in einer abstrakten Bildrepräsentation werden als *Geschwister-Image-Patches* bezeichnet; sie beschreiben die gesamte sichtbare Oberfläche eines Szenenobjektes.

6.5.1.2 Kantenklassifikation

Eine Reihe von nichtphotorealistischen Renderingverfahren benötigt explizit Informationen über die Kanten der darzustellenden Szenenobjekte. Kanten sind wesentliche Elemente von nichtphotorealistisch gerenderten Bildern. In der abstrakten Bildrepräsentation werden die Kanten für jeweils alle Image-Patches eines Szenenobjektes klassifiziert.

Die Klassifikation betrachtet dabei immer die Kanten einer Geschwistergruppe von Image-Patches; diese Gruppe kann direkt ermittelt werden, weil jeder Image-Patch sein Zwischenobjekt, dem er entstammt, kennt.

- *Randkanten*: Eine Randkante *(boundary edge)* bezeichnet Trennlinien zwischen den Bestandteilen eines Szenenobjektes. Diese Trennlinien können im Szenenobjekt enthaltenen

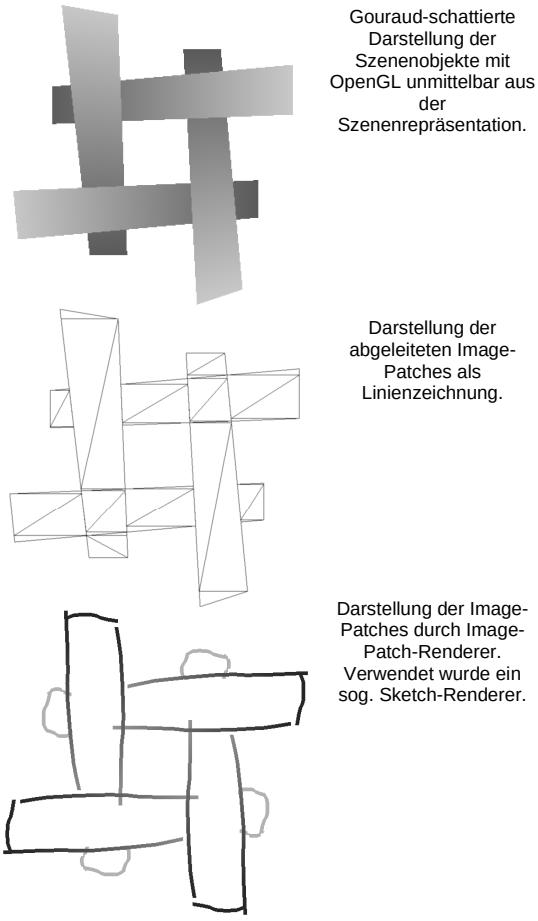

Abbildung 30. Szenendarstellung mit OpenGL (oben). Abstrakte Bildrepräsentation der Szene (Mitte). Umsetzung in ein konkretes Bild mit einer Linienzeichentechnik (unten).

Polygonkanten entsprechen oder infolge des Clippings eingefügt worden sein. Darüber hinaus kann ein Szenenobjekt, z.B. ein Polyeder, selbst festlegen, ob seine Modellkanten bzgl. der Schattierung als „harte“ oder „weiche“ Kanten aufzufassen sind.

- *Silhouettenkanten*: Eine Silhouettenkante (*silhouette edge*) repräsentiert eine äußere Grenzlinie des Szenenobjektes; es grenzt die Fläche des Szenenobjektes im Bild gegen die Flächen anderer Szenenobjekte im Bild sowie den Hintergrund ab. Zur Berechnung der Silhouettenkanten werden die Image-Patches der Geschwistergruppe vereinigt. Die äußeren Kanten des resultierenden, möglicherweise komplexen Polygons entsprechen den Silhouettenkanten des zugehörigen Szenenobjekts. Die Nachbar-Image-Patches, die an einer Silhouettenkante hängen, gehören niemals zum gleichen Szenenobjekt. Silhouettenkanten sind immer Randkanten.
- *Schnittkanten*: Eine Schnittkante (*cut edge*) repräsentiert eine Trennlinie in der polygonalen Darstellung eines Szenenobjektes, die infolge des Clippings entstanden ist. Schnittkanten sind all die Silhouettenkanten, die keine Entsprechung in der Zwischenrepräsentation besitzen. Schnittkanten sind immer Silhouettenkanten und damit auch Randkanten.
- *Innere Kanten*: Eine innere Kante (*inner edge*) bezeichnet eine durch Polygontriangulierung entstandene Kante.

Gegeben sei ein Szenenobjekt S und das daraus abgeleitete Zwischenobjekt S'. Für die Geschwistergruppe der Image-Patches I_i: intermediate(I_i)=S' gilt:

$$Randkanten_{S'} \supset Silhouettenkanten_{S'} \supset Schnittkanten_{S'}$$

$$Kanten_S := Randkanten_{S'} \cup innere\ Kanten_{S'}$$

Abbildung 31 zeigt die Kantenklassifikation eines Szenenobjektes S_1. Die Szene enthält weiter ein Szenenobjekt S_2. Die zugehörigen Zwischenobjekte bestehen im Fall von S_1 aus zwei Facetten und im Fall von S_2 aus einer Facette. In der abstrakten Bildrepräsentation verdeckt S_2 teilweise S_1. Die Geschwistergruppe der Image-Patches zu S_1 umfasst I_1, I_3 und I_4; die Geschwistergruppe zu S_2 besteht aus nur einem Image-Patch I_2.

6.5.1.3 Eckdaten

Zu den Daten, die in einer polygonalen Darstellung jeder Ecke zugeordnet werden können, zählen Eckkoordinaten, Ecknormale, Ecktexturkoordinaten und Eckfarbe. Diese Daten stehen auch den nichtphotorealistischen Renderingverfahren zur Verfügung und können von ihnen als graphische Parameter berücksichtigt werden.

Für Kanten, die sich aus dem Szenenobjekt (bzw. dessen Zwischenrepräsentation) unmittelbar in der abstrakten Bildrepräsentation wiederfinden, stehen diese Daten direkt zur Verfügung. Für Schnittkanten müssen sie durch Interpolation auf der Basis einer baryzentrischen Darstellung der Schnittpunkte berechnet werden.

6.5.1.4 Semantikerhaltung

Die Möglichkeit, in der abstrakten Bildrepräsentation auf die ursprüngliche Szenenbeschreibung zuzugreifen und damit vollständig die Semantik eines Bildbestandteils zu erfragen, stellt eine wesentliche Eigenschaft des hier vorgestellten Ansatzes dar. Nichtphotorealistische Renderingverfahren können davon profitieren, indem sie ihre Arbeitsweise auf die Semantik der dargestellten Objekte abstimmen.

Die folgende Aufstellung fasst die insgesamt in der abstrakten Bildrepräsentation zur Verfügung stehenden Informationen zusammen:

- Image-Patch: 2D-Bild- und 3D-Objektkoordinaten, Nachbar-Image-Patches
- Geschwistergruppe von Image-Patches: Kantenklassifikation.
- Zwischenobjekt: graphische Attribute und Kontextinformation, z.B. Kameraeinstellung.
- Szenenobjekt: Szenengraphknoten, Shapes und Attribute.

6.5.2 Rendering von Image-Patches

Das Rendering von Image-Patches erfolgt auf der Basis von Image-Patch-Renderern und Image-Patch-Attributen. Diese Objekte können im generischen Renderingsystem als spezielle NPR-Techniken und NPR-Attribute implementiert werden.

Sowohl Image-Patch-Renderer als auch Image-Patch-Attribute können als Renderingobjekte des generischen Renderingsystems in einem generischen Szenengraphen spezifiziert werden. Wird ein derart ausgestatteter Szenengraph von einer speziellen Engine, die die abstrakte Bildrepräsentation erstellt, traversiert, dann werden diese Attribute und Techniken direkt mit den Image-Patches referenziert.

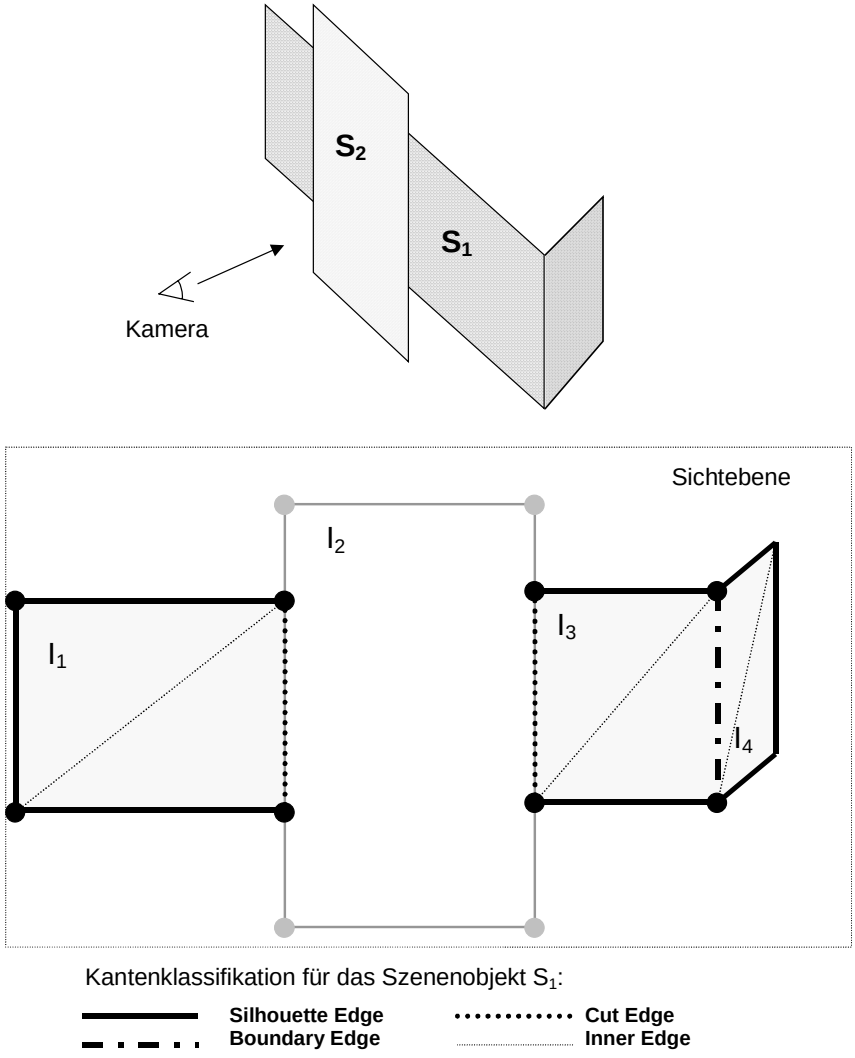

Abbildung 31. Beispiel einer Kantenklassifikation.

6.5.2.1 Image-Patch-Renderer

In der abstrakten Bildrepräsentation sind die Renderingalgorithmen in sog. Image-Patch-Renderer-Objekten gekapselt. Grundsätzlich kann für jeden einzelnen Image-Patch einer Szene ein separater Image-Patch-Renderer verwendet werden. Die Auswahl der Image-Patch-Renderer erfolgt praktisch jedoch auf Szenenobjekt-Basis, d.h. ein Image-Patch-Renderer wird einem oder mehreren Szenenobjekten zugewiesen. Im Gegensatz zu bildpräzisen NPR-Verfahren (z.B. G-Buffer-Verfahren) verfügt damit die abstrakte Bildrepräsentation über einen hohen Grad an gestalterischer Flexibilität, da ein NPR-Verfahren szenenobjektsensitiv angewendet werden kann. Falls ein Image-Patch-Renderer objektunabhängig im Bildraum operiert, dann wird nur ein einziger Image-Patch-Renderer zum Rendering einer abstrakten Bildrepräsentation benötigt.

6.5.2.2 Image-Patch-Attribute

Image-Patch-Attribute beeinflussen die Arbeitsweise von Image-Patch-Renderern und stellen die graphischen Parameter für die abstrakte Bildrepräsentation dar. Image-Patch-Attribute beinhalten Standard-Attribute wie z.B. Farbe, Materialeigenschaften und Textur, aber auch sog. NPR-Attribute, die die Arbeitsweise einer Reihe von NPR-Verfahren zu steuern vermögen; NPR-Attribute sind meist abstrakte Attribute, die z.B. die kommunikativen Ziele eines Bildes beschreiben können. Als Beispiele für NPR-Attribute seien genannt:

- *Unsicherheit*: Das Unsicherheitsattribut beschreibt den Grad der Unsicherheit, der in die Darstellung kodiert werden soll. Eine Linienzeichnung kann die Unsicherheit in einer Polygon-Darstellung z.B. durch das Zeichnen verwackelter, mehrfach schief übereinanderliegender Kanten ausdrücken.
- *Wichtigkeit*: Das Wichtigkeitsattribut beschreibt den Grad der visuellen Dominanz. Gestalterisch kann dieses Attribut z.B. mit Hilfe von Farbe oder Linienstärke umgesetzt werden.
- *Hervorhebung*: Das Hervorhebungsattribut wird genutzt, um Szenenobjekte visuell zu kennzeichnen. Mit Hilfe einer Hervorhebung kann z.B. der visuelle Fokus in einer Darstellung gesteuert werden. Gestalterisch kann dieses Attribut z.B. durch die verstärkte Zeichnung der Silhouette oder durch Positionierung eines Spotlights erfolgen.

6.5.2.3 Implementierung von NPR-Verfahren

Objektpräzise Liniendarstellungen, wie sie z.B. in Strothotte [96] beschrieben werden, lassen sich unmittelbar auf der Grundlage der Image-Patches implementieren. Ein Image-Patch-Renderer kann unmittelbar die Kanten eines Image-Patches unter Berücksichtigung der Kantenklassifizierung zeichnen. Das Linienverfahren kann z.B. mehrere Linienzüge, deren Anzahl und exakte Position von dem Unsicherheitsattribut und deren Stärke von dem Wichtigkeitsattribut kontrolliert werden, verwenden. Die Tiefeninformation, die durch die 3D-Objektkoordinaten zur Verfügung steht, kann zusätzlich in die graphische Gestaltung einfließen.

Pinselzeichnungen und Strahl-basierte Zeichnungen, wie sie z.B. in Haeberli [41] beschrieben sind, können implementiert werden, indem eine Menge von Pinselstrichen für jeden Image-Patch unter Berücksichtigung der Bildauflösung erstellt und angewendet wird. Die Lage, Größe und Richtung eines Pinselstrichs können abhängig von der Image-Patch-Geometrie oder der Szenenobjektgeometrie festgelegt werden. Die Form des Pinsels lässt sich z.B. durch ein gesondertes NPR-Attribut kontrollieren.

Der *Painterly*-Renderingansatz, der von Meier [60] beschrieben wird, kann ebenfalls unmittelbar auf der Basis der abstrakten Bildrepräsentation implementiert werden: Die hier notwendigen Partikel lassen sich über Image-Patches unter Berücksichtigung der 3D-

Geometrie platzieren; Oberflächennormalen und die graphischen Parameter lassen sich über die Szenenobjekte ermitteln.

6.5.3 NPR-Bilder

Konzeptionell kann nun genau festgelegt werden, aus welchen Bestandteilen sich ein NPR-Bild der hier vorgestellten Bildrepräsentation aufbaut. Die Komponenten können in drei Schichten eingeteilt werden, die Szenengraphrepräsentation, die Zwischenrepräsentation und die abstrakte Bildrepräsentation (siehe Abbildung 32). Ein NPR-Bild wird durch einen Szenengraph spezifiziert. Durch Auswertung des Szenengraphen werden die Zwischenrepräsentation und die abstrakte Bildrepräsentation automatisch konstruiert. Die im Szenengraphen enthaltenen NPR-Attribute und NPR-Techniken übernehmen die Synthese eines nichtphotorealistisch gerenderten Bildes. Durch die Anwendung der Image-Patch-Renderer entsteht ein konkretes Bild; die abstrakte Bildrepräsentation kann – einmal berechnet – stets zur Synthese neuer Bilder genutzt werden, falls sich nur NPR-Attribute verändern. Bei einer Kameraveränderung ist die Neuberechnung der abstrakten Bildrepräsentation notwendig.

Ferner besitzen NPR-Bilder analytische Eigenschaften, die sich aus der Struktur der Image-Patches ergeben und zur Implementierung von Abfragen eingesetzt werden können:

- Abfrage von Image-Patches, die ein gegebenes Szenenobjekt im Bild repräsentieren.
- Abfrage von Nachbarschafts-Image-Patches.
- Abfrage von Image-Patches, die eine bestimmte Fläche im Bild überdecken.
- Abfrage von Image-Patches mit bestimmten Flächeninhalt, mit bestimmter Kantenzahl, mit bestimmter Neigung.

Aus einer objektorientierten Sicht kann das NPR-Bild als Klasse verstanden werden, die Methoden zur Konstruktion eines graphisch-geometrischen Sachverhalts, zu seiner nichtphotorealistischen Umsetzung und zur Untersuchung des Bildraums bereitstellt.

6.5.4 Bewertung der abstrakten Bildrepräsentation

Die Stärken der abstrakten Bildrepräsentation sind:

- *Analytische Bildbeschreibung:* Eine abstrakte Bildrepräsentation ist auflösungsunabhängig; sie kann konkrete NPR-Bilder in jeder Auflösung synthetisieren. Skalierbare Ausgaben, wie

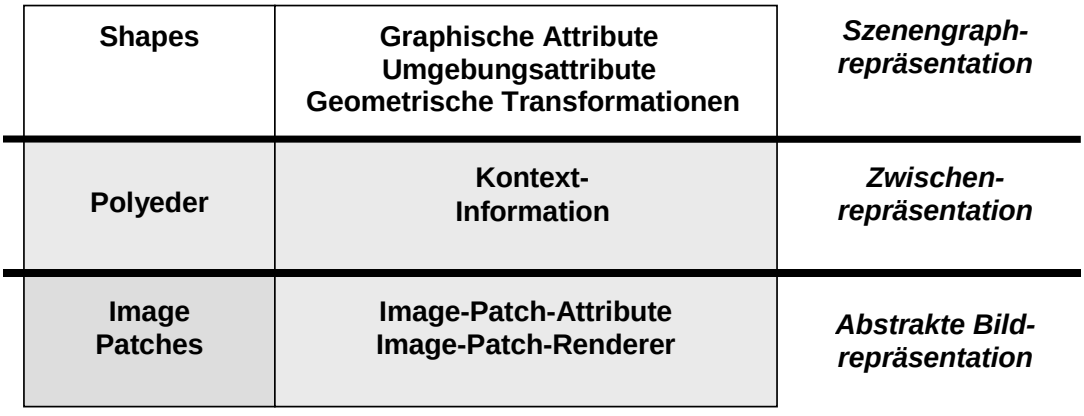

Abbildung 32. Konzeptioneller Aufbau eines NPR-Bildes.

sie z.B. im Fall von PostScript auftreten, lassen sich dadurch einfach herstellen.

- *Generische Bildbeschreibung*: Eine abstrakte Bildrepräsentation dient als Grundlage für die Synthetisierung konkreter NPR-Bilder, die durch Anwendung von Image-Patch-Renderern geschieht. Unterschiedliche NPR-Verfahren lassen sich somit auf eine Bildrepräsentation und auf ihre einzelnen Bestandteile anwenden, ohne dabei die Repräsentation neu berechnen zu müssen.
- *Kompakte Bildbeschreibung:* Werden in einem generischen Szenengraphen Image-Patch-Attribute und Image-Patch-Renderer eingefügt, so können sowohl die Zwischenrepräsentation als auch die abstrakte Bildrepräsentation automatisch berechnet werden. Ein NPR-Bild kann demzufolge mit Hilfe eines generischen Szenengraphen kompakt und vollständig beschrieben werden.
- *Semantikerhaltung:* Die abstrakte Bildrepräsentation erhält in allen Schichten die Semantik der Szenenobjekte. Abfrage-Operationen für die Semantik von Bildbereichen lassen sich dadurch implementieren.
- *Kompatibilität:* Abstrakte Bildrepräsentationen können unmittelbar aus Szenengraph-basierten Beschreibungen erstellt werden (z.B. VRML-Szenengraphen). Ihre Spezifikation ist insofern methodisch kompatibel zu einer große Klasse von existierenden Szenenbeschreibungen.

6.6 Berechnung der abstrakten Bildrepräsentation

Die Berechnung der Image-Patches stellt in Hinblick auf die Geschwindigkeit, mit der eine abstrakte Bildrepräsentation berechnet werden kann, den aufwendigsten Abschnitt dar. Als Algorithmus zur objektpräzisen Berechnung der Image-Patches setzen wir eine modifizierte Version des Hidden-Surface-Removal-Algorithmus von Weiler und Atherton [111] ein. Im Vergleich dazu sind bildpräzise Ansätze, z.B. auf der Basis von Saito und Takahashi [86] entwickelten G-Buffer, mit heutiger Hardware in Echtzeit nutzbar, besitzen allerdings nicht vergleichbare analytische Eigenschaften.

6.6.1 Weiler-Atherton HSR-Algorithmus

In seiner ursprünglichen Fassung verarbeitet der Weiler-Atherton-Algorithmus eine Eingabeliste von Polygonen und liefert eine Ausgabeliste von sichtbaren Polygonen zurück. Für den Clipping-Vorgang selektiert der Algorithmus zunächst das Polygon mit der kleinsten z-Koordinate als Clip-Polygon und clippt dagegen alle anderen Polygone. Das Ergebnis besteht aus zwei sortierten Listen, einer Liste mit allen Polygonfragmenten im Inneren des Clip-Polygons und eine Liste mit allen Fragmenten außerhalb des Clip-Polygons. Polygonfragmente im Inneren werden direkt verworfen, falls sie hinter dem Clip-Polygon liegen, denn sie sind unsichtbar. Für Polygone im Inneren, die vor dem Clip-Polygon liegen, wird das Clipping rekursiv aufgerufen; das Clipping erfolgt gegen diese Polygone und als Eingabeliste dient die Liste der inneren Polygonfragmente. Die verbleibenden inneren Polygone werden als sichtbare Polygone verbucht. Der Algorithmus fährt fort, indem ein neues Clip-Polygon aus der Liste der äußeren Polygone ausgewählt wird. Eine ausführliche Beschreibung des Algorithmus findet sich bei Foley et al. [33].

6.6.2 Modifizierter HSR-Algorithmus

Für die abstrakte Bildrepräsentation wurde der Algorithmus wie folgt modifiziert:

- *Culling:* Die Semantik der sichtbaren Polygone bleibt erhalten, da ihre Herkunft mit Hilfe der Zwischenobjekte ermittelt werden kann. Dadurch ist es möglich, objektbasiert Polygone im Vorfeld auszuschließen, um die Zahl der insgesamt zu testenden Polygone zu reduzieren. Insbesondere kann Back-Face-Culling und Occlusion-Culling auf Szenenobjekte angewandt werden. Weiter können solche Szenenobjekte ausgeschlossen werden, die vollständig unsichtbar, d.h. außerhalb des Sichtvolumens der Kamera liegen.
- *Transparente Polygone:* Ein transparentes Eingabepolygon wird nicht als Clip-Polygon eingesetzt, weil es keine Flächen (vollständig) verdecken kann. Daneben clippt der Algorithmus ein transparentes Polygon gegen das aktuelle Clip-Polygon nur, wenn das transparente Polygon hinter dem opaken Clip-Polygon liegt. Ein Beispiel für die Handhabung transparenter Polygone findet sich in Abbildung 33.
- *Kantenklassifikation.* Der Clipping-Algorithmus markiert solche Kanten eines sichtbaren Polygons, die durch einen Schnitt entstehen. Nach der Berechnung aller Image-Patches werden die Kanten während eines Postprocessings weiter in Rand-, Silhouetten- und innere Kanten klassifiziert.
- *3D-Koordinaten:* Der Algorithmus beruht auf dem Polygon-Clipping im Zweidimensionalen. Für Ecken von Kanten, die infolge des Clippings neu entstehen, werden nach dem 2D-Clipping die 3D-Objektkoordinaten erstellt. Die z-Koordinate wird mit Hilfe der Ebenengleichung berechnet. Im Anschluss müssen ggf. weitere Eckdaten durch Interpolation ermittelt werden.
- *Korrekte rekursive Berechnung innerer Polygone:* In der ursprünglichen Fassung des HSR-Algorithmus wurde geprüft, ob ein Polygon vollständig hinter dem Clip-Polygon liegt. Andernfalls wurde unterstellt, dass sich das Polygon vollständig vor dem Clip-Polygon befindet. Jedoch kann es der Fall sein, dass ein Polygon sowohl vor als auch hinter dem

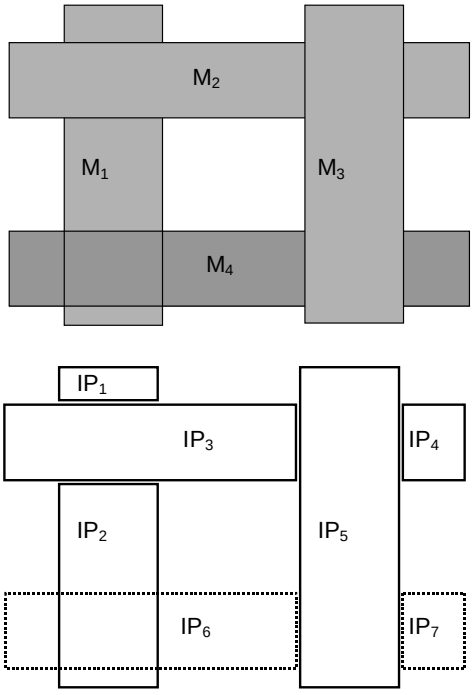

Abbildung 33. Opakes (M_{1-3}) und transparentes (M_4) Zwischenobjekt (oben). Abstrakte Bildrepräsentation: Reguläre Image-Patches IP_{1-5} und transparente Image-Patches IP_{6-7} (unten).

Clip-Polygon liegt. In diesem Fall muss der Algorithmus das Polygon weiter unterteilen.

Der Algorithmus, der in der Prozedur *buildImage* (s. unten) beschrieben ist, erwartet eine Zwischenrepräsentation *M* und liefert eine Liste von Image-Patches *IP*. Das Culling entfernt a priori unsichtbare Teile der Szenengeometrie. Danach wird eine räumliche Zugriffsstruktur aufgebaut, in der alle Polygone nach der z-Koordinate sortiert werden, die den effizienten Zugriff auf Polygone zu einer gegebenen Raumposition gewährleistet. Zum Beispiel kann ein adaptiver Quadtree eingesetzt werden. Anschließend startet das eigentliche Clipping. Die Unterteilungsstufe entfernt alle unsichtbaren Polygone und beginnt mit dem vordersten, nicht-transparenten Polygon. Sie endet, wenn alle nichttransparenten Polygone entfernt sind. Zum Schluss werden alle verbliebenen transparenten Polygone – die ggf. unterteilt wurden – in die Ausgabeliste übernommen.

```
S : Stack of Polygons

procedure buildImage( in M : IntermediateRepresentation,
                  in T_proj : Camera, out IP : List of ImagePatches)
begin
  L : Polygon Container
  { transform & decompose M into polygons}
  L ← transform ( M, T_proj )
  {remove a-priori invisible scene parts}
  L ← culling ( M, T_proj )
  {build spatial lookup structure and sort by z-coordinates}
  L ← sort (L, T_proj )
  clear S

  while L has opaque polygons  do
  begin
    clipP : Polygon ← first opaque polygon of L
    subdivide ( clipP, L, IP )
  end

  convert and add transparent polygons of L to IP
  calculate z coordinates for image patches
end
```

Im Algorithmus, der die Unterteilung vornimmt und in der unten aufgeführten Prozedur *subdivide* wiedergegeben ist, werden die Polygone aus *L* gegen das Clip-Polygon *clipP* geschnitten. Entstehende Image-Patches werden in die Ausgabeliste *IP* eingefügt. Für das Clipping wird im Allgemeinen das ursprüngliche Polygon statt des möglicherweise schon geclippten Polygons gewählt, um numerische Ungenauigkeiten, die durch das Clipping entstanden sein könnten, zu vermeiden.

Die Polygone der Inside-Liste werden auf Mehrdeutigkeiten bzgl. der Lage vor oder hinter dem Clip-Polygon untersucht. Nichtsichtbare Polygone werden direkt entfernt. Die verbleibenden Polygone der Inside-Liste werden dann rekursiv ausgewertet, wobei ein globaler Stack zur Kommunikation bereits ausgewerteter Clip-Polygone genutzt wird. Am Ende enthält die Inside-Liste alle sichtbaren Polygone im Bereich des Clip-Polygons, aus denen nun Image-Patches entstehen.

```
procedure subdivide(in clipP : Polygon,
                                  in/out L : Polygon Container,
                                  in/out IP : List of ImagePatches)
begin
  L' : Polygon Container
  { lookup polygons covering/touching clipP }
  L' ← lookUp ( L, clipP )
  { remove these polygons from L }
  L ← L \ L'

  insideList : Polygon Container
  for each p : Polygon in L' do
```

```
  begin
    if (p and clipP belong to the same polygon) continue
    clip p against ancestor of clipP
    place inside parts in insideList
    insert outside parts into L
  end

  calculate z-coordinates (if not already done) for polygons
  in insideList

  break polygons from insideList which are both behind and
  in front of clipP

  remove all polygons from insideList which are completely
  behind clipP

  for each p : Polygon in insideList do
  begin
    if ( p is not on stack S ) and
       ( p and clipP do not belong to the same polygon ) and
       ( p is not transparent ) do
    begin
      push clipP onto stack S
      subdivide ( p, insideList, IP )
      pop clipP from stack S
    end
  end

  for each polygon in insideList do
  begin
    calculate vertex data for newly introduced vertices
  end

  add insideList to IP
end
```

6.6.3 Implementierung

Die räumliche Zugriffsstruktur, die zur Ermittlung für das Clipping relevanter Polygone eingesetzt wird, entscheidet wesentlich über die Laufzeit des HSR-Algorithmus. Die Struktur kann an die Szenenkomplexität und den vorhandenen Speicher angepasst werden.

Außerdem benötigt der HSR-Algorithmus einen robusten 2D-Clipping-Algorithmus. In unserer Implementierung gelangt der Algorithmus von Vatti [106] zum Einsatz, der als Polygon-Clip-Algorithmus eine hohe numerische Stabilität besitzt. Als Nebeneffekt erhalten wir hier zugleich alle geclippten Polygone in triangulierter Form.

Ebenfalls können noch während des Clipping-Vorgangs solche Polygone entfernt werden, die benutzerdefinierte kritische Werte, z.B. eine minimale Fläche, unterschreiten. Dadurch können insbesondere kaum sichtbare Polygone von der weiteren Verarbeitung ausgeschlossen werden.

Der Clipping-Algorithmus, die Image-Patches und die Zwischenobjekte operieren auf einer einheitlichen Polygon-Datenstruktur, um sowohl den Implementierungsaufwand als auch die Speicheranforderungen und Speicherallokationen zu minimieren.

6.7 Image-Patch-Renderer

In diesem Abschnitt werden Image-Patch-Renderer vorgestellt, die auf der Grundlage der abstrakten Bildrepräsentation implementiert wurden. Vordringliches Ziel war es, zu testen, wie sich bekannte, objektpräzise NPR-Verfahren in das Konzept einfügen.

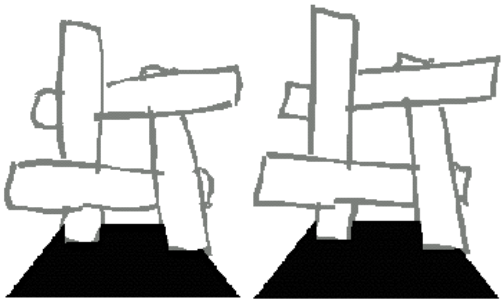

Abbildung 34. Linienzeichnung erstellt mit dem Sketch-Renderer. Unterschiedliche Wölbungen resultieren von unterschiedlichen Unterteilungen der Linien.

6.7.1 Sketch-Renderer

Der Sketch-Renderer simuliert einfache, handgefertigte Linienzeichnungen. Der Renderer operiert auf den klassifizierten Kanten einer Geschwistergruppe von Image-Patches. Folgende NPR-Attribute steuern die Arbeitsweise des Sketch-Renderers:

- *NPR-Kantenattribut.* Legt fest, welche Klasse von Kanten graphisch umgesetzt werden.
- *NPR-Unsicherheitsattribut:* Kontrolliert, mit welchem Grad an Störung (d.h. Verwackelung) die Linien gezeichnet werden.
- *NPR-Sketch-Attribut:* Legt die Zahl der Linienstriche pro Linie und die Linienstärke fest.
- *NPR-Pinselattribut.* Kontrolliert die Anzahl Pinselstriche, mit denen Linien gezeichnet werden.

Der Sketch-Renderer unterteilt jede zu zeichnende Kante in Teillinien. Ein zusätzlicher Abstand wird für die Anfangs- und Endpunkte der Teillinien berechnet. Abbildung 34 zeigt zwei Linienzeichnungen, die mit diesem Image-Patch-Renderer synthetisiert wurden. Weitere Ansätze, die sich in diesem Kontext integrieren lassen, finden sich u.a. bei [60], [88] und [96].

6.7.2 Texture-Stroke-Renderer

Eine andere Art der Linienzeichnung verwendet Texturen, um Linien zu zeichnen. Die Arbeitsweise dieses Image-Patch-Renderers ist der des Sketch-Renderers ähnlich und nutzt die gleichen NPR-Attribute. Das Zeichnen der Linien erfolgt jedoch dadurch, dass (schmale) Polygone für die Image-Patch-Kanten generiert und sodann texturiert gezeichnet werden. Dieses Verfahren gebraucht Texturierung als graphisches Primitiv [40]; bedingt durch die Unterstützung der Texturierung durch die Hardware, ist dieses Verfahren effizient auf heutiger Hardware implementierbar. Die Texturen entscheiden wesentlich über das Aussehen der Linien und lassen sich extern editieren. Die Linientextur wird über ein NPR-Stroke-Texture-Attribut eingestellt.

6.7.3 Stippling-Renderer

Der Stippling-Renderer realisiert ein alternatives Schattierungsverfahren. Die Schattierung wird bildlich mit unterschiedlich dicht positionierten Punkten dargestellt. Zu diesem Zweck werden die Image-Patches in das Kamerakoordinatensystem transformiert; für jede Ecke des Image-Patches wird die Beleuchtungsintensität (z.B. mit Hilfe eines Phong-Beleuchtungsmodells) berechnet. Für jedes Dreieck eines Image-Patches wird durch Mittelung eine Intensität kalkuliert.

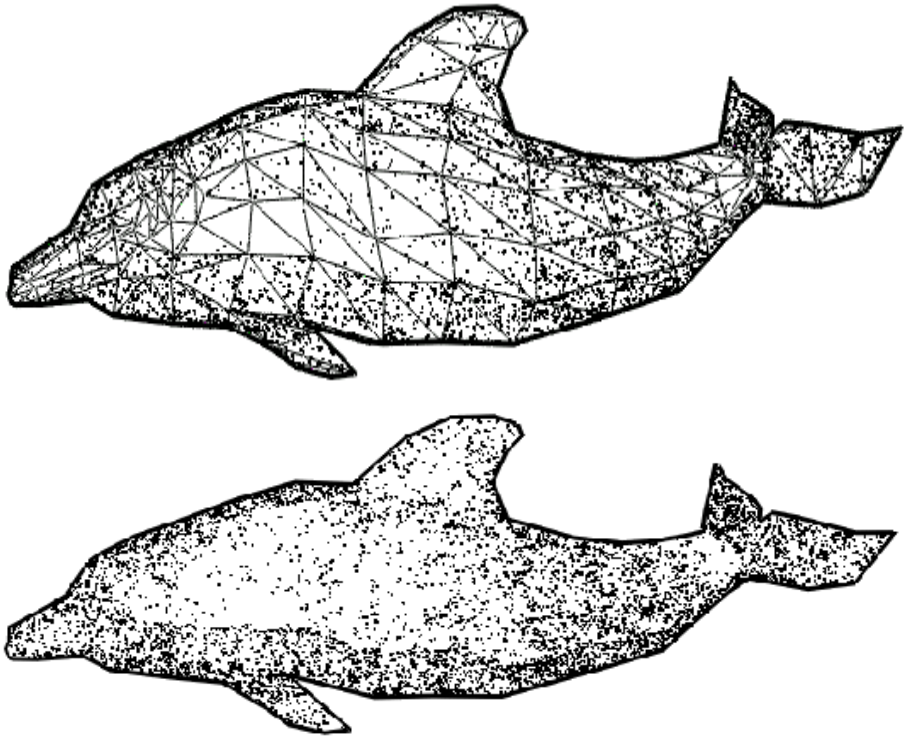

Abbildung 35. NPR-Bild mit Stippling-Technik.

Zusätzlich wird die Größe des Dreiecks ermittelt. Die Multiplikation der Intensität, der Dreiecksgröße und eines zusätzlichen Faktors, der aus dem Wichtigkeitsattribut entnommen wird, führen zu einem Wert, der bestimmt, wieviele Punkte in dem Dreieck zufällig verteilt dargestellt werden.

Dieser Image-Patch-Renderer nutzt im Gegensatz zum Sketch- und Texture-Stroke-Renderer die 3D-Information, die für die Image-Patches verfügbar ist. Abbildung 35 zeigt ein NPR-Bild, das mit Hilfe der Stippling-Technik erstellt wurde. In der oberen Abbildung sind die inneren Kanten sowie die Silhouettenkanten visualisiert. Die untere Abbildung zeigt das Objekt ausschließlich im Stippling-Stil.

6.7.4 Emphasis-Renderer

Der Emphasis-Renderer hebt Szenenobjekte in der bildlichen Darstellung durch eine Umrandung hervor. Dazu werden die Silhouettenkanten in einer Stärke und Farbe gezeichnet, die vom Wichtigkeitsattribut abhängen. Der Emphasis-Renderer lässt sich unabhängig von anderen NPR-Techniken einsetzen und kann z.B. zur visuellen Kennzeichnung markierter Objekte eingesetzt werden.

In Abbildung 36 finden sich zwei NPR-Bilder, in denen die Hervorhebung von Szenenobjekten variiert.

6.7.5 PostScript-Renderer

Der PostScript-Renderer ist ein vereinfachter Sketch-Renderer, der die Kanten eines NPR-Bildes als PostScript-Linienprimitive [1] ausgibt. Aufgrund der Natur der abstrakten Bildrepräsentation, die die Polygone der Sichtebene überlappungsfrei (für opake Polygone) berechnet, gestaltet sich die Implementierung dieses Renderers einfach.

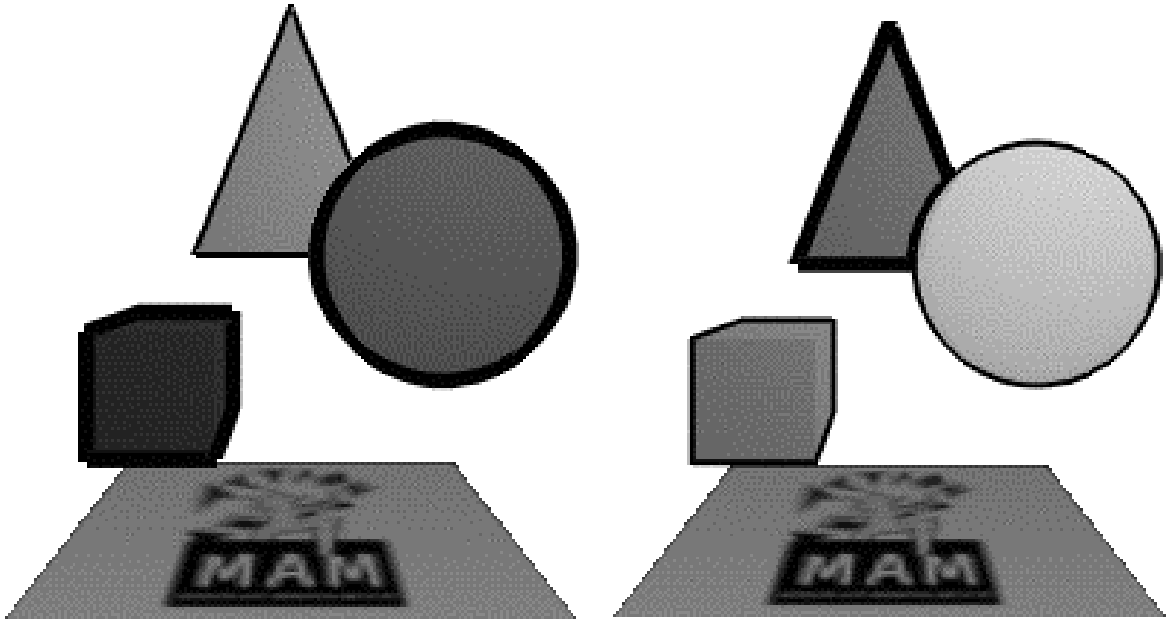

Abbildung 36. Eine Szene mit wechselnder Hervorhebung (links: Kugel, rechts: Kegel).

6.7.6 NPR-Bildergalerie

In Abbildung 37 findet sich eine Sammlung konkreter Bilder, die von einer abstrakten Bildrepräsentation mit Hilfe verschiedener Image-Patch-Renderer abgeleitet wurden. Die verwendeten Image-Patch-Renderer sind in folgender Tabelle zusammengefasst.

Bild	Technik
1	Gouraud-schattierte Ausgabe des polygonalen Szenenobjekts mit OpenGL.
2	Einfache Linienzeichnung mit dem Sketch-Renderer.
3	Linienzeichnung mit erhöhtem Unsicherheitswert und erhöhter Zahl von Linien, realisiert mit dem Sketch-Renderer.
4	Texturbasierte Linienzeichnung mit dem Texture-Stroke-Renderer.
5	Linienzeichnung mit dem Emphasis-Renderer.
6	Schattierte Darstellung mit dem Tone-Shading-Renderer.
7	Darstellung mit dem Stippling-Renderer.
8	Anwendung des Stippling-Renderers und des Sketch-Renderers auf unterschiedliche Image-Patch-Gruppen des gleichen Szenenobjekts.
9	Anwendung des Tone-Shading-Renderers und Sketch-Renderers auf unterschiedliche Image-Patch-Gruppen des gleichen Szenenobjekts.

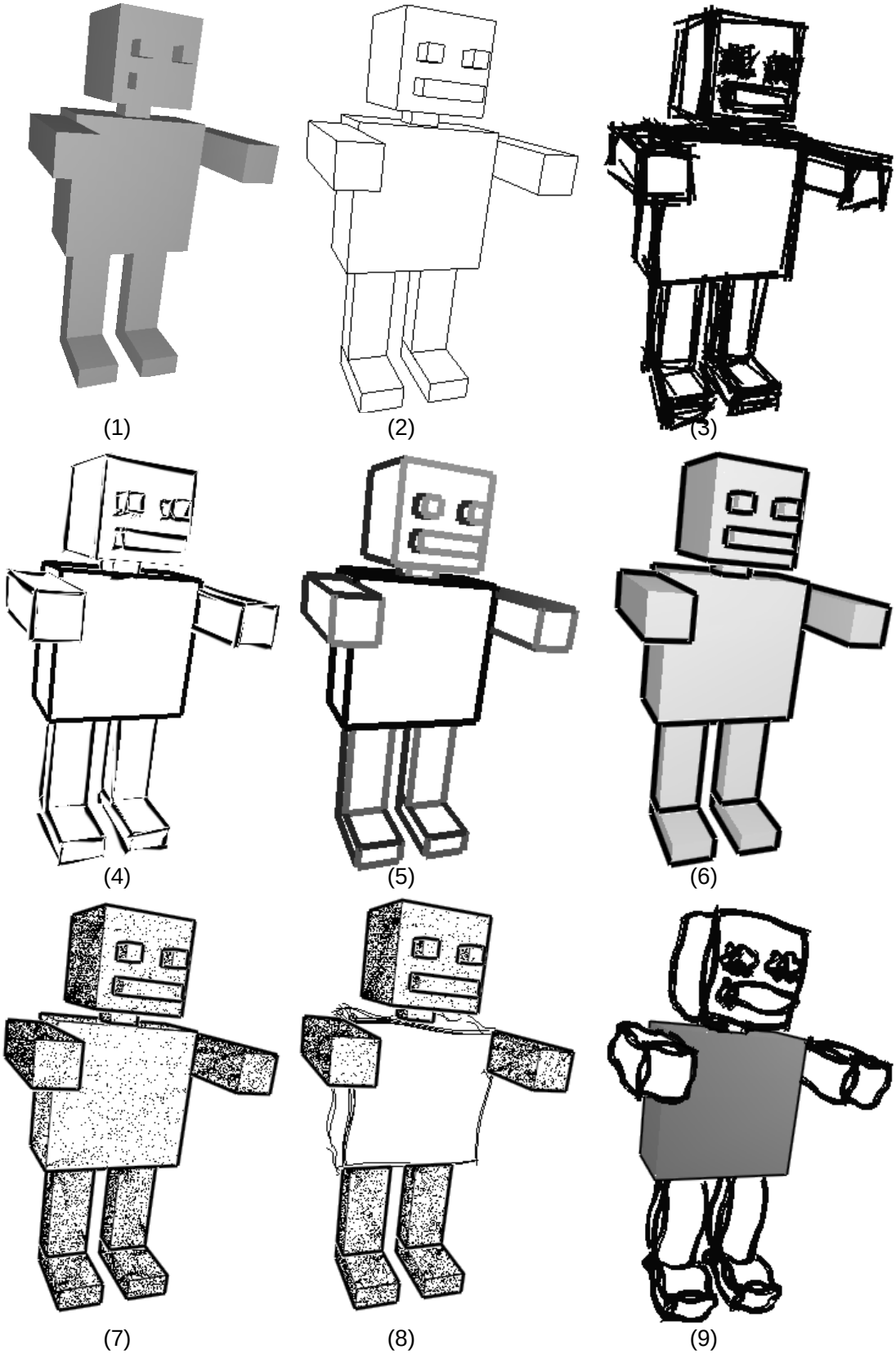

Abbildung 37. Verschiedene NPR-Techniken angewendet auf ein polygonales Modell.

6.8 Hybrides Rendering

Die abstrakte Bildrepräsentation kann in das generische Renderingsystem in Verbindung mit dem generischen Szenengraphen wie folgt integriert werden:

- *NPR-Attribute und NPR-Techniken.* Sie lassen sich als Attribute und Techniken des generischen Renderingsystems objektorientiert modellieren.
- *NPR-Engine zur Konstruktion von NPR-Bildern.* Die Konstruktion eines NPR-Bildes erfolgt durch eine spezialisierte Engine-Klasse, die die Zwischenrepräsentation während der Traversierung des Szenengraphen ableitet und die abstrakte Bildrepräsentation berechnet.
- *Konstruktion eines internen NPR-Szenengraphen.* Die Image-Patches werden gebündelt nach gemeinsamen Image-Patch-Attributen und Image-Patch-Renderern in einem internen Szenengraphen vorgehalten. Dieser Szenengraph wird sodann von einer Engine, die vom jeweiligen NPR-Renderingverfahren abhängt, traversiert. Zum Beispiel kann von den Image-Patch-Renderern, die ein konkretes Bild mit OpenGL rendern, die OpenGL-Engine genutzt werden.

Diese Herangehensweise stellt sicher, dass ein Szenengraph einer Anwendung mit Attributen und Techniken für das NPR erweitert wird, aber seine Grundstruktur behält. Dadurch lässt

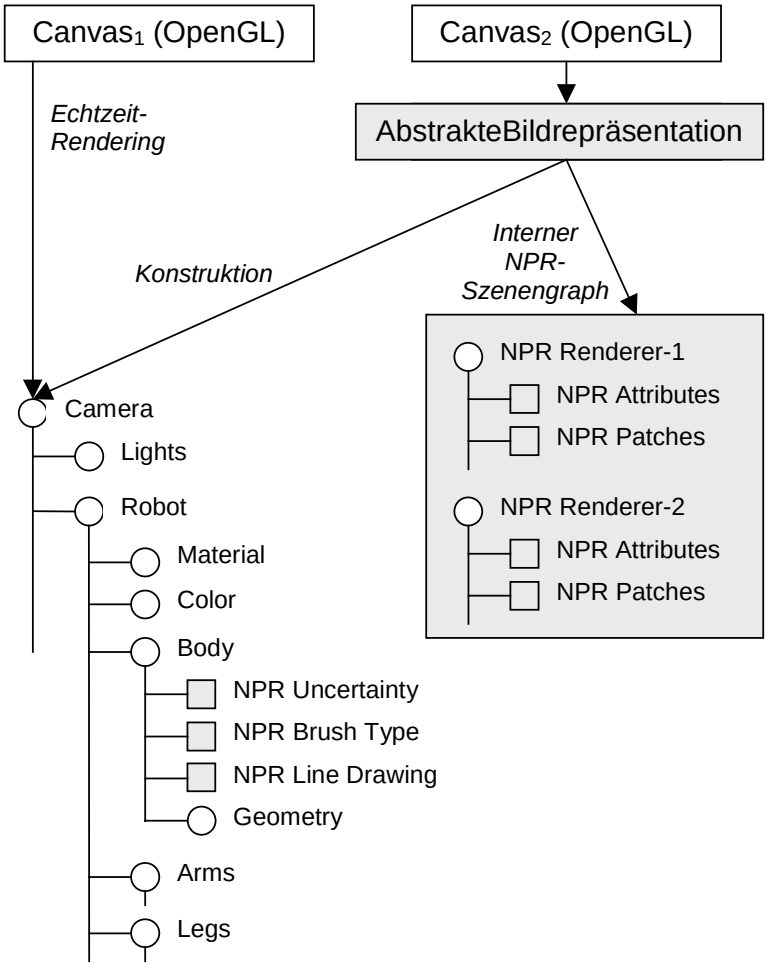

Abbildung 38. Beispiel eines Szenengraphen für den hybriden Einsatz von Renderingverfahren.

sich der Szenengraph weiterhin z.B. für das Echtzeit-Rendering nutzen; das nichtphotorealistische Rendering auf der Grundlage der abstrakten Bildrepräsentation kann so nahtlos in eine computergraphische Anwendung integriert werden.

In Abbildung 38 ist beispielhaft ein Szenengraph dargestellt, der NPR-Attribute und NPR-Techniken enthält. Der Szenengraph repräsentiert den Roboter der Beispielbilder der „Bildergalerie“. Der Szenengraph bildet die Grundlage für die Echtzeit-Darstellung und die NPR-Darstellung. Er enthält auch Unsicherheits- und Pinselattribute sowie die Image-Patch-Renderer.

Da der Sketch-Renderer auf der Basis von OpenGL implementiert ist, kann der resultierende interne Szenengraph zur Synthetisierung eines NPR-Bildes eine OpenGL-Engine verwenden. Wird die OpenGL-Engine des Echtzeit-Fensters dafür verwendet, kann sogar das NPR-Bild vollständig in die Echtzeit-Darstellung integriert werden. Diese Form des *hybriden Renderings* eröffnet insbesondere die Möglichkeit für die interaktive Nachbearbeitung und Anreicherung von OpenGL-generierten Bildern.

Innerhalb eines Rendering-Canvas können OpenGL- und NPR-Verfahren kombiniert eingesetzt werden. Die Voraussetzung hierfür ist, dass der interne NPR-Szenengraph logisch dem Hauptszenengraph zugeordnet wird. Dadurch wird erreicht, dass beide Teilgraphen die gleiche Engine, eine OpenGL-Engine, verwenden.

Der Aufbau des Gesamtszenengraphen muss dabei folgende Funktionalität besitzen:

- Im ersten Schritt wird die Szene in den Tiefenbuffer des Framebuffers gerendert; der Color-Buffer wird gesperrt.
- Im zweiten Schritt – falls noch nicht geschehen – wird die abstrakte Bildrepräsentation in Form des internen Szenengraphen konstruiert. Dieser wird dann gerendert, wobei der Tiefentest ausgeschaltet wird, um störende Artefakte zu vermeiden.
- Im dritten Schritt wird die Echtzeit-Darstellung erstellt, wobei der Tiefentest aktiviert wird.

Weitere Varianten des hybriden Einsatzes bieten sich an. Gemeinsam wird ihnen sein, dass sie über die geteilten OpenGL-Ressourcen kommunizieren und einen graphisch-geometrischen Sachverhalt durch die Anwendung mehrerer Renderingverfahren in das Zielmedium Bild umsetzen können.

6.9 Bewertung der Integration

Nichtphotorealistisches Rendering stellt aufgrund seiner grundsätzlich anderen Handhabung des Renderings eine Herausforderung für die heutigen Software-Architekturen computergraphischer Systeme dar. In allen in dieser Arbeit analysierten computergraphischen Systemen ist es nicht vertreten, da die Integration von NPR-Verfahren aufgrund der Software-Architektur nicht verwirklichbar ist.

Die abstrakte Bildrepräsentation, die in diesem Abschnitt vorgestellt wurde, bietet eine Möglichkeit, nichtphotorealistische Renderingverfahren zu implementieren. Mit Hilfe des generischen Renderingsystems, das keine Annahmen über die Struktur der Rendering-Pipeline aufstellt, sondern allgemein Renderingobjekte einer Engine zur Interpretation strukturiert übergibt, kann diese Form der Bildrepräsentation implementiert und nahtlos in das Renderingsystem und den generischen Szenengraphen eingefügt werden.

Wesentliche Elemente der Implementierung sind die Definition einer spezialisierten Engine zur Interpretation eines Szenengraphen für die abstrakte Bildrepräsentation und die Definition spezieller NPR-Attribute und NPR-Techniken zur Auswertung der Image-Patches.

7 FALLSTUDIE: TERRAIN-RENDERING

Geovisualisierung befasst sich mit Strategien und Techniken für die Präsentation und Exploration von Geo-Objekten und Geo-Prozessen. Unter *Geo-Objekt* und *Geo-Prozess* werden im folgenden diskretisierte Beschreibungen von Objekten und Prozessen im Raum verstanden, die durch raumzeitliche Daten erfasst sind. Das *Terrain-Rendering* repräsentiert eine Kategorie von grundlegenden Verfahren, die in der Geovisualisierung benötigt werden. Terrain-Rendering dient der bildhaften Umsetzung von raumzeitlichen Daten auf der Grundlage eines digitalen Geländemodells. Diese Renderingverfahren stehen vor der Herausforderung, meist große und größte Datenmengen für eine Darstellung in Echtzeit aufbereiten zu müssen. Eine weitere Herausforderung liegt in der Wahl geeigneter Visualisierungsstrategien und Metaphern, um für den Betrachter intuitive Darstellungen zu finden, die präzise Einblicke in die Geo-Objekte und Geo-Prozesse ermöglichen.

In dieser Fallstudie werden Terrain-Renderingverfahren für die kartenbasierte Darstellung raumzeitlicher Daten vorgestellt, die auf der Basis des Virtuellen Renderingsystems implementiert wurden. Dessen Fähigkeit, anwendungsspezifische Erweiterungen zu integrieren und Renderingalgorithmen flexibel zu handhaben, bildet die Voraussetzung für eine effiziente Implementierung dieser zeitkritischen Verfahren. Das Besondere in dieser Fallstudie liegt darin, dass ein objektorientiertes computergraphisches Software-System für das Echtzeit-Rendering großer Mengen von Geo-Daten verwendet wurde – ohne an Renderingleistung einzubüssen und Kompromisse bei der Nutzung der hardwarespezifischen Stärken einzugehen.

In dieser Fallstudie wird zunächst der Begriff der interaktiven 3D-Karte eingeführt. Ein Entwurf zur objektorientierten Modellierung interaktiver 3D-Karten schließt sich an. Im Einzelnen werden Geo-Visualisierungsstrategien, insbesondere texturbasierte Verfahren [23], erläutert und ihre Anwendung im Bereich der Konstruktion visueller Werkzeuge zur Exploration raumzeitlicher Daten [25] vorgestellt. Die Fallstudie schließt mit einer Übersicht der Software-Architektur eines interaktiven, kartenbasierten Visualisierungssystems.

7.1 Interaktive 3D-Karten

In vielen Anwendungsbereichen sind Karten grundlegende Werkzeuge zur Kommunikation räumlicher Daten. In den letzten Jahren wurden insbesondere digitale, dynamische 3D-Karten entwickelt, die auf der Basis von echtzeitfähiger 3D-Graphik einen interaktiven Zugang zu diesen Daten ermöglichen. Die Einsatzgebiete dieser Karten sind vielfältig und liegen zum Beispiel im Bereich der Geoinformationssysteme (GIS), der Navigationssysteme und in multi-

medialen Anwendungen. Allgemein können wir nach MacEachren [58] diese Karten als Werkzeuge und Bestandteile virtueller Geo-Umgebungen auffassen, die zur Präsentation, Exploration und Manipulation räumlicher und raumzeitlicher Daten verwendet werden [57].

Im Mittelpunkt dieser Fallstudie stehen interaktive 3D-Karten und Mechanismen zur Realisierung eines dynamischen Entwurfs des Karteninhalts. Die Unterstützung von Navigation und Orientierung durch gestalterische Mittel zählt hierbei zu den wichtigen Eigenschaften dieses Ansatzes, da diese Unterstützung die Verwendbarkeit und Effizienz des Kommunikationsprozesses zwischen Benutzer und Karte wesentlich bestimmt [55].

3D-Karten werden in unserem Ansatz aus einer Sammlung von 3D-Kartenobjekten konstruiert. Konzeptionell unterscheiden wir zwischen visuellen, strukturellen und verhaltengebenden 3D-Kartenkomponenten. Als Hauptkomponente enthalten 3D-Karten ein Geländemodell, das die im 3D-Raum eingebettete Referenzoberfläche beschreibt. Die Benutzbarkeit von 3D-Karten steht in enger Verbindung zum Entwurf des Karteninhalts: Darstellungsart, Intensität und Verteilung von räumlicher und raumzeitlicher Information müssen an die Benutzeranforderungen und die Bildauflösung adaptiert werden. 3D-Karten benötigen dazu leistungsfähige und flexible Methoden zum Entwurf des Karteninhalts. Effektive Navigation und Orientierung in 3D-Karten erfordern dynamische, kontextsensitive Gestaltung des Karteninhalts.

In unserem Ansatz können Kartenobjekte interaktiv über die integrierte Skriptsprache erzeugt, manipuliert, verknüpft, abgefragt und gelöscht werden. Auf diese Art lassen sich Karteninhalte dynamisch spezifizieren und anwendungsspezifische Funktionen integrieren. Die Skriptsprache dient dabei auch als „Klebesprache“ zur Verbindung weiterer vorgefertigter Software-Komponenten (z.B. Datenbanken) zu einer Anwendung; sie ermöglicht damit, wie Ousterhout [74] darlegt, das Programmieren auf einem deutlich höheren Abstraktionsniveau.

Technisch werden Werkzeuge zur Unterstützung von Navigation und Orientierung durch texturbasierte Kartenobjekte implementiert. Mehrfache Texturschichten sind das Hauptkonstrukt, um Techniken zur Fokussierung, Restriktion und Adaption von Information in 3D-Karten umzusetzen. Die präsentierten Techniken wurden in einem Framework für Geovisualisierung implementiert, wobei als Grundlage das Virtuelle Renderingsystem VRS in Verbindung mit OpenGL, dem Industriestandard für 3D-Renderingsysteme, eingesetzt werden. Für die Umsetzung dieses Projekts wurden insbesondere Renderingkomponenten für die Texturierung entwickelt, die sich nahtlos in VRS einfügen. Dadurch konnte eine transparente und leistungsfähige Implementierung erzielt werden, die einerseits von der Abstraktion durch VRS profitiert, andererseits alle technischen Eigenschaften der Computergrahik-Hardware geospezifisch nutzt.

7.2 Komponenten von 3D-Karten

Software-Bibliotheken mit Anwendungskomponenten für Geovisualisierung zeichnen sich durch einen deutlichen Zuschnitt auf Geo-Daten und eine höhere Abstraktion im Vergleich zu domänenübergreifenden Graphikbibliotheken aus. So entstand z.B. im Kontext von VRML eine Sammlung geospezifischer VRML-Knoten [82]. Die hier vorgestellten Komponenten verfolgen die gleiche Absicht.

Dieser Abschnitt diskutiert den Aufbau einer interaktiven 3D-Karte aus sog. *Kartenkomponenten*. Eine solche Karte wird aus einem Geländemodell, einer oder mehreren Kartenansichten und einer Sammlung von weiteren 3D-Kartenobjekten zur Spezifikation des Karteninhalts zusammengefügt. Das konzeptionelle Modell dieses Ansatzes ist in Abbildung 39 illustriert.

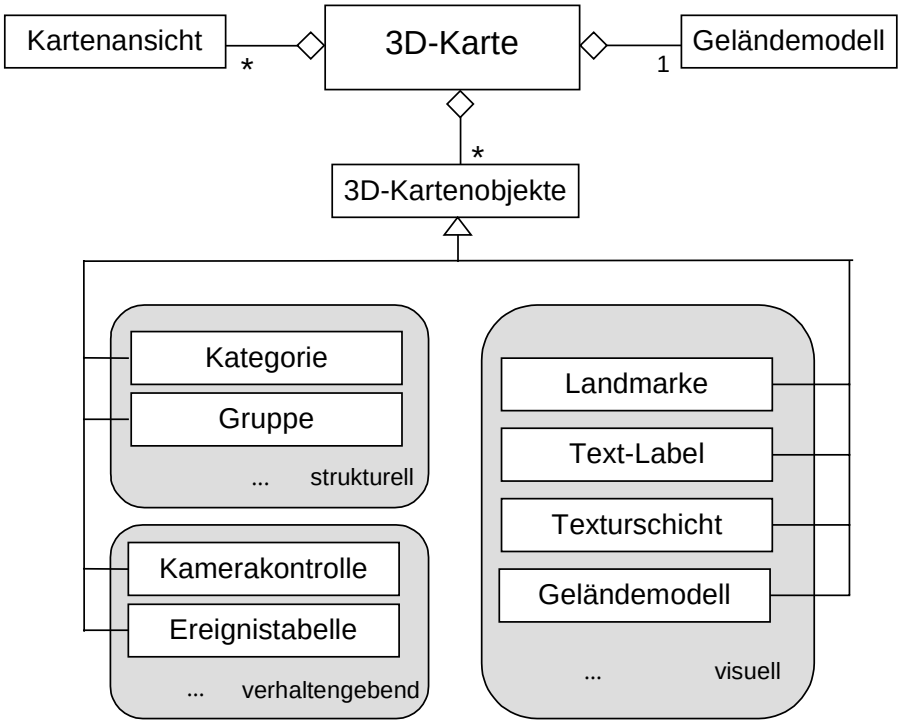

Abbildung 39. Konzeptionelles Modell der 3D-Kartenkomponenten.

Im Vergleich zu den Renderingkomponenten von VRS handelt es sich hierbei um Anwendungskomponenten, die spezifisch für Geovisualisierungsaufgaben entworfen wurden, wobei deren Implementierungen ausschließlich Renderingkomponenten, die von VRS bereitgestellt werden, nutzen.

7.2.1 Geländemodell

Das Geländemodell einer 3D-Karte repräsentiert die Geometrie der im Raum eingebetteten Referenzoberfläche. Im Allgemeinen wird dazu das Geländemodell in ein Multiresolutionsmodell übertragen. Es stellt die Geländeoberfläche, in einzelne Flächenstücke zerlegt, approximativ in unterschiedlichen Auflösungsstufen dar. Ein Multiresolutionsmodell ist notwendig, um mit komplexer, großer Geländegeometrie in Echtzeit umgehen zu können. In der Vergangenheit wurden hierzu unterschiedliche Techniken entwickelt, so z.B. die Multiresolutionstriangulierung von DeFloriani et al. [18], die progressiven Netze von Hoppe [50] und die restringierte Quadtree-basierte Triangulierung von Pajarola [76].

In unserem Ansatz verwenden wir ein generisches Multiresolutionsmodell, den *Approximationsbaum*, dessen Implementierung ausführlich bei Baumann, Döllner und Hinrichs [6] beschrieben ist. Der Approximationsbaum kombiniert reguläre Gitter mit TIN-Strukturen, um Regionen von besonderem Interesse topologisch und geometrisch exakt darzustellen, zu einem *hybriden Geländemodell* und ermöglicht so die genaue Wiedergabe der Geländemorphologie. Der Approximationsbaum nutzt je nach Typ der Geländedarstellung verschiedene Multiresolutionstechniken innerhalb eines hybriden Geländemodells. Der konzeptionelle Aufbau eines Approximationsbaums ist in Abbildung 40 illustriert.

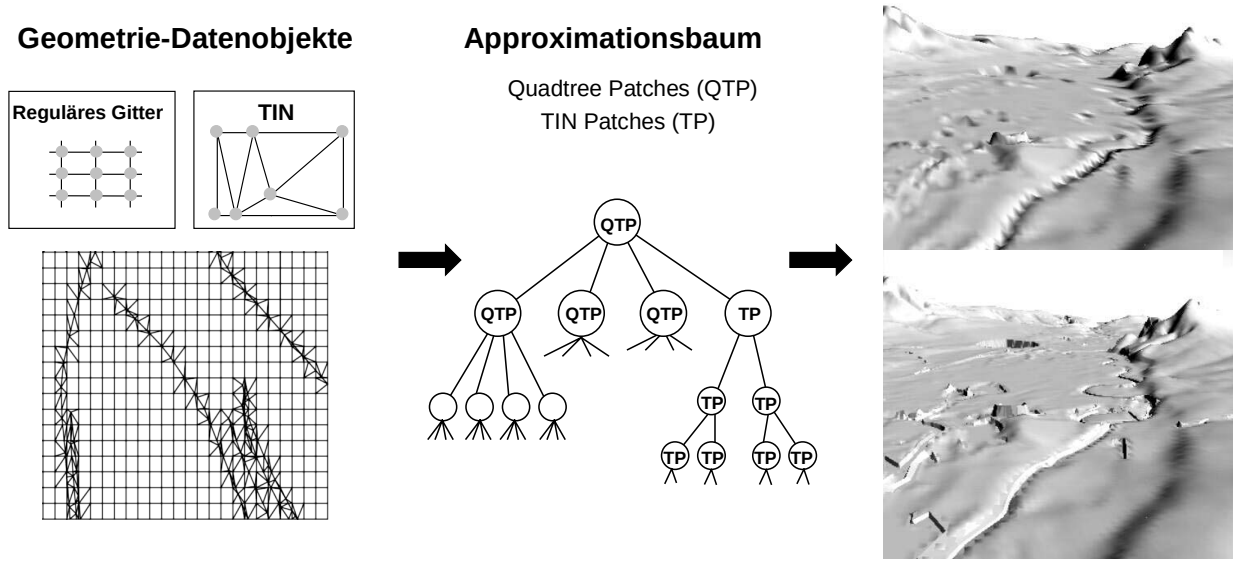

Abbildung 40. Konzeptionelle Sicht der Geometrie-Datenobjekte und eines zugehörigen Approximationsbaumes. Das reguläre Gitter und die TINs bilden ein hybrides Geländemodell. Die Visualisierung zeigt das Geländemodell ohne (oben) und mit TINs (unten).

7.2.1.1 Approximationsbaum als Multiresolutionsmodell

Der Approximationsbaum repräsentiert ein mit *Geometrie-Datenobjekten* beschriebenes Geländemodell durch einem Baum, der das Geländemodell hierarchisch in unterschiedlichen Auflösungen repräsentiert. Die Geometrie-Datenobjekte dienen als Eingabedaten für die Konstruktion eines Approximationsbaums. Sie umfassen Datenobjekte für reguläre Gitter und TINs.

Die Knoten des Baums werden als *Geometrie-Patches* bezeichnet. Jeder Knoten enthält eine approximierte Variante der Geländeoberfläche für einen rechteckigen Ausschnitt der Geländegrundfläche. Die Knotenklasse, der ein Geometrie-Patch angehört, bestimmt das Simplifikationsverfahren, das für die Berechnung der approximierten Oberfläche und für die weitere Unterteilung, dargestellt durch Kindknoten, verantwortlich ist. Für das hybride Geländemodell, das vom Approximationsbaum unterstützt wird, werden hauptsächlich zwei Knotenklassen eingesetzt, eine Gitter- und eine TIN-Knotenklasse.

Die in allen Knotenklassen definierte Renderingmethode selektiert Kindknoten auf der Basis eines bildraumabhängigen Fehlerkriteriums und wählt so je nach Kameraeinstellung und Fehlerschranke approximierende Geländeflächenstücke für die Bildsynthese aus. Eine detaillierte Beschreibung der Arbeitsweise des Approximationsbaums findet sich in [6].

Dieses Multiresolutionsmodell zur Darstellung von Geländemodellen lässt sich wie folgt charakterisieren:

- *Visuelle Qualität*. Durch die Hinzunahme von Mikrostrukturen in ein reguläres Gitter können morphologisch komplexe Geländeabschnitte exakt dargestellt werden. Die Vorteile des geringen Speicherbedarfs von Gittern und die Exaktheit von TINs werden dadurch kombiniert.
- *Semantikerhaltung.* Die Identität einzelner, verfeinernder TINs bleibt im Geländemodell erhalten. Bei einer späteren Editierung und Exploration des Modells ist dies insbesondere für ausgezeichnete Abschnitte des Geländes notwendig.

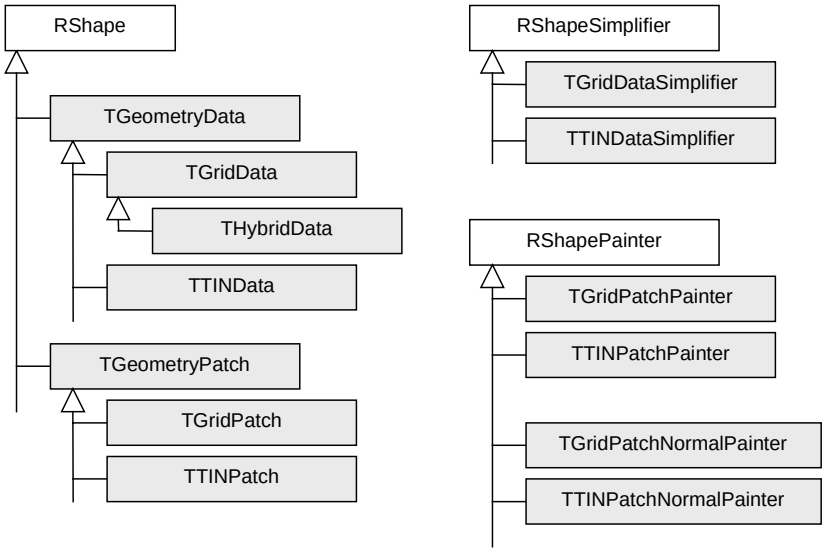

Abbildung 41. Integration der Geländedaten und Geländemultiresolutionsdarstellung in das Virtuelle Renderingsystem als Renderingkomponenten.

- *Wahl des Level-of-Detail-Algorithmus.* Aufgrund der objektorientierten Software-Architektur des Approximationsbaums gestaltet sich die Integration neuer Level-of-Detail-Algorithmen einfach. Sie werden als spezialisierte Knotenklassen implementiert.

7.2.1.2 Integration des Approximationsbaums

Beim Approximationsbaum handelt es sich um den Prototypen einer anwendungsspezifischen Datenstruktur, die möglichst effizient in den Renderingprozess eingebunden werden muss. Wäre eine Konvertierung der Geometrie-Patches in Renderingprimitive notwendig, dann könnte eine Darstellung in Echtzeit kaum realisiert werden, da die Objektverwaltungs- und Konvertierungskosten im zeitkritischen Renderingvorgang nicht zu rechtfertigen wären – eine direkte Implementierung mit OpenGL würde von den meisten Entwicklern vorgezogen werden.

Zur Integration des Approximationsbaums in das Virtuelle Renderingsystem werden Geometrie-Patches durch spezialisierte Shape-Typen modelliert (siehe Abbildung 41). Für TIN-Knoten und Grid-Knoten werden dazu je eine Shape-Klasse (*TGridPatch* und *TTINPatch*) definiert. Objekte dieser Shape-Klassen referenzieren einen Knoten des zugehörigen Approximationsbaums. Für jede dieser Shape-Klassen existiert ein korrespondierender OpenGL-Shape-Painter, der für die Umsetzung der approximierten Geländeoberfläche mit Hilfe von OpenGL verantwortlich ist (*TGridPatchPainter* und *TTINPatchPainter*).

Die Integration modelliert außerdem die Geometrie-Datenobjekte als Shapes (*TGridData* und *TTINData*). Dies stellt sicher, dass diese Objekte ebenfalls – aber ohne Level-of-Detail – gezeichnet werden können. Weiter existieren für sie Shape-Simplifier, die die Geländedaten in Form von Standard-Shapes (konkret: Polygone) des Virtuellen Renderingsystems übersetzen. Damit ist gewährleistet, dass ein Gelände mit allen adaptierten Renderingsystemen gezeichnet werden kann; z.B. ist damit eine Ausgabe mit RenderMan implizit erklärt. Ferner sind dadurch auch andere Services als das Rendering, z.B. Strahl-Objekt-Intersektionstest, automatisch für Geländemodelle verfügbar.

Für den Approximationsbaum existieren daneben sog. Normalen-Painter (*TGridPatchNormalPainter* und *TTINPatchNormalPainter*). Sind diese Painter im generischen Kontext einer Engine installiert, dann rendern sie eine Geländeoberfläche durch Umsetzung der Geländeober-

flächennormalen in Farbwerte. Diese Umsetzung ist notwendig, sollen die Oberflächennormalen in Form einer zweidimensionalen Farbtextur kodiert werden. Eine solche Textur kann in Verbindung mit den von nVidia eingeführten Register-Combiners [72], die eine Beleuchtungsberechnung pro Pixel ermöglichen, als Grundlage für das topographische Texturieren verwendet werden. Die Normalenwerte aus der Textur werden pro Pixel der Geländeoberfläche in die Beleuchtungsberechnung einbezogen. Die Variation in der Umsetzung geometrischer Primitive wird an diesem Beispiel deutlich: Die Interpretation eines Gelände-Patches obliegt den Paintern, die je nach Anforderung ausgetauscht werden können.

In diesem Entwurf kann das Virtuelle Renderingsystem mit seiner breiten Funktionalität voll genutzt werden, wird aber an der kritischen Stelle, dem Rendering der Geländeoberfläche, durch spezielle Shape-Typen ergänzt. Die Implementierung wird insgesamt im Vergleich zu einer direkten Implementierung mit OpenGL wesentlich vereinfacht, da sie von den VRS-Renderingkomponenten (z.B. Transformationen, Lichtquellen, Texturen) profitiert.

7.2.2 3D-Kartenobjekte

Der Inhalt einer 3D-Karte wird durch die mit ihr assoziierten 3D-Kartenobjekte definiert. Wir unterscheiden drei Kategorien von 3D-Kartenobjekten:

- *Visuelle 3D-Kartenobjekte* repräsentieren sichtbare, georeferenzierte 2D- und 3D-Objekte, die über graphische Attribute, wie z.B. Farbe, Größe, Form, Orientierung verfügen. Zu den visuellen 3D-Kartenobjekten zählen Landmarken, Text-Labels, 2D-Texturschichten (siehe Abschnitt 7.2.3) und 3D-Modelle. Ein konkretes visuelles 3D-Kartenobjekt ist das Geländemodell. Verglichen zu Primitiven von 3D-Graphiksystemen weisen visuelle 3D-Kartenobjekte eine fachspezifische Ausrichtung auf und verfügen über eine komplexere Struktur sowie ein komplexeres Verhalten.
- *Strukturelle 3D-Kartenobjekte* werden benutzt, um 3D-Kartenobjekte konzeptionell zu ordnen und um Beziehungen zwischen ihnen auszudrücken. Strukturen erlauben es dem Benutzer, große, individuelle Sammlungen von Kartenobjekten zu klassifizieren und zu organisieren. In 3D-Karten sind Kartenobjekte hierarchisch angeordnet, wobei die Hierarchien vielfältig sein können. Strukturelle 3D-Kartenobjekte sind z.B. Kategorie-Objekte und Gruppen-Objekte.
- *Verhaltengebende 3D-Kartenobjekte* können auf Ereignisse und Veränderungen durch den Aufruf von (anwendungsspezifischen) Aktionen reagieren, d.h. sie definieren die Interaktivität und die Dynamik von 3D-Karten. Beispiele für verhaltengebende 3D-Kartenobjekte sind Kamerakontrollen und Ereignistabellen (siehe Abschnitt 7.2.4). Eine Ereignistabelle ist in der Regel einem visuellen 3D-Kartenobjekt zugeordnet und erlaubt dadurch eine objektbezogene Spezifikation des Verhaltens.

Die integrierte Skriptsprache Tcl/Tk [73] ermöglicht die Konstruktion, die Nutzung und die Destruktion von 3D-Kartenobjekten zur Laufzeit. Der Einsatz dieser interaktiven Skriptsprache ermöglicht eine dynamische Individualisierung und Konfiguration von 3D-Karten.

7.2.3 Texturschichten

Für den visuellen Entwurf einer 3D-Karte ist die Gestaltung der Geländeoberfläche wesentlich. Die Gestaltung erfolgt in unserem Ansatz hauptsächlich mit Texturschichten. Eine *Texturschicht* verwaltet georeferenzierte Texturdaten, die auf das Geländemodell abgebildet werden, und berechnet eine Multiresolutionsdarstellung der Texturdaten, die in Korrelation zur Multiresolutionsdarstellung des Geländemodells strukturiert wird. Hierzu wird die *Texturbaum-*

Datenstruktur [7] eingesetzt. Die Multiresolutionsdarstellung ist besonders im Fall von großen Texturdaten notwendig, wie etwa bei kartographischen Texturen oder Satelliten-Bildern, um den begrenzten Texturspeicher der Graphikhardware optimal auszunutzen [15].

Eine 3D-Karte kann mehrere Texturschichten enthalten, d.h. das Geländemodell kann mit beliebig vielen Texturbäumen assoziiert werden. Texturschichten brauchen dabei das Geländemodell nicht vollständig zu überdecken, d.h. Texturschichten lassen sich partiell für das Gelände definieren. Dort wo sich Texturschichten überlappen, müssen Texturschicht-Operationen festgelegt werden, wie z.B. Addition, Verblenden und Multiplikation.

Das Rendering einer Karte mit mehreren Texturschichten kann mit Hilfe eines Multi-Pass-Algorithmus erfolgen, der das Geländemodell mit der jeweiligen Textur mehrfach im Bildraum zeichnet. Falls die Graphikhardware jedoch Multitexturing [108] unterstützt, d.h. mehrere Texturen simultan auf die darzustellende Geometrie anwenden kann, so wird die Zahl der Renderingdurchläufe reduziert und entsprechend die Darstellungsgeschwindigkeit erhöht. Multitexturing ist zunehmend selbst auf preiswerter Graphikhardware verfügbar (z.B. nVidia GeForce).

7.2.3.1 Texturschichttypen

Konzeptionell unterscheiden wir zwischen vier Typen von Texturschichten:

- Eine *Thematik-Texturschicht* repräsentiert räumliche Daten, die als Farbbilder oder Graustufenbilder visualisiert werden. Thematische Texturen liegen entweder direkt, z.B. als Satellitenbild, vor oder werden auf der Grundlage von Gestaltungsregeln generiert, wie z.B. im Fall von kartographischen Texturen. Die Hauptaufgabe einer Thematik-Texturschicht ist die Kommunikation von thematischer Information und ihrer räumlichen Zusammenhänge in Bezug auf das Geländemodell.
- Eine *Topographie-Texturschicht* kodiert die Schattierungsinformation des Geländemodells. Sie wird benutzt, um die pixelpräzise Schattierung des Geländemodells unabhängig von der zugrundeliegenden Level-of-Detail-Geometrie zu ermöglichen. Die Hauptaufgabe der Topographie-Texturschicht ist die Kommunikation der morphologischen Information.
- Eine *Luminanz-Texturschicht* repräsentiert Helligkeitswerte, die zur Modifikation der Erscheinung anderer Texturschichten verwendet werden. Technisch verbirgt sich hinter einer Luminanz-Texturschicht eine 1-Kanal Textur, die z.B. automatisch von der Anwendung erzeugt werden kann. Die Hauptaufgabe einer Luminanz-Texturschicht ist die Helligkeitsvariation von Geländeregionen (siehe Abschnitt 7.3.2).
- Eine *Alpha-Texturschicht* enthält Gewichtungen (z.B. Transparenzwerte), die zur Verblendung zweier anderer Texturschichten verwendet werden (siehe Abschnitt 7.3.3). Technisch verbirgt sich hinter einer Alpha-Texturschicht eine 1-Kanal Textur, die automatisch von der Anwendung erzeugt werden kann.

7.2.3.2 Implementierung von Texturschichten

Texturschichten sind in der Gesamtarchitektur des interaktiven 3D-Kartensystems Fachobjekte; sie werden mit Hilfe von Renderingobjekten des Virtuellen Renderingsystems implementiert. Zur Umsetzung einer Texturschicht wird insbesondere die Multitexturing-Technik (*RMultitexturing*) verwendet, die es erlaubt, eine oder mehrere Texturen auf geometrische Objekte anzuwenden. Texturschichten sind als Technik-Objekte modelliert, da je nach Zahl der Texturen und den Fähigkeiten der Hardware mehrere Renderingdurchläufe notwendig sind.

Die Implementierung des Texturschichtkonzepts unterscheidet drei Arten der Mehrfachtexturierung:

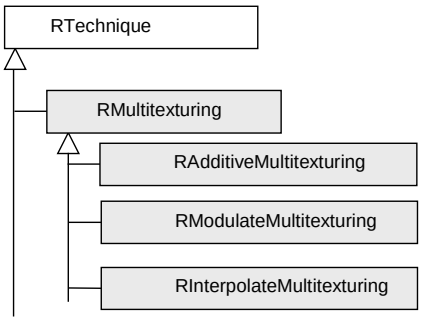

Abbildung 42. Integration der Multitexturing in das virtuelle Renderingsystem.

- *RAdditiveMultitexturing*. Diese Technik wendet mehrere Texturen additiv auf die geometrischen Objekte an. Das Gewicht einer einzelnen Textur wird als Konstante spezifiziert. Die Addition erfolgt im Bildraum.
- *RModulateMultitexturing*. Diese Technik multipliziert mehrere Texturen miteinander. Die Multiplikation erfolgt im Bildraum.
- *RInterpolateMultitexturing*. Diese Technik mischt mehrere Texturen auf der Basis von Gewichten, die in einer separaten Alpha-Textur spezifiziert sind. Die Mischung erfolgt im Bildraum.

7.2.4 Intelligente 3D-Kartenobjekte

3D-Kartenobjekte können mit komplexem, objektspezifischem Verhalten ausgestattet werden. Das Verhalten kann die Erscheinung eines Kartenobjekts dynamisch modifizieren und bestimmen, wie auf Ereignisse bzw. Situationsänderungen reagiert wird. Das Verhalten der einzelnen 3D-Kartenobjekte legt insgesamt das visuelle Design und die Interaktivität einer 3D-Karte fest. Insbesondere erlauben intelligente 3D-Kartenobjekte die Individualisierung und Konfigurierung von vorgefertigten 3D-Karten, da benutzerspezifische Funktionalität durch Hinzufügen von Verhalten integriert werden kann.

In unserem Ansatz wird das Verhalten in Form von Ereignisreaktionen mit Hilfe von Skripten beschrieben, die durch den integrierten Tcl-Interpreter evaluiert werden. Damit kann das Verhalten einer 3D-Karte zur Laufzeit ohne erneute Kompilation verändert werden. Jedes visuelle 3D-Kartenobjekt verfügt über eine Ereignistabelle. Die Ereignistabelle berücksichtigt Ereignisse in Form von Veränderungen des Kartenzustands und des Zustands einzelner Kartenobjekte (z.B. Sichtbarkeit, Distanz, Zeit) sowie Ereignisse in Form von Benutzerinteraktionen (z.B. Objektselektion und Objektintersektion). Jeder Eintrag in einer Ereignistabelle ist mit einer Aktion, d.h. einem Tcl-Skript, verknüpft. Die Ereignistabellen der einzelnen Kartenobjekte wachen über den Eintritt von Ereignissen.

7.2.4.1 Landmarken

Landmarken sind visuelle 3D-Kartenobjekte, die durch eine räumliche Position, einen Namen, eine Gruppenzugehörigkeit und eine Kategoriezugehörigkeit definiert sind. Landmarken können interaktiv durch die Skriptsprache erzeugt und in der 3D-Karte platziert werden. Jede Landmarke verfügt über mindestens eine Ereignistabelle.

7.2.4.2 Automatische Kamerakontrolle

Die automatische Kamerakontrolle ist ein verhaltengebendes 3D-Kartenobjekt, das zur Einschränkung von Kameraposition und –richtung eingesetzt wird. Die Idee ist dabei, eine Menge von Landmarken mit unterschiedlichen Gewichten zu platzieren, um dann die Kameraparameter zu berechnen, bei denen alle Landmarken sichtbar sind und die Landmarke mit dem höchsten Gewicht im Zentrum der 3D-Kartenansicht steht [70]. Wenn sich die Gewichtungen oder die Positionen ändern, werden die Kameraeinstellungen neu errechnet.

Die automatische Kamerakontrolle optimiert die Kameraeinstellungen für Situationen, in denen der Benutzer nur eingeschränkt navigieren soll. Weitere Ansätze zur automatischen Kamerakontrolle (z.B. Kamera-Constraints [44]) lassen sich in Analogie integrieren.

7.3 Terrain-Renderingverfahren

Konzeptionell ist nun der Aufbau einer interaktiven 3D-Karte geklärt: Geländemodell, Texturschichten und weitere 3D-Kartenobjekte konstituieren eine Karte. Die Frage stellt sich nun, mit welchen Renderingverfahren diese Bestandteile computergraphisch umgesetzt werden können. Die Verfahren zur computergraphischen Umsetzung von kartenbasierten Darstellungen werden im folgenden unter dem Begriff *Terrain-Renderingverfahren* zusammengefasst.

Das Terrain-Rendering muss insbesondere die präzise morphologische Darstellung und die dynamische Gestaltung der Karte erlauben. Damit können insbesondere die Navigation und Orientierung des Benutzers unterstützt werden, da dadurch sowohl die visuelle Präzision als auch die kognitive Belastung des 3D-Karteninhalts an den Kontext und den Benutzer angepasst werden können [75]. Beide Faktoren haben einen großen Einfluss auf die Verwendbarkeit der Karte, weil sie die präzise und effektive Nutzung des Bildraums erlauben.

7.3.1 Darstellung der Gelände-Morphologie

Die genaue Wahrnehmung der Karten-Morphologie ist eine Voraussetzung für effektive Navigation und Orientierung. Wir nehmen die Morphologie eines Geländes, das auf ein zweidimensionales Fenster projiziert wird, über die Geländeschattierung und Geländesilhouette war. Insbesondere zeigen Untersuchungen in der Psychophysik, dass das Erkennen einer dreidimensionalen Figur allein über seine Schattierung eine grundlegende präkognitive Fähigkeit darstellt [54]. In unserem Ansatz benutzen wir topographische Texturen zur Schattierung des Geländes, d.h. wir ersetzen die übliche geometriebasierte Schattierung des Geländemodells durch eine Schattierungstextur aus zweierlei Gründen:

- *Gestaltbarkeit der Schattierung:* Topographische Texturen bieten mehr Freiheitsgrade bei der Gestaltung der Schattierung, denn das lokale Beleuchtungsmodell des zugrundeliegenden Renderingsystems beschränkt nicht das visuelle Design der topographischen Textur. So können wir zum Beispiel morphologisch interessante Geländebereiche, wie Flussverläufe oder Bahntrassen, durch besonderes graphisches Design und durch Anwendung besonderer Schattierungsalgorithmen betonen.
- *Pixelpräzise Schattierung:* Topographische Texturen werden pro Pixel der rasterisierten Geländeoberfläche berechnet und hängen damit nicht von der momentanen geometrischen Auflösung der Level-of-Detail-Darstellung des Geländemodells ab. Diese Art der texturbasierten Schattierung garantiert somit eine stetige Erscheinung des Geländes während des Navigierens.

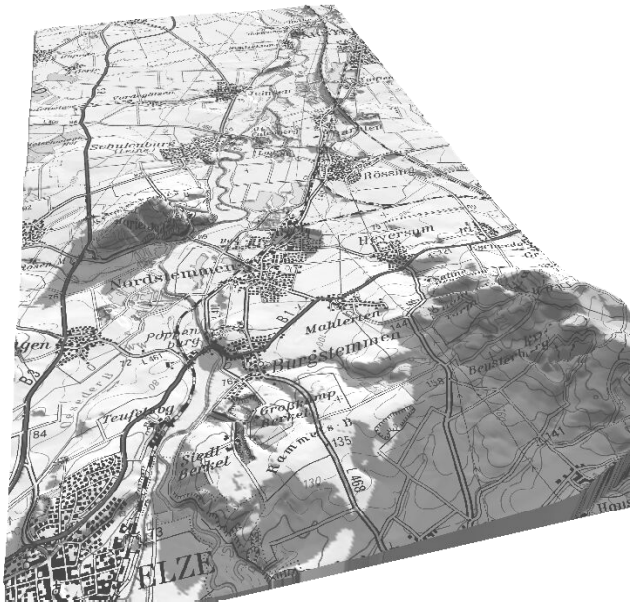

Abbildung 43. Karte mit topographischer Textur, die Selbstschattierung enthält, kombiniert mit einer kartographischen Textur.

Abbildung 44. Kartographisches Schattierungsverfahren für eine Karte des Himalaya-Gebirges.

- Außerdem kann mit diesem Verfahren die Renderinggeschwindigkeit erhöht werden, da kostenintensive Beleuchtungsberechnungen, die pro Eckpunkt des geometrischen Modells durchgeführt werden müssten, vermieden werden.

Die Erstellung topographischer Texturen bedeutet in der Praxis keinen merkbaren Mehraufwand, denn diese Texturen können automatisiert berechnet werden. Zur automatisierten Berechnung wird eine orthogonale Projektion des Geländemodells in einem nichtsichtbaren Renderingfenster gezeichnet (z.B. Backbuffer eines OpenGL-Fensters), dessen Inhalt dann in eine 2D-Textur kopiert wird. Die Automatisierung ist zudem notwendig, falls topographische Texturen infolge von Änderungen der Schattierungsparameter neu berechnet werden müssen.

Abbildung 43 zeigt eine 3D-Karte mit topographischer Textur, die zusätzlich eine Selbstschattierung enthält. Eine solche topographische Textur kann automatisiert berechnet werden, indem zunächst eine lokal beleuchtete topographische Textur mit dem oben beschriebenen Verfahren erstellt wird. Anschließend kann die Selbstschattierung für jeden Geländepunkt dadurch ermittelt werden, dass ein Teststrahl vom Geländepunkt zur Lichtquelle gesendet wird. Die Strahl-Objekt-Intersektionsberechnung ist ein Standard-Service des Virtuellen Renderingsystems und ist direkt für die Geländeoberfläche verfügbar.

Abbildung 44 zeigt eine 3D-Karte des Himalaja-Gebirges, deren topographische Textur von Kartographen in einem separaten Arbeitsschritt vorberechnet wurde. Derartige Verfahren, die meist mit Hilfe von Regeln und Prozeduren spezifiziert werden, können in das interaktive 3D-Kartensystem mit der dort definierten (derzeit noch experimentellen) *prozeduralen Texturschicht* implementiert werden. Eine solche Texturschicht definiert den Inhalt kontext- und regelabhängig; sie kann pro Frame automatisch im Hintergrund generiert werden.

7.3.2 Visuelle Fokussierung von Informationen

Durch die Kontrolle eines visuellen Fokus in der 3D-Karte kann die Navigation und die Orientierung des Benutzers unterstützt werden, indem wir die Aufmerksamkeit des Benutzers in der Kartenansicht und in den in ihr dargestellten Prozessen lenken. Technisch werden zu diesem Zweck Luminanz-Texturen verwendet, die eine visuelle Fokuskontrolle ermöglichen. Zwei Anwendungen dieser visuellen Fokussierung sind im folgenden kurz illustriert:

- *Raumzeitliche Prozesse:* Wenn wir den visuellen Fokus animieren, können wir raumzeitliche Prozesse visualisieren, wie z.B. das Wandern entlang eines Weges oder Fahren eines Fahrzeugs.
- *Räumliche Positionen:* Räumliche Positionen, z.B. Kartenpunkte und Kartenregionen, können durch den visuellen Fokus hervorgehoben werden. Die visuelle Fokuskontrolle kann an Benutzeraktionen geknüpft werden, z.B. indem die Luminanz-Texturschicht in Abhängigkeit der Benutzerinteraktion modifiziert wird.

Die Echtzeit-Visualisierung ist für die Interaktivität der visuellen Fokuskontrolle notwendig und wird dadurch gewährleistet, dass die Luminanz-Texturschicht unabgängig von anderen Texturschichten auf das Geländemodell angewendet werden kann. Andernfalls wären aufwendige Bildoperationen nötig, um alle aktiven Texturschichten zu einer Texturschicht zu kombinieren; diese Bildoperationen wären insbesondere für hochaufgelöste Texturdaten nicht in Echtzeit durchführbar. Der Einsatz von Multi-Pass-Rendering in Verbindung mit Multitexturing ist in der Praxis hingegen selbst für große Texturen effizient durchführbar.

Der visuelle Fokus wird durch eine Luminanz-Textur spezifiziert, und dadurch kann z.B. die Form einer Linse und die Gewichtung, mit der die Inhalte anderer Texturschichten darin erscheinen, frei gewählt werden. Die Luminanz-Texturen können z.B. in Form von Bilddateien in eine 3D-Karte importiert oder direkt von der Anwendung generiert werden. Im Allgemeinen sind für Luminanz-Texturen geringe Auflösungen, wie z.B. 128x128 Pixel, ausreichend, da sie direkt im Bildraum – durch Texturfilterung zusätzlich geglättet – angewendet werden.

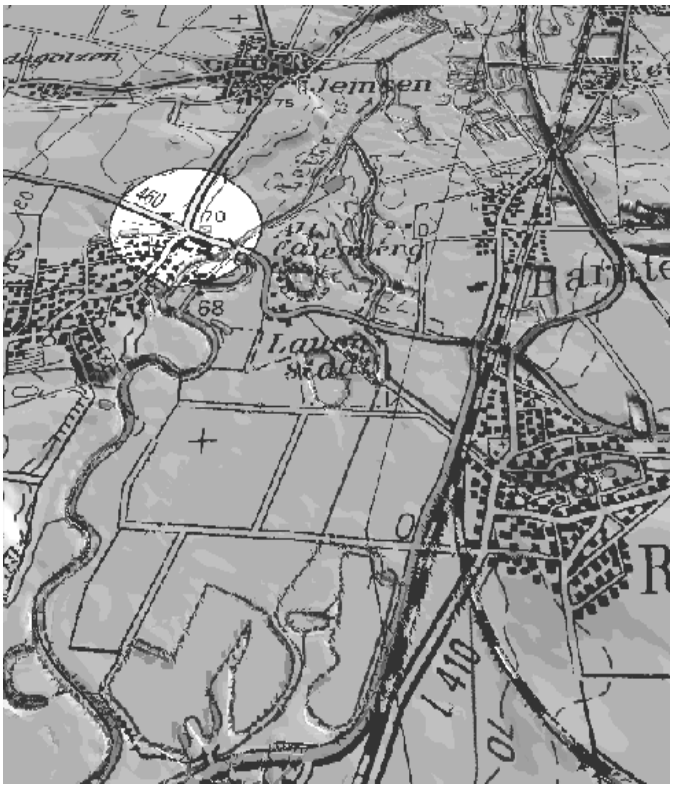

Abbildung 45. Fokussierung räumlicher Positionen durch zirkuläre Luminanz-Linse.

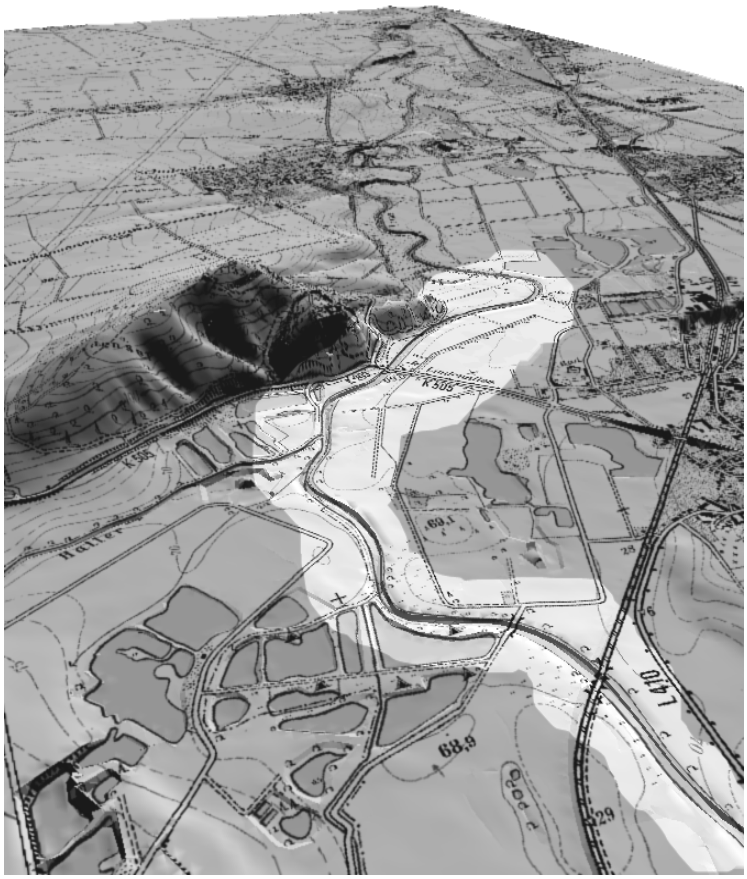

Abbildung 46. Fokussierung räumlicher Positionen durch eine Pfad-Linse.

In Abbildung 45 wird ein visueller Fokus verwendet, um räumliche Positionen hervorzuheben. Dazu wird eine Luminanz-Texturschicht definiert, die einen kreisförmigen Bereich, die

Linse, mit hoher Intensität beschreibt und im restlichen Bereich eine konstant niedrigere Intensität besitzt. Angewendet auf andere Texturschichten bewirkt dies eine Verdunkelung außerhalb der Linse. Die Position der Linse kann dadurch interaktiv manipuliert werden, dass der Mittelpunkt der Linse an die aktuelle Mausposition gekoppelt wird. Dazu wird permanent ein Strahl, der von der Mausposition in das Gelände hinein gesendet wird, mit dem Geländemodell geschnitten; falls eine Intersektion vorliegt, werden die Texturkoordinaten der Luminanzschicht mit Hilfe der Texturmatrix justiert, so dass die Linse der Maus folgt.

Abbildung 46 zeigt einen programmgenerierten visuellen Fokus, der dem Verlauf eines Flusses folgt. Der Flussverlauf ist in Form einer Liste von Landmarken gegeben. Eine Luminanzschicht kann automatisiert als prozedurale Texturschicht erstellt werden, die mit Hilfe von 2D-Maloperationen eine Region entlang des Flussverlaufs mit hoher Intensität markiert.

7.3.3 Visuelle Restriktion von Information

Die Visualisierung thematischer Information erstreckt sich in klassischen Karten gleichmäßig über die gesamte (Papier-)Karte, um den Informationsgehalt zu maximieren. Werden solche thematischen Daten in interaktiven 3D-Karten visualisiert, dann erstrecken sich thematische Texturen ebenfalls vollständig über die gesamte Geländeoberfläche der Karte. Die Gestaltung

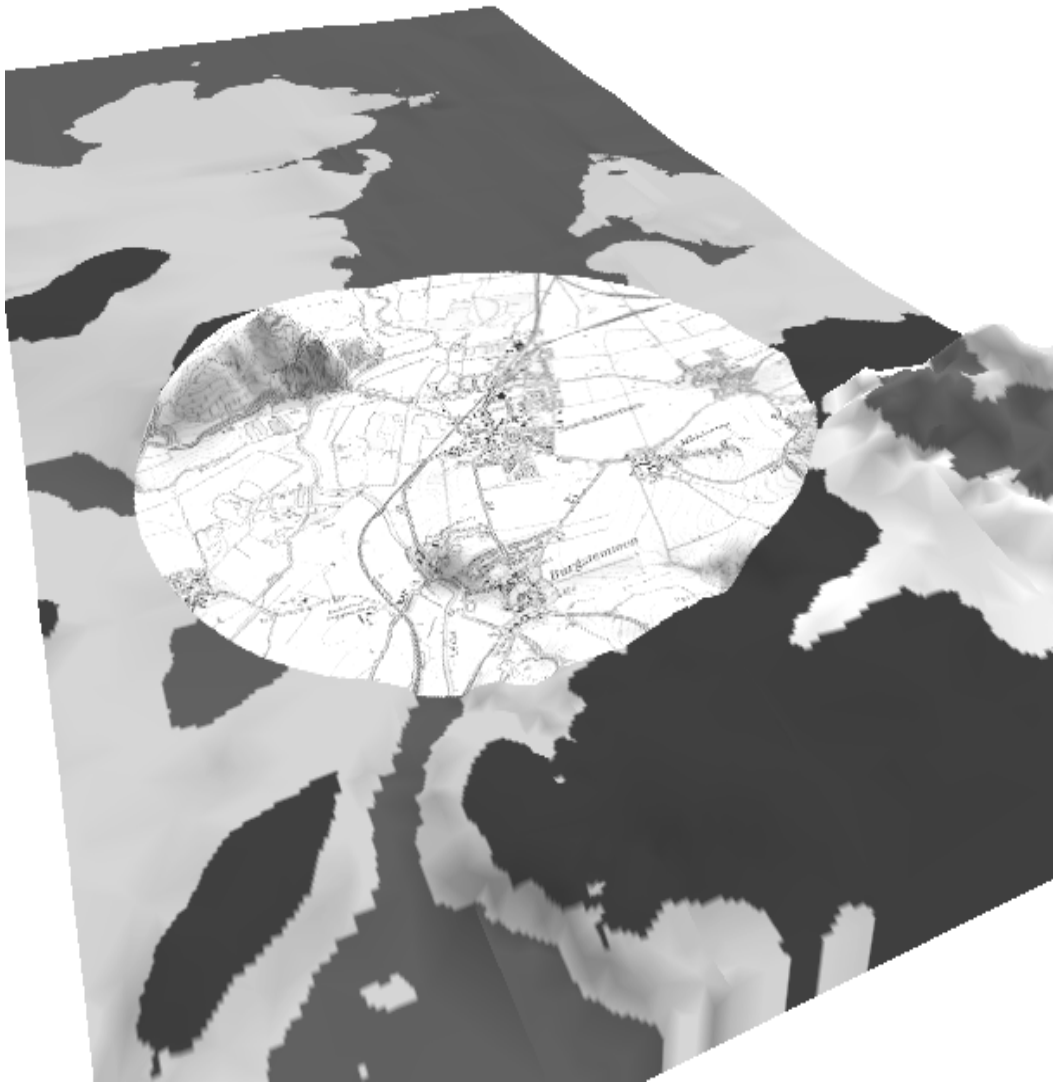

Abbildung 47. Thematische Linse zur Assoziation verschiedener Thematiken am Rand der Linse.

Abbildung 48. Nichtkreisförmige Linse zur partiellen Reduktion der thematischen Komplexität.

einer interaktiven 3D-Karte kann jedoch verbessert werden, indem die Visualisierung thematischer Informationen auf Bereiche von besonderem Interesse restringiert wird. Dadurch wird insbesondere die kognitive Belastung der Karte optimiert. Folgende Anwendungen illustrieren dieses Prinzip:

- *Visuelle Assoziation:* Die partielle Kombination von thematischen Texturen erlaubt es, unterschiedliche Thematiken miteinander in Verbindung zu bringen. Abbildung 47 zeigt eine 3D-Karte mit einer groben, höhenabhängig gefärbten Texturschicht. Innerhalb der thematischen Linse wird zusätzlich eine kartographische Textur visualisiert. Entlang des Linsenübergangs können wir beide Informationen, d.h. die Beziehung zwischen Höhenregion und kartographischer Textur, assoziieren.
- *Reduktion der Inhaltskomplexität:* Visuelle Restriktion von Information impliziert, dass Information dort sichtbar wird, wo der Benutzer die Information sehen möchte. Die Orientierung in der Karte wird dadurch erhöht, dass die Komplexität des Karteninhalts in Bereichen von geringem Interesse reduziert wird. In Abbildung 48 wird eine nichtkreisförmige Linse eingesetzt, um bestimmte Texturschichten, die alle kartographische Symbole enthalten, auszublenden. Durch diese Radier-Funktion können ausgewählte Texturschichten isoliert sichtbar werden.

Die visuelle Restriktion von Information in interaktiven 3D-Karten erfolgt mit Hilfe von Alpha-Texturen, die die Gewichte für die Kombination jeweils zweier thematischer Texturschichten enthalten. Die Erstellung solcher Texturen erfolgt im Allgemeinen programmgesteuert und kann ggf. zur interaktiven Kontrolle an Eingabegeräte gekoppelt werden. Auch benutzerdefinierte Alpha-Texturen können verwendet werden.

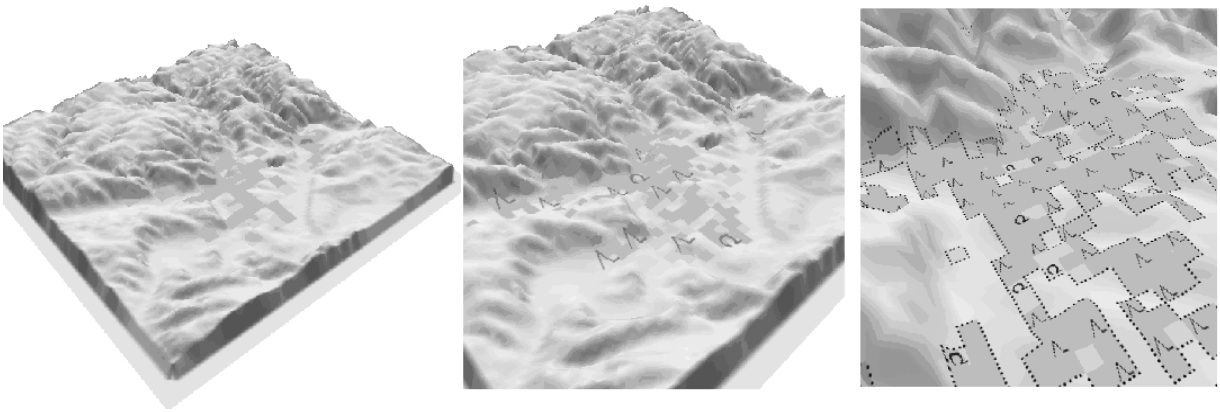

Abbildung 49. Distanzabhängige thematische Texturschichten mit Landnutzungsinformation.

7.3.4 Visuelle Adaption von Information

Häufig resultieren Probleme in der Wahrnehmung des Karteninhalts digitaler Karten aus einem inadäquaten Entwurf des Inhalts. Die Qualität des Entwurfs für interaktive 3D-Karten kann verbessert werden, wenn der Entwurf Kameradistanz, Kamerarichtung und Bildauflösung als wesentliche Parameter einbezieht (z.B. bei der Positionierung von Labels bei van Dijk [105] oder bei der Gestaltung der Kartensymbole bei Haeberling [43]). Ein adaptiver Entwurf sichert eine adäquate Repräsentation topographischer und thematischer Informationen in Abhängigkeit vom momentanen Ansichtskontext.

Um Informationen flexibel darzustellen, d.h. den Bildraum effektiv zu nutzen, muss es möglich sein, Texturschichten zur Laufzeit neu zu erzeugen. Darüber hinaus wird eine Textur-Beschreibungssprache notwendig, um den Texturinhalt in einer abstrakten, prozeduralen Art und Weise zu formulieren. Diese Lösung hat den Vorteil, dass auch Designschemata und –regeln, z.B. zur kartographischen Generalisierung [67], in den Entwurf aufgenommen werden können.

Interaktive 3D-Karten benötigen von daher eine prozedurale Textur-Beschreibungssprache und unterstützen eine automatische Erzeugung der Texturschichten. Dies kann z.B. zur Entwicklung von dynamischen, thematischen Texturen benutzt werden, die abhängig von der Kameradistanz und Bildauflösung passende Signaturen für thematische Daten enthalten [105].

Abbildung 49 zeigt unterschiedliche Detailstufen einer Landnutzungs-Texturschicht. Basierend auf einer prozeduralen Spezifikation wurde die Landnutzungsinformation auf farbige Bereiche, auf farbige Bereiche mit Signaturen und in der höchsten Stufe auf farbige Bereiche mit Signaturen und zusätzlichen Rändern abgebildet.

7.4 Software-Architektur des LandExplorer-Systems

Die vorgestellten Konzepte für interaktive 3D-Karten wurde prototypisch im *LandExplorer*-System umgesetzt. Abbildung 50 illustriert dessen Software-Architektur.

Die oberste Schicht enthält die Kartenkomponenten; sie ist primär für Software-Entwickler von interaktiven 3D-Karten gedacht. Die Kartenkomponenten stützen sich einerseits auf das Multiresolutionsmodell für Gelände-Geometrie, den Approximationsbaum, und das Multiresolutionsmodell für Texturdaten, den Texturbaum. Sie nutzen andererseits Renderingkomponenten von VRS. Die Implementierung des Approximationsbaums und des Texturbaums erweitern

VRS um neue Renderingkomponenten. Die einzelnen Schichten der Software-Architektur sind in Form von C++-Klassenbibliotheken implementiert.

Um einen interaktiven Zugriff auf die Funktionalität der 3D-Karte und ihre Kartenobjekte zu erhalten, wurden ausgewählte C++-Klassen mit einer Tcl/Tk-Schnittstelle [73] ausgestattet, d.h. für jede dieser C++-Klassen existiert eine Sammlung von Tcl-Kommandos, die die Konstruktion von Objekten und deren Manipulation durch Methodenaufrufe ermöglichen.

Eine Anwendung, die eine interaktive 3D-Karte verwenden möchte, integriert sowohl die C++-Klassenbibliothek als auch den Tcl/Tk-Interpreter. Der Anwender erhält damit Zugriff auf die Interpreter-Kommandos. 3D-Kartenobjekte können über die Skriptsprache erzeugt, manipuliert, verknüpft, abgefragt und gelöscht werden.

In unserer Erfahrung bietet die hybride Lösung bestehend aus System-Programmiersprache und Skriptsprache exzellente Möglichkeiten für die Individualisierung und Konfiguration komplexer Visualisierungsanwendungen. Obwohl die meisten Kommandos durch die Skriptsprache ausgelöst werden, führen C++-Objekte letztendlich die zeitkritischen Operationen durch. Aus diesem Grund kann der zusätzliche Overhead der Skriptingsprache toleriert werden.

7.5 Bewertung der Integration

Die Umsetzung von interaktiven 3D-Karten mit Hilfe computergraphischer Systeme bietet ein komplexes Beispiel für die vielschichtigen Software-Architekturanforderungen, die ein computergraphisches System zu erfüllen hat, um eine effiziente Implementierung von anwendungsspezifischen Renderingverfahren zu gewährleisten.

Die Grundlage jeden Terrain-Renderings, das Level-of-Detail-Modell der Geländeoberfläche, konnte durch spezialisierte Shape-Typen in den Kern des Virtuellen Renderingsystems aufgenommen werden. Das Konzept der Texturschichten wurde auf der Grundlage der Multitexturing-Technik implementiert und kooperiert mit der Multiresolutionsdarstellung der Texturdaten, dem Texturbaum. Weitere Kartenobjekte, wie z.B. Labels oder Glyphen, werden mit Hilfe der Shape-Sammlung des Virtuellen Renderingsystems implementiert. Selbst mit einfacher Graphikhardware können interaktive Bildraten, auch im Fall von 4 oder 8 Textur-

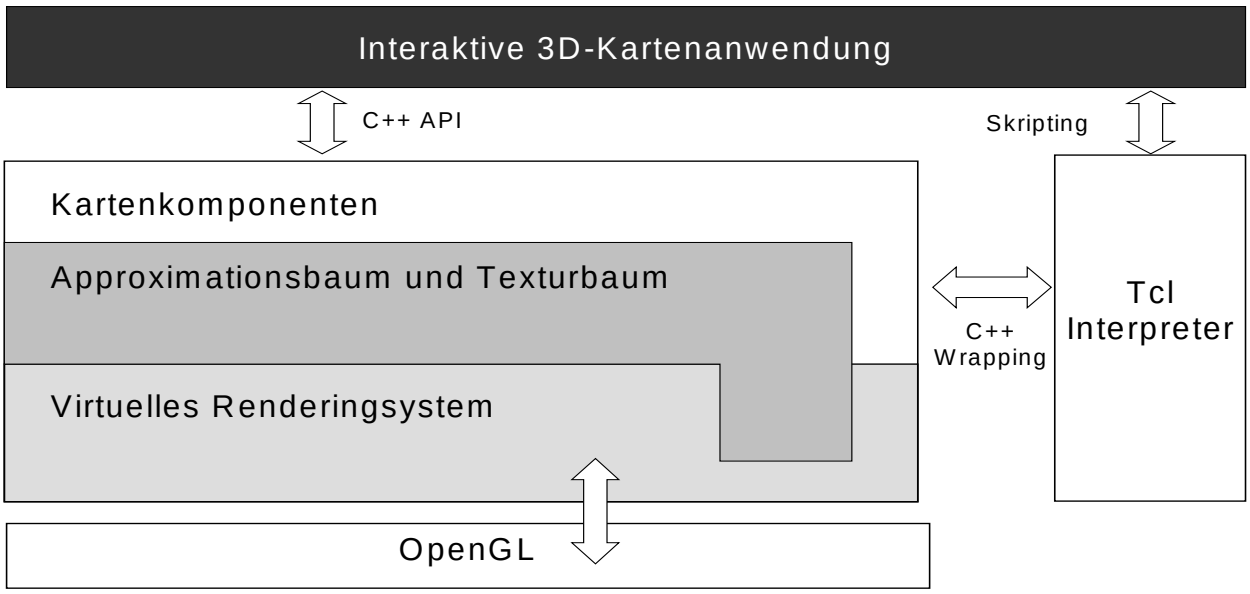

Abbildung 50. Software-Architektur des interaktiven 3D-Kartensystems *LandExplorer*.

schichten, dank des Multiresolutionsmodells für Texturdaten und der Unabhängigkeit jeder einzelnen Texturschicht sowie durch die Nutzung von Multitexturing realisiert werden.

Die Interaktionsfähigkeiten der Kartenkomponenten greifen in ihrer Implementierung auf den Strahl-Objekt-Intersektionstest zurück, der für alle Shape-Objekte, auch für das Geländemodell, verfügbar ist. Dadurch wird eine präzise Ermittlung von 3D-Intersektionspunkten bereitgestellt. Das hybride Geländemodell stellt darüber hinaus sicher, dass die Semantik der Geo-Datenobjekte in der computergraphischen Repräsentation erhalten bleibt.

Zusammenfassend kann gesagt werden, dass das zeitkritische und in den Verfahren komplexe Terrain-Rendering auf der Grundlage des Virtuellen Renderingsystems vollständig implementiert werden konnte. Dies zeigt, dass ein objektorientiertes Renderingsystem unter der Voraussetzung der Offenheit gegenüber anwendungsspezifischen Komponenten und hardwarenaher Unterstützung von Renderingeigenschaften im Vergleich zu einer direkten Umsetzung mit OpenGL zu mindestens leistungsgleichen Produkten sowie langfristig zu abstrakteren und transparenteren Software-Architekturen führt.

8 SCHLUSSFOLGERUNGEN

Die vorliegende Arbeit hat die Software-Architektur computergraphischer Systeme, insbesondere die der Renderingsysteme, diskutiert. Die Untersuchung bestehender computergraphischer Systeme ergab, dass die Verständlichkeit, Verwendbarkeit und Erweiterbarkeit ihrer Leistungsmerkmale ungeachtet ihrer Implementierungskonzepte die entscheidenden Faktoren für ihre Verbreitung und Nutzung darstellen. Jedoch besteht die Notwendigkeit, ihre Konzepte in einen größeren Zusammenhang zu stellen, um diese Systeme verstehen und vergleichen zu können.

Das vorgestellte Konzept eines generischen Renderingsystems versucht, einen solchen einheitlichen Framework zu schaffen. Es definiert ein konzeptionelles Modell für Renderingkomponenten und darauf aufbauende Renderingalgorithmen. Es begreift Rendering als eine Auswertung von Renderingkomponenten mit Hilfe von Auswertungsalgorithmen und Auswertungsstrategien. Insofern ist es ein generisches Renderingsystem, da es keine Begrenzungen hinsichtlich der Arbeitsweise und der Zielmedien der Auswertung aufstellt.

Die wesentlichen Merkmale der Software-Architektur des generischen Renderingsystems sind die strikte Trennung von Shapes und Attributen, die strikte Trennung von deklarativen Bestandteilen (Shapes und Attribute) und ihrer Implementierung (Handler) sowie die Abstrahierung der Auswertung der Renderingkomponenten durch Engines und Techniken.

Aufbauend auf dem generischen Renderingsystem wurde gezeigt, wie ein generischer Szenengraph zur Kodierung von hierarchisch strukturierten, graphisch-geometrischen Sachverhalten entworfen werden kann. Wesentliches Merkmal seiner Software-Architektur ist die strikte Trennung von Szenenstruktur (Knoten) und Szeneninhalt (Renderingkomponenten). Insbesondere ermöglicht der generische Szenengraph die Verwendung von Renderingkomponenten für unterschiedliche Renderingsysteme innerhalb des gleichen Szenengraphen, unterstützt die Implementierung von Multi-Pass-Rendering und ermöglicht deklarative Szenenmodellierung.

Die Anwendung des generische Renderingsystems und des generischen Szenengraphen wurde anhand dreier Fallstudien illustriert und vertieft. Die Fallstudie des Multi-Pass-Renderings zeigt, dass diese ausdrucksstarke Klasse von Renderingverfahren im generischen Renderingsystem grundlegend verankert und leistungsfähig unterstützt wird. Die Fallstudie des nichtphotorealistischen Renderings zeigt, dass neuartige Bildrepräsentationen und Bildsyntheseverfahren als kanonische Erweiterungen des generischen Renderingsystems implementiert und in den generischen Szenengraph integriert werden können. Die Fallstudie des Terrain-Renderings zeigt schließlich, dass in der Praxis für konkrete und zeitkritische Anwendungen tragfähige und effiziente Software-Lösungen mit Hilfe der gezielten Erweiterung des generischen Renderingsystems entwickelt werden können.

Zugleich wird deutlich, dass existierende computergraphische Systeme mit Hilfe einer auf sie aufbauenden Software-Architektur konzeptionell und technisch „ummantelt“ werden können. Die Ummantelung führt zu einer Abstraktion, die die Verständlichkeit, Wiederverwendbarkeit und Erweiterbarkeit dieser Systeme erhöht. Sie ermöglicht es auch, Systeme zu vergleichen und einzuordnen. Die Abstraktion dient aber auch der Entwicklung neuer computergraphischer Systeme, deren Software-Architektur auf einem stabilen und in der Praxis erprobten Fundament aufbauen kann.

LITERATUR

[1] Adobe Systems Incorporated. *PostScript Language Reference*. 3rd Edition, Addison-Wesley, 1999.

[2] S. Amann, C. Streit, H. Bieri. BOOGA – A Component-Oriented Framework for Computer Graphics. *GraphiCon '97 Proceedings*, 193-200, 1997.

[3] A. Apodaca, L. Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures.* The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, 1999.

[4] A. Appel, F.J. Rolf, A.J. Stein. The Haloed Line Effect for Hidden Line Elimination. *Computer Graphics (Proceedings SIGGRPAH '79)*, 151-157, 1979.

[5] Autodesk. *Heidi Developer Guide.* Version 6.0, Autodesk Corporation, 1999.

[6] K. Baumann, J. Döllner, K. Hinrichs, O. Kersting. A Hybrid, Hierarchial Data Structure for Real-Time Terrain Visualization. *Computer Graphics International CGI 99*, 85-92, 1999.

[7] K. Baumann, J. Döllner, K. Hinrichs. Integrated Multiresolution Geometry and Texture Models for Terrain Visualization. *Proceedings of the Joint EUROGRAPHICS and IEEE TCGV Symposium on Visualization*, 157-166, 2000.

[8] E. Beier. A Generic Approach to Computer Graphics. In: C. Hege, K. Polthier (Hrsg.), *Visualization and Mathematics*. Springer-Verlag, 1997.

[9] E. Beier. Issues on Hierarchical Graphical Scenes. In: R.C. Veltkamp, E. H. Blake (Eds.). *Programming Paradigms in Graphics*, Springer Computer Science, 3-12, 1995.

[10] E. Beier, U. Bozzetti. A Generic Graphics Kernel and a Customized Derivative. In 6th EuroGraphics Workshop on Rendering, 1995.

[11] D.R. Begault. *3D Sound for Virtual Reality and Multimedia*. Academic Press Professional, 1994.

[12] P. Bergeron. A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics and Applications*, 6(9):17-28, 1986.

[13] B. Bühlmann. *Extensions and Applications of the Graphics Framework BOOGA*. Inauguraldissertation, Universität Bern, 1998.

[14] P.R. Calder, M.A. Linton. Glyphs: Flyweight Objects for User Interfaces. *Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology*, 1990.

[15] D. Cline, P. Egbert. Interactive Display of Very Large Textures. *Proceedings Visualization '98*, 343-350, 1998.

[16] F. Crow. Shadow Algorithms for Computer Graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 242-248, 1977.

[17] C.J. Curtis, S.E. Anderson, J.E. Seims, K.W. Fleischer, D.H. Salesin. Computer-Generated Watercolor. *Computer Graphics (Proceedings SIGGRAPH '97)*, 421-430, 1997.

[18] L. De Floriani, P. Magillo, E. Puppo. Efficient Implementation of Multi-Triangulations. *Proceedings Visualization '98*, 43-50, 1998.

[19] P. Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-based Multi-pass Rendering.* Ph.D. Thesis, University of Pennsylvania, 1996.

[20] P. Diefenbach, N. Badler. Pipeline Rendering: Interactive Refractions, Reflections, and Shadows. *Displays*. Special Issue on Interactive Graphics, 15(3):173-180, 1994.

[21] S. Dietrich. Dot Product Texture Blending and Per-Pixel Lighting. NVIDIA Corporation, White Paper, http://www.nvidia.com, 2000.

[22] D.J. Duke, I. Herman. Programming Paradigms in an Object-Oriented Multimedia Standard. *Computer Graphics forum*, 17(4):249-261, 1998.

[23] J. Döllner, K. Baumann, K. Hinrichs. Texturing Techniques for Terrain Visualization. *Proceedings IEEE Visualization 2000*, 227-234, 2000.

[24] J. Döllner. Abstract Image Representation for Non-Photorealistic 3D Graphics. *SCCG 2000 Spring Conference on Computer Graphics*, 76-85, 2000.

[25] J. Döllner, O. Kersting. Visuelle Unterstützung von Navigation und Orientierung in interaktiven 3D-Karten. Workshop "Guiding Users through Interactive Experiences", Universität Paderborn, 2000, im Druck.

[26] J. Döllner, K. Hinrichs. Interactive, Animated 3D Widgets. *Computer Graphics International CGI 98*, 278-286, 1998.

[27] J. Döllner, K. Hinrichs. The Design of 3D Rendering Meta System. In: F. Arbab, P. Slussalek (Hrsg.). *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Computer Graphics*, 43-54, 1997.

[28] J. Döllner, K. Hinrichs. Object-Oriented 3D Modeling, Animation and Interaction. *The Journal of Visualization and Computer Animation*, 8(1):33-64, 1997.

[29] P.K. Egbert, W.J. Kubitz. Application graphics modeling support through object-orientation. *IEEE Computer* 25(10):84-91, 1992.

[30] P.K. Egbert. Utilizing Renderer Efficiencies in an Object-Oriented Graphics System. *Programming Paradigms in Graphics*, Springer-Verlag, 13-22, 1995.

[31] C. Everitt. One-Pass Silhouette Rendering with GeForce and GeForce2. NVIDIA Corporation, White Paper, http://www.nvidia.com, 2000.

[32] J. Foley. 3D Graphics Standards Debate: PEX versus OpenGL. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 408-409, 1992.

[33] J. Foley, A. van Dam, S. Feiner, J.F. Hughes. *Computer Graphics. Principles and Practice*, Addison-Wesley, 1990.

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

[35] A. Glassner. *An Introduction to Ray-Tracing.* Academic Press Inc., 1989.

[36] B. Gooch, P.J. Sloan, A. Gooch, P. Shirley, R. Riesenfeld. Interactive Technical Illustration. *Symposium on Interactive 3D Graphics*, 31-38, 1999.

[37] A. Gooch, B. Gooch, P. Shirley, E. Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. *Computer Graphics (Proceedings of SIGGRAPH '98)*, 447-452, 1998.

[38] C.M. Goral, K.E. Torrance, D.P. Greenberg, B. Battaile. Modeling the Interaction of Light Between Diffuse Surfaces. *Computer & Graphics*, 18(3):213-222, 1984.

[39] L. Gritz, J.K. Hahn. BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools*, 1(3):29-47, 1996.

[40] P. Haeberli, M. Segal. Texture Mapping as a Fundamental Drawing Primitive, *Proceedings of the Fourth Eurographics Workshop on Rendering*, 259-266, 1993.

[41] P. Haeberli. Paint By Numbers: Abstract Image Representations. *Computer Graphics (Proceedings of SIGGRAPH '90)*, 24(4):207-214.

[42] P. Haeberli, K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 309-318, 1999.

[43] C. Haeberling. Symbolization in Topograhic 3D Maps. Conceptual Aspects for User-Oriented Design. *19th International Cartographic Conference*, 1037-1044, 1999.

[44] A. Hanson, E. Wernert. Constrained 3D Navigation with 2D Controllers. *Proceedings IEEE Visualization 97*, 175-182, 1997.

[45] W. Heidrich, K. Daubert, J. Kautz, H.P. Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. *Computer Graphics (Proceedings SIGGRAPH 2000)*, 2000.

[46] W. Heidrich, H.P. Seidel. Realistic, Hardware-accelerated Shading and Lighting. *Computer Graphics (SIGGRAPH '99 Proceedings)*, 171-178, 1999.

[47] W. Heidrich, P. Slusallek, H.P. Seidel. An Image-Based Model for Realistic Lens Systems in Interactive Computer Graphics. *Proceedings of Graphics Interface '97*, 68-75, 1997.

[48] R. Herrell, J. Baldwin, C. Wilcox. High-Quality Polygon Edging. *IEEE Computer Graphics and Applications*, 15(4):68-73, 1995.

[49] A. Hertzmann. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In: *Nonphotorealistic Rendering SIGGRAPH Course Notes*, S. Green (Hrsg.), 1999.

[50] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *Proceedings Visualization '98*, 35-42, 1998.

[51] ISO/IEC 14772-1:1997. *The Virtual Reality Modeling Language*. Verfügbar bei http://www.vrml.org/Specifications, 1997.

[52] M.J. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs. *GDC 2000 - Advanced OpenGL Game Development*, 2000.

[53] M.J. Kilgard. Improving Shadows and Reflections via the Stencil Buffer. NVIDIA Corporation, White Paper, http://www.nvidia.com, 2000.

[54] D.A. Kleffner, V.S. Ramachandran. On the perception of shape from shading. *Perception & Psychophysics*, 52(1), 18-36, 1992.

[55] M.J. Kraak. Interactive Modelling Environment for Three-dimensional Maps: Functionality and Interface Issues. In: A. M. MacEachren, D. R. F. Taylor (Eds.), *Visualization in Modern Cartography*, Pergamon, 269-285, 1994.

[56] J. Lansdown, S. Schofield. Expressive Rendering: A Review of Nonphotorealistic Techniques. *IEEE Computer Graphics and Applications*, 15(3):29-37, 1995.

[57] A.M. MacEachren, R. Edsall, D. Haug, R. Baxter, G. Otto, R. Masters, S. Fuhrmann, L. Qian. Exploring the Potential of Virtual Environments for Geographic Visualization. *Association of American Geographers*, 1999.

[58] A.M. MacEachren, M.J. Kraak, E. Verbree. Cartographic Issues in the Design and Application of Geospatial Virtual Environments. *19th International Cartographic Conference*, 657-665, 1999.

[59] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[60] L. Markosian, M.A. Kowalski, S.J. Trychin, L.D. Bourdev, D. Goldstein, J.F. Hughes. Real-Time Nonphotorealistic Rendering. *Computer Graphics (SIGGRAPH '97 Proceedings)*, 415-420, 1997.

[61] M. Masuch, S. Schlechtweg, B. Schönwälder. daLi! - Drawing Animated Lines!. In: O. Deussen, P. Lorenz (Eds.): *Proceedings Simulation and Animation '97*, 87-96, 1997.

[62] T. McReynolds, D. Blythe, B. Grantham. Advanced Graphics Programming Techniques Using OpenGL. *SIGGRAPH 99 Course Notes*, 1999.

[63] B.J. Meier. Painterly Rendering for Animation. *Computer Graphics (SIGGRAPH '96 Proceedings)*, 477-484, 1996.

[64] B. Meyer. *Object-Oriented Software Construction.* 2nd Edition, Prentice Hall, 1997.

[65] S. Meyers. *More Effective C++. 35 Ways to Improve Your Programs and Design*. Addison-Wesley, 1997.

[66] T. Möller, E. Haines. *Real-Time Rendering.* A K Peters, 1999.

[67] D. Morgenstern, D. Schürer. A Concept for model generalization of digital landscape models from finer to coarse resolution. *19th International Cartographic Conference*, 1021-1928, 1999.

[68] MPEG (Motion Picture Expert Group). *Overview of the MPEG-4 Standard.* Verfügbar bei http://www.mpeg.org, 1994.

[69] MPEG (Motion Picture Expert Group). *Overview of the MPEG-7 Standard.* Verfügbar bei http://www.mpeg.org, 1997.

[70] T. Noma, N. Okada. Automating Virtual Camera Control for Computer Animation. In: *Creating and Animating the Virtual World*, N. Magnenat-Thalmann, D. Thalmann (Hrsg.), 177-187, 1992.

[71] J.D. Northrup, L. Markosian. Artistic Silhouettes: A Hybrid Approach. *Symposium on Non-Photorealistic Animation and Rendering (Proceedings of NPAR 2000)*, 31-37, 2000.

[72] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, http://www.nvidia.com, 1999.

[73] J. Ousterhout. *Tcl/Tk*. Addison-Wesley, 1994.

[74] J. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23-30, 1998.

[75] S. Oviatt. Multimodel Interfaces for Dynamic Interactive Maps. *CHI 96*, 95-102, 1996.

[76] R. Pajarola. Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation. *Proceedings IEEE Visualization '98*, 19-26, 1998.

[77] B. Paul. Mesa3D. http://www.mesa3d.org

[78] Persistency of Vision Ray Tracer (POV-Ray). User's Documentation, verfügbar bei http://www.pov.org.

[79] PHIGS+ Committee, A. van Dam. PHIGS+ functional description, Rev. 3.0, *Computer Graphics*, 22(3):125-218, 1988.

[80] P.J. Plauger. *The Draft Standard C++ Library*. Prentice Hall, 1995.

[81] W.T. Reeves, D.H. Salesin, R.L. Cook. Rendering Antialiased Shadows with Depth Maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 283-291, 1987.

[82] M. Reddy, L. Iverson, Y. Leclerc. Under the Hood of GeoVRML 1.0. *Proceedings of The Fifth Web3D/VRML Symposium*, 23-28, 2000.

[83] J. Rumbaugh, I. Jacobson, G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[84] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[85] M.P. Salisbury, M.T. Wong, J.F. Hughes, D.H. Salesin. Orientable Textures or Image-Based Pen-and-Ink Illustration. *Computer Graphics (SIGGRAPH '97 Proceedings)*, 401-406, 1997.

[86] T. Saito, T. Takahashi. Comprehensible Rendering of 3-D Shapes. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):197-206, 1990.

[87] M.P. Salisbury, S.E. Anderson, R. Barzel, D.H. Salesin. Interactive Pen-and-Ink Illustrations. *Computer Graphics (SIGGRAPH '94 Proceedings)*, 101-108, 1994.

[88] S. Schlechtweg. Lines and how to draw them. In: Norsk samarbeid inner grafisk databehandling Ed.): NORSIGS info, medlemsblad for NORSIGD, 2/97, 4-6, 1997.

[89] M. Segal, K. Akeley. *The Design of the OpenGL Graphics Interface*. White Paper, Silicon Graphics, 1994.

[90] D.D. Seligmann, S. Feiner. Automated Generation of Intent-Based 3D Illustrations. *Computer Graphics (SIGGRAPH '91 Proceedings)*. 123-132, 1991.

[91] Silicon Graphics. *OpenGL Optimizer Technical Info*. White Paper, 1998.

[92] P. Slusallek, H.-P. Seidel. Vision: An architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77-96.

[93] P. Slusallek, H.-P. Seidel. Object-Oriented Design for Image Synthesis. In: R. C. Veltkamp, E. H.Blake (Eds.). *Programming Paradigms in Graphs*, Springer Computer Science, 23-34, 1995.

[94] M. Stein, E. Bowman, G. Perce. *Direct3D: A Professional Reference*. New Riders Publishing, 1997.

[95] C. Streit. *BOOGA. Ein Komponentenframework für Grafikanwendungen*. Inauguraldissertation, Universität Bern, 1997.

[96] T. Strothotte. *Computational Visualization. Graphics, Abstraction, and Interactivity*. Springer Verlag, 1998.

[97] T. Strothotte, S. Schlechtweg. *Non-Photorealistic Computer Graphics: Computational Visualistics in Action.* Vorlesungsskript, Otto-von-Guericke-Universität Magdeburg, 2000.

[98] B. Stroustrup. *Die C++ Programmiersprache.* 3. Auflage, Addison-Wesley, 1998.

[99] H. Sowizral. Scene Graphs in the New Millennium. *IEEE Computer Graphics and Applications*, 20(1):56-57, 2000.

[100] H. Sowizral, K. Rushforth, M. Deering. *The Java 3D API Specification*. 2nd Edition, Addison-Wesley, 2000.

[101] P. Strauss, R. Carey. An object-oriented 3D graphics toolkit. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 341-449, 1992.

[102] C. Szyperski. *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[103] N. Tsingos, J.D. Gascuel. Soundtracks for Computer Animation: Sound Rendering in Dynamic Environments with Occlusions. *Graphics Interface (Proceedings GI '97)*, 9-16, 1997.

[104] S. Upstill. *The RenderMan Companion. A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley, 1989.

[105] S. van Dijk, M. van Krefeld, T. Strijk, A. Wolff. Towards an Evaluation of Quality for Label Placement Methods. *19th International Cartographic Conference*, 905-913, 1999.

[106] B.R. Vatti. A Generic Solution to Polygon Clipping. *Communications of the ACM*, 35(7):56-63, 1992.

[107] L. Vepstas. The GLE Tubing and Extrusion Library. Online-Documentation.

[108] M. Woo, J. Neider, T. Davis, D. Shreiner. *OpenGL Programming Guide.* 3rd Edition. Addison-Wesley, 1999.

[109] D. Wang, I. Herman, G.J. Reynolds. The Open Inventor Toolkit and the PREMO Standard. *Computer Graphics forum*, 16(4):159-175, 1997.

[110] G. Ward Larson, R. Shakespeare. *Rendering with Radiance*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, 1998.

[111] K. Weiler, P. Atherton. Hidden Surface Removal Using Polygon Area Sorting. Computer Graphics (*SIGGRAPH '77 Proceedings)*, 214-222, 1977.

[112] J. Wernecke. *The Inventor Mentor. Programming Object-Oriented 3D Graphics with OpenInventor*. Addison-Wesley, 1993.

[113] G. Wiegand, B. Covey. *HOOPS Reference Manual, Version 3.0.* Ithaca Software, 1991.

[114] T.F. Wiegand. Interactive Rendering of CSG Models. *Computer Graphics forum* 15(4):249-261, 1996.

[115] G. Winkenbach, D.H. Salesin. Computer-Generated Pen-and-Ink Illustrations. *Computer Graphics (SIGGRAPH ’94 Proceedings)*, 91-100, 1994.

[116] P. Wisskirchen. *Object-Oriented Graphics. From GKS and PHIGS to Object-Oriented Systems*. Springer-Verlag, 1990.

[117] H. Züllighoven. *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. dpunkt.verlag, 1998.