Entwurf und Implementierung eines computergraphischen Systems zur Integration komplexer, echtzeitfähiger 3D-Renderingverfahren

Von der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam
zur Erlangung des akademischen Grades
„doctor rerum naturalium“ (Dr. rer. nat.)
in der Wissenschaftsdisziplin Informatik
angenommene Dissertation

Florian Kirsch
geb. am 12. Februar 1976
in Delmenhorst

Tag der Disputation: 20. Oktober 2005

DANKSAGUNG

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter des Fachgebiets für Computergraphische Systeme am Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam in der Zeit von April 2001 bis April 2005.

An dieser Stelle möchte ich mich bei Prof. Dr. Jürgen Döllner für seine stetigen fachlichen Anregungen zur Forschung in die verschiedensten Richtungen, für die Betreuung dieser Arbeit, die Übernahme des Gutachtens und für die gute Zusammenarbeit sehr herzlich bedanken.

Weiterhin bedanke ich mich bei Prof. Dr. Klaus Rebensburg (Universität Potsdam) und Prof. Dr. Guido Wirtz (Universität Bamberg), ebenfalls für die Übernahme des Gutachtens, sowie bei Prof. Dr. Klaus Hinrichs (Westfälische-Wilhelms-Universität Münster) für die Bereitsschaft dazu.

Ein besonderer Dank gilt meinen derzeitigen und früheren Kollegen meiner Arbeitsgruppe am HPI, Dr. Konstantin Baumann, Hans Bohnet, Henrik Buchholz, Dr. Oliver Kersting, Haik Lorenz, Stefan Maaß und Marc Nienhaus, für viele fruchtbare Diskussionen und für die Weiterentwicklung der Graphikbibliothek VRS. Speziell bei Henrik Buchholz und Marc Nienhaus möchte ich mich bedanken für die Zusammenarbeit im Rahmen gemeinsamer Projekte, nämlich bei der Transparenzdarstellung von Alternativen in einem Stadtmodell bzw. bei der Entwicklung eines Algorithmus für bildbasiertes, transparentes CSG.

Dem Studenten Stephan Brumme danke ich für die Erlaubnis zur Nutzung einiger 3D-Modelle von Schachfiguren, die er im Rahmen eines Seminars entwickelt hat.

Der ObjektScan GmbH, Potsdam, Thomas Bauer und Dr. Ralf König danke ich für die Erlaubnis zur Nutzung des 3D-Modells eines prähistorischen Schädels.

Nicht zuletzt bedanke ich mich ganz besonders bei meiner Frau Heike.

Berlin, im November 2005

Florian Kirsch

ZUSAMMENFASSUNG

Thema dieser Arbeit sind echtzeitfähige 3D-Renderingverfahren, die 3D-Geometrie mit über der Standarddarstellung hinausgehenden Qualitäts- und Gestaltungsmerkmalen rendern können. Beispiele sind Verfahren zur Darstellung von Schatten, Reflexionen oder Transparenz. Mit heutigen computergraphischen Software-Basissystemen ist ihre Integration in 3D-Anwendungssysteme sehr aufwändig: Dies liegt einerseits an der technischen, algorithmischen Komplexität der Einzelverfahren, andererseits an Ressourcenkonflikten und Seiteneffekten bei der Kombination mehrerer Verfahren. Szenengraphsysteme, intendiert als computergraphische Softwareschicht zur Abstraktion von der Graphikhardware, stellen derzeit keine Mechanismen zur Nutzung dieser Renderingverfahren zur Verfügung.

Ziel dieser Arbeit ist es, eine Software-Architektur für ein Szenengraphsystem zu konzipieren und umzusetzen, die echtzeitfähige 3D-Renderingverfahren als Komponenten modelliert und es damit erlaubt, diese Verfahren innerhalb des Szenengraphsystems für die Anwendungsentwicklung effektiv zu nutzen. Ein Entwickler, der ein solches Szenengraphsystem nutzt, steuert diese Komponenten durch Elemente in der Szenenbeschreibung an, die die sichtbare Wirkung eines Renderingverfahrens auf die Geometrie in der Szene angeben, aber keine Hinweise auf die algorithmische Implementierung des Verfahrens enthalten. Damit werden Renderingverfahren in 3D-Anwendungssystemen nutzbar, ohne dass ein Entwickler detaillierte Kenntnisse über sie benötigt, so dass der Aufwand für ihre Entwicklung drastisch reduziert wird.

Ein besonderer Augenmerk der Arbeit liegt darauf, auf diese Weise auch verschiedene Renderingverfahren in einer Szene kombiniert einsetzen zu können. Hierzu ist eine Unterteilung der Renderingverfahren in mehrere Kategorien erforderlich, die mit Hilfe unterschiedlicher Ansätze ausgewertet werden. Dies erlaubt die Abstimmung verschiedener Komponenten für Renderingverfahren und ihrer verwendeten Ressourcen.

Die Zusammenarbeit mehrerer Renderingverfahren hat dort ihre Grenzen, wo die Kombination von Renderingverfahren graphisch nicht sinnvoll ist oder fundamentale technische Beschränkungen der Verfahren eine gleichzeitige Verwendung unmöglich machen. Die in dieser Arbeit vorgestellte Software-Architektur kann diese Grenzen nicht verschieben, aber sie ermöglicht den gleichzeitigen Einsatz vieler Verfahren, bei denen eine Kombination aufgrund der hohen Komplexität der Implementierung bislang nicht erreicht wurde. Das Vermögen zur Zusammenarbeit ist dabei allerdings von der Art eines Einzelverfahrens abhängig: Verfahren zur Darstellung transparenter Geometrie beispielsweise erfordern bei der Kombination mit anderen Verfahren in der Regel vollständig neuentwickelte Renderingverfahren; entsprechende Komponenten für das Szenengraphsystem können daher nur eingeschränkt mit Komponenten für andere Renderingverfahren verwendet werden.

Das in dieser Arbeit entwickelte System integriert und kombiniert Verfahren zur Darstellung von Bumpmapping, verschiedene Schatten- und Reflexionsverfahren sowie bildbasiertes CSG-Rendering. Damit stehen wesentliche Renderingverfahren in einem Szenengraphsystem erstmalig komponentenbasiert und auf einem hohen Abstraktionsniveau zur Verfügung. Das System ist trotz des zusätzlichen Verwaltungsaufwandes in der Lage, die Renderingverfahren einzeln und in Kombination grundsätzlich in Echtzeit auszuführen.

ABSTRACT

This thesis is about real-time rendering algorithms that can render 3D-geometry with quality and design features beyond standard display. Examples include algorithms to render shadows, reflections, or transparency. Integrating these algorithms into 3D-applications using today's rendering libraries for real-time computer graphics is exceedingly difficult: On the one hand, the rendering algorithms are technically and algorithmically complicated for their own, on the other hand, combining several algorithms causes resource conflicts and side effects that are very difficult to handle. Scene graph libraries, which intend to provide a software layer to abstract from computer graphics hardware, currently offer no mechanisms for using these rendering algorithms, either.

The objective of this thesis is to design and to implement a software architecture for a scene graph library that models real-time rendering algorithms as software components allowing an effective usage of these algorithms for 3D-application development within the scene graph library. An application developer using the scene graph library controls these components with elements in a scene description that describe the effect of a rendering algorithm for some geometry in the scene graph, but that do not contain hints about the actual implementation of the rendering algorithm. This allows for deploying rendering algorithms in 3D-applications even for application developers that do not have detailed knowledge about them. In this way, the complexity of development of rendering algorithms can be drastically reduced.

In particular, the thesis focuses on the feasibility of combining several rendering algorithms within a scene at the same time. This requires to classify rendering algorithms into different categories, which are, each, evaluated using different approaches. In this way, components for different rendering algorithms can collaborate and adjust their usage of common graphics resources.

The possibility of combining different rendering algorithms can be limited in several ways: The graphical result of the combination can be undefined, or fundamental technical restrictions can render it impossible to use two rendering algorithms at the same time. The software architecture described in this work is not able to remove these limitations, but it allows to combine a lot of different rendering algorithms that, until now, could not be combined due to the high complexities of the required implementation. The capability of collaboration, however, depends on the kind of rendering algorithm: For instance, algorithms for rendering transparent geometry can be combined with other algorithms only with a complete redesign of the algorithm. Therefore, components in the scene graph library for displaying transparency can be combined with components for other rendering algorithms in a limited way only.

The system developed in this work integrates and combines algorithms for displaying bump mapping, several variants of shadow and reflection algorithms, and image-based CSG algorithms. Hence, major rendering algorithms are available for the first time in a scene graph library as components with high abstraction level. Despite the required additional indirections and abstraction layers, the system, in principle, allows for using and combining the rendering algorithms in real-time.

INHALTSVERZEICHNIS

Kapitel 1

SOFTWARE-ENGINEERING IN DER ECHTZEIT-COMPUTERGRAPHIK

> *„**Systems integration.** This is the problem of keeping all the balls in the air at once, that is, how to use all the tricks in one production. Just because one researcher can do cloth, one can do faces, and one can do hair doesn't mean that all animation system systems can suddenly put them all together.“*
>
> Jim Blinn, „Ten More Unsolved Problems in Computer Graphics“, 1998, [7].

Unsere wahrnehmbare Umwelt ist von enormer visueller Komplexität. Seit den Anfängen der Computergraphik ist eines ihrer großen Zeile, ein möglichst photorealistisches, digitales 2D-Abbild einer 3D-Szene erzeugen zu können. Mittlerweile ist in diesem Gebiet ein hoher Grad an Perfektion erreicht: Heute können einzelne Bilder erzeugt werden, bei denen ein Mensch die Entscheidung, ob es sich um eine photoreale oder um eine computergenerierte Abbildung handelt, nicht mehr ohne weiteres treffen kann. Die Ergebnisse sind einer breiten Öffentlichkeit zugänglich, beispielsweise durch vollständig computeranimierte Trickfilmproduktionen oder computergenerierte Spezialeffekte in Realfilmen.

Alle Algorithmen zur Berechnung photorealistischer Bilder, wie *Radiosity*, *Ray-Tracing* und *Photon-Mapping*, haben sehr hohe Rechenzeitanforderungen. Für die Bilderzeugung in interaktiven Anwendungen werden sie daher nicht eingesetzt. Hier verwendet man unter Nutzung spezialisierter Graphikhardware *Echtzeitrendering*. Eine Definition dieses Begriffs ist nach Akenine-Möller und Haines [3] nicht exakt möglich: Sie geben eine Bilderzeugungsrate von etwa 6

Bildern pro Sekunde (frames per second, fps) an, ab der Interaktion mit einer Anwendung möglich wird. Bei etwa 15 fps verschwindet beim Menschen der subjektive Eindruck von Einzelbildern, und bei etwa 72 fps liegt die Wahrnehmungsschwelle, ab der keine Verbesserung der Qualität einer graphischen Animation mehr erkennbar ist.

1.1 Echtzeitfähige Renderingverfahren

Echtzeitrendering wird heute fast ausschließlich mit spezialisierter Graphikhardware realisiert, die als zentrales Leistungsmerkmal eine *Rasterisierung* und Texturierung von Dreiecken in das Bildraster vornimmt und die Farben der entstehenden Fragmente in den Bildspeicher überträgt. Entsprechende Hardware zählt seit einigen Jahren zur Standardausstattung von PCs.

Im Vergleich zu nicht echtzeitfähigen Renderingalgorithmen, die globale Beleuchtungsmodelle verwenden, nutzt man im Echtzeitrendering ein *lokales Beleuchtungsmodell*. Dabei wird die Färbung eines Dreiecks ausschließlich anhand der Position und Farbe von virtuellen Lichtquellen in der Szene sowie mit Hilfe von Texturen und Materialparametern berechnet, andere Dreiecke in der Szene beeinflussen die Beleuchtungsberechnung dagegen nicht. Der Vorteil ist, dass so eine parallele Verarbeitung mehrerer Dreiecke durch die Graphikhardware möglich ist und dadurch eine drastische, Echtzeitrendering ermöglichende Beschleunigung erreicht wird.

Renderingeffekte und Renderingverfahren

Der Nachteil eines lokales Beleuchtungsmodells ist die gegenüber photorealistischen Verfahren stark verminderte Darstellungsqualität. In dieser Arbeit bezeichnet der Begriff *Renderingeffekt* ein ausgezeichnetes, einzeln identifizierbares, optisches Qualitätsmerkmal bei der Darstellung von 3D-Geometrie. Phänomene wie Schatten, spiegelnde Oberflächen und Lichtbrechungen sind Beispiele für Renderingeffekte. Ihr Fehlen ist ein typischer, sofort erkennbarer Mangel beim Echtzeitrendering.

Um einzelne Renderingeffekte im Echtzeitrendering approximieren zu können, gibt es eine Reihe von *Renderingverfahren*. Dieser Begriff bezeichnet in dieser Arbeit einen Algorithmus zur Erzeugung eines Renderingeffekts durch Konfiguration von Einstellungen der Graphikhardware. In der Regel gibt es zur Darstellung einer Art von Renderingeffekt verschiedene, manchmal in der Implementierung fundamental unterschiedliche Renderingverfahren.

Die Entwicklung von Renderingverfahren ist eines der Hauptarbeitsgebiete in der Computergraphik. In echtzeit-computergraphischer Anwendungssoftware wurden sie dabei bis vor einigen Jahren wenig eingesetzt, weil die damalige Graphikhardware schon bei Verwendung von Standardschattierungsverfahren ausgelastet war. Mit der Explosion der Leistung der Graphikhardware und der nur im geringen Maße gestiegenen Bildschirmauflösung stehen heute pro Pixel weitaus mehr Rechenoperationen zur Verfügung. Außerdem können durch die Entwicklung programmierbarer Shader (Abschnitt 2.3) bei der Verarbeitung eines Dreiecks Berechnungen auf der Graphikhardware weitaus flexibler vorgenommen werden. Echtzeitfähige Renderingverfahren im Sinne dieser Arbeit sind damit in realen Anwendungen einsetzbar geworden.

Neue Renderingverfahren werden in der Regel zunächst als Machbarkeitsstudien, das heißt durch minimale Beispielanwendungen implementiert. Aus Sicht des Software-Engineerings ist damit die Frage, wie die Verfahren in praktische Anwendungen integriert werden können, noch nicht beantwortet. Annahmen über die Geometrie oder die Konfiguration der Graphikhardware, die in einer Machbarkeitsstudie vorausgesetzt werden können, sind in einer allgemeinen Anwendung nicht ohne weiteres gültig. Zudem ist die Implementierung eines Verfahrens durch die Anzahl der Konfigurationsoptionen der Graphikhardware, die passend zueinander eingestellt werden müssen, ohnehin kompliziert und fehleranfällig.

Kombination von Renderingverfahren

Das Zitat von Jim Blinn zu Beginn dieses Kapitels konstatiert ein vernachlässigtes Problem der Computergraphik: Es ist softwaretechnisch bedeutend einfacher, verschiedene Renderingverfahren isoliert voneinander zu entwickeln, als diese Renderingverfahren zu kombinieren und gleichzeitig einzusetzen.

Es treten bei der Kombination von Renderingverfahren zwei Probleme auf: Zum einen beeinflussen sich verschiedene Renderingeffekte in ihrer Wirkung gegenseitig. Zum anderen benötigen verschiedene Renderingverfahren vielfach die gleichen Ressourcen der Graphikhardware und sind damit nicht ohne weiteres zusammen einsetzbar. Diese Fragen des Software-Engineerings sind bisher in der Computergraphik nicht ausreichend bearbeitet worden.

1.2 APIs für die Entwicklung von Echtzeit-Computergraphik

OpenGL [73] und *Direct3D* [31] sind die beiden relevanten APIs zur systemnahen Programmierung der Graphikhardware. Sie werden als *Immediate-Mode Renderingsysteme* bezeichnet, weil sie keine Informationen über Dreiecke intern vorhalten, sondern jedes übermittelte Dreieck sofort verarbeiten. Daneben stellen sie aus Sicht des Anwendungsentwicklers eine Menge von Funktionen zur Konfiguration der Graphikhardware bereit, die einstellen, auf welche Weise die Dreiecke verarbeitet werden.

Die wachsende Komplexität von 3D-Graphikanwendungen erfordert es zunehmend, Methoden des Software-Engineerings zu ihrer Entwicklung einzusetzen. Eine direkte Entwicklung mit einem Immediate-Mode Renderingsystem wird für immer weniger Projekte genutzt. Stattdessen bildet sich zunehmend eine Softwareschicht zwischen Immediate-Mode Renderingsystem und Anwendungssoftware, die dem Anwendungsentwickler häufig benötigte Lösungen für wiederkehrende Anforderungen der Computergraphik zur Verfügung stellt.

Der Spezialisierungsgrad dieser Softwareschicht unterscheidet sich je nach Anwendungsfall. Auf der einen Seite dienen *Renderingbibliotheken* zur graphisch optimierten Darstellung beispielsweise von Bäumen oder medizinischen Volumendaten. Sie kapseln in der Regel hochspezialisierte Renderingalgorithmen durch eine vereinfachte Schnittstelle. Im Gegensatz dazu bieten *Graphiksysteme* die Infrastruktur, um vollständige computergraphische Anwendungen zu entwickeln. Hier unterscheidet man zwischen speziellen Graphiksystemen, wie zum Beispiel für Computerspiele oder VR-Anwendungen [4], und allgemeinen Graphiksystemen. *Spezielle Graphiksysteme* treffen eine Reihe von Annahmen über die darzustellende Graphik und die zu erwartenden Anwendungsszenarien. Beispielsweise sind Computerspiele-Engines oft auf die Darstellung von Außen- oder Innenarealen spezialisiert, implementieren speziell dafür geeignete Renderingalgorithmen und bilden außerdem ausgewählte, für das jeweilige Genre relevante physikalische Gesetze in der simulierten Welt nach.

Allgemeine Graphiksysteme haben keinen derartig fokussierten Einsatzzweck. Ihre Stärke liegt im Gegenteil darin, dass sie für grundsätzlich alle Arten von Anwendungen verwendbar sind. Die Struktur von allgemeinen Graphiksystemen ist in der Regel die folgende: Sie enthalten eine zentrale, hierarchische Datenstruktur, den *Szenengraph*, die die darzustellende 3D-Szene beschreibt; aufgrund dieser internen Datenhaltung der Szene bezeichnet man allgemeine Graphiksysteme auch als *Retained-Mode Renderingsysteme*. Zur Darstellung der Szene wird diese Datenstruktur traversiert, und dabei werden die benötigten Funktionen des Immediate-Mode Renderingsystems oder gegebenenfalls einer Renderingbibliothek zum Rendern der Szene aufgerufen. Abbildung 1 zeigt das Schichtenmodell einer 3D-Graphikanwendung, die ein allgemeines Graphiksystem nutzt.

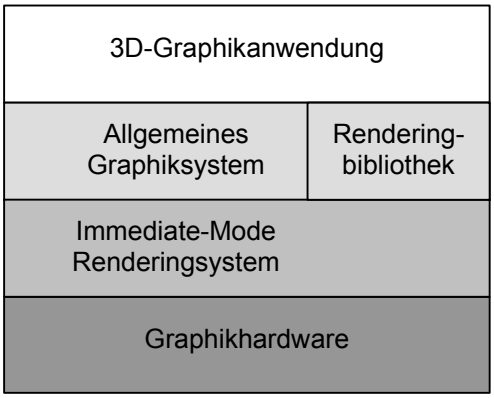

Abbildung 1: Schichtenmodell einer 3D-Graphikanwendung bei Verwendung eines allgemeinen Graphiksystems.

Ein Hauptinteresse allgemeiner Graphiksysteme ist es, durch Abstraktion vom Immediate-Mode Renderingsystem die Entwicklung von Graphikanwendungen zu vereinfachen. Damit liegt es nahe, dass sie insbesondere den Zugang zu Renderingverfahren erleichtern sollten, um eine einfache Verwendung der entsprechenden Renderingeffekte in praktischen Anwendungen zu ermöglichen. Stattdessen ist in diesem Gebiet ein großer Mangel zu verzeichnen: Allgemeine Graphiksysteme unterstützen den Anwendungsentwickler bei der Verwendung der meisten Renderingeffekte nicht. Falls überhaupt möglich, muss ein zugehöriges Renderingverfahren auf der gleichen Abstraktionsebene wie bei direkter Nutzung des Immediate-Mode Renderingsystems implementiert werden. Damit wird die Entwicklung der Verfahren nicht vereinfacht und erfordert nach wie vor fundamentale Kenntnisse über die Funktionsweise der Graphikhardware und des Renderingsystems. Dieser Mangel ist vermutlich der Grund, dass Renderingeffekte wie Schatten und Reflexionen in den meisten computergraphischen Anwendungen – mit Ausnahme von Computerspielen, die aber meist spezielle Graphiksysteme einsetzen – nicht verwendet werden.

1.3 Ziele der Arbeit

Das Ziel dieser Arbeit ist es, die Nutzung von Renderingeffekten bei der Entwicklung computergraphischer Anwendungen zu vereinfachen und die Kombination mehrerer Renderingeffekte nachvollziehbar zu ermöglichen. Ausgangsbasis sind allgemeine Graphiksysteme, denn diese sind aufgrund ihres Anspruchs, die Entwicklung von 3D-Anwendungen zu erleichtern, auf natürliche Weise dafür prädestiniert. Die folgenden Fragestellungen werden betrachtet:

- Nicht alle Renderingeffekte und zugehörige Renderingverfahren können potentiell zusammenarbeiten. Für einige Renderingeffekte muss explizit definiert werden, in welcher Form sie sich gegenseitig beeinflussen.
- Bisherige Graphiksysteme ermöglichen es nicht, die Deklaration eines Renderingeffekts von der Auswertung durch ein entsprechendes Renderingverfahren zu trennen. Dieses soll in einem allgemeinen Graphiksystem ermöglicht werden, damit die Deklaration durch den Entwickler einer Anwendung erfolgen kann und die Implementierung des zugehörigen Renderingverfahrens durch das allgemeine Graphiksystem bereitgestellt wird.
- Es ist zu klären, mit welchen Ausdrucksmitteln eines allgemeinen Graphiksystems die Deklaration von Renderingeffekten als Teil der Beschreibung der darzustellenden Szene erfolgen kann.

- Es ist zu klären, welche Konzepte für die Auswertung von Renderingeffekten in einem allgemeinen Graphiksystem nötig sind. Durch komponentenbasierte Modellierung der Renderingverfahren soll insbesondere die Auswertung mehrerer Renderingeffekte gleichzeitig ermöglicht werden.
- Die entwickelte Software-Architektur wird anhand einer Reihe von Renderingeffekten praktisch erprobt. Hierzu werden Oberflächenshader, Schatten- und Reflexionsverfahren integriert, daneben auch spezielle Verfahren wie bildbasiertes CSG-Rendering.

Als zugrundeliegendes allgemeine Graphiksystem wird das *Szenengraphsystem* VRS verwendet [18]. Diese Wahl liegt insofern nahe, weil der Fokus bei diesem Graphiksystem nicht in erster Linie auf Renderinggeschwindigkeit, sondern auf einfache Verwendung liegt, und weil VRS das Konzept der Trennung von Deklaration und Auswertung, beispielsweise für Geometrie, seit jeher verfolgt. Die Ergebnisse der Arbeit können dennoch auf andere allgemeine Graphiksysteme übertragen werden.

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich wie folgt:

- Kapitel 2 befasst sich mit dem Aufbau der Renderingpipeline in einem Immediate-Mode Renderingsystem und bespricht die technischen Grundlagen, die für die Entwicklung von Renderingverfahren erforderlich sind.
- Kapitel 3 beschreibt und kategorisiert verschiedene Renderingeffekte und erläutert die wichtigsten Verfahren zu ihrer Umsetzung. Neben bekannten Renderingverfahren werden auch in der Arbeit neuentwickelte vorgestellt. Am Ende des Kapitels werden Schwierigkeiten bei der Deklaration und Auswertung kombinierter Renderingeffekte beschrieben.
- Kapitel 4 enthält eine Einführung des Szenengraphen, der zentralen Datenstruktur zur Beschreibung einer Szene in Szenengraphsystemen.
- Kapitel 5 beschreibt und bewertet Möglichkeiten zur Deklaration von Renderingeffekten in einer Szenenbeschreibung.
- Kapitel 6 beschreibt Konzepte, um in einer Szenenbeschreibung deklarierte Renderingeffekte auszuwerten. Verschiedene Ansätze sind erforderlich, um unterschiedliche Typen von Renderingverfahren zu integrieren. Dies ermöglicht auch die Kombination mehrerer Renderingverfahren zur kombinierten Auswertung unterschiedlicher Renderingeffekte.
- Kapitel 7 befasst sich mit bildbasierten Verfahren der konstruktiven Festkörpergeometrie, das heißt mit bildbasiertem CSG-Rendering. Es werden Renderingverfahren und Geschwindigkeitsoptimierungen vorgestellt. Die gewonnenen Algorithmen sind der Kern einer Renderingbibliothek für bildbasiertes CSG-Rendering, die in VRS verwendet wird. Das Kapitel schließt mit der Kombination des Effekts zusammen mit Schatten, Reflexionen und Transparenz.
- Kapitel 8 fasst die Ergebnisse der Arbeit zusammen und skizziert weiterführende Aspekte.

Kapitel 2

BASISTECHNIKEN FÜR ECHTZEITFÄHIGE RENDERINGVERFAHREN

Immediate-Mode Renderingsysteme sind die systemnahesten APIs zur portablen Programmierung der Graphikhardware. Ihr Aufbau ist dadurch gekennzeichnet, dass die darzustellende Geometrie als Dreiecksinformation prozedural an das API übergeben wird, wo ihre Verarbeitung in der *Renderingpipeline* erfolgt, um die von den Dreiecken überdeckten Pixel zu aktualisieren. Alle Stationen der Renderingpipeline können über die Menge aller Renderingoptionen, den *Renderingkontext* als internen Bestandteil des Renderingsystems, vielfältig konfiguriert werden. Ein Immediate-Mode Renderingsystem hält intern keine Geometriedaten vor – bzw. mit *Display-Listen* allenfalls statische, in der Anwendung nicht mehr zugängliche Geometriedaten.

Dieses Kapitel beschreibt den allgemeinen Aufbau der Renderingpipeline und auf dieser Grundlage Basistechniken, die bei der Entwicklung echtzeitfähiger Renderingverfahren häufig benötigt werden, beispielsweise die dynamische Erzeugung von Texturen. Die Darstellung erfolgt auf Grundlage des Immediate-Mode Renderingsystems OpenGL [73].

2.1 Festverdrahtete Renderingpipeline

Die Renderingpipeline ist eine mehrstufige Verarbeitungskette, in der die darzustellenden Daten in der *Applikationsstufe* von der Anwendung gehalten und aufbereitet, als Geometriedaten in der *Geometriestufe* transformiert, und nach der Rasterisierung in Pixeldaten durch die *Rasterisierungsstufe* in den Bildspeicher, den *Framebuffer* übernommen werden [25].

Applikationsstufe

Die Applikationsstufe liegt unter der Verantwortung des Anwendungsentwicklers, das heißt, sie wird nicht vom Immediate-Mode Renderingsystem implementiert. Aufgabe der Applikationsstufe ist die Datenhaltung der darzustellenden Szene und die Konfiguration des Renderingkon-

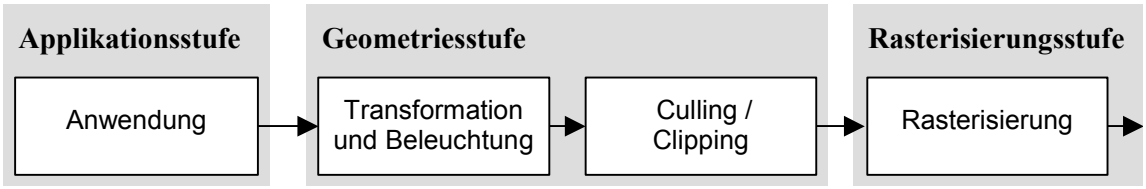

Abbildung 2: Aufbau der Renderingpipeline (Teil 1).

textes der Graphikhardware, das heißt der nachfolgenden Abschnitte der Renderingpipeline. Zum Auslösen der Verarbeitung von Geometrie durch die Renderingpipeline werden in der Applikationsstufe die Eckpunktinformationen der Geometrie an die Geometriestufe übergeben.

Geometriestufe: Transformation und Beleuchtung

Die Geometriestufe wird vom Renderingsystem implementiert. In dieser Stufe erfolgt die Verarbeitung der Eckpunktinformationen, das heißt von jeweils einem *Vertex*. Zunächst erfolgt die *Transformation und Beleuchtung* (transform and lighting) der Eckpunkte: Jeder Eckpunkt wird dabei durch eine Abfolge geometrischer Transformationen zunächst in das *Kamerasichtvolumen*, das sogenannte *View-Frustum* transformiert, und von dort in einen normierten Quader, das *kanonische Sichtvolumen* projiziert. Als interne Zwischenarbeitsschritte pro Eckpunkt fallen je nach Einstellung des Renderingkontextes die Beleuchtungsberechnung anhand idealisierter Lichtquellen und die Generierung oder Transformation von Texturkoordinaten an. Bei der festverdrahteten Renderingpipeline sind die Funktionsvorschriften, nach denen diese Berechnungen durchgeführt werden, lediglich innerhalb enger Grenzen konfigurierbar. Beispielsweise wird die Beleuchtung nach dem *Blinn-Phong Beleuchtungsmodell* anhand einer vorgegebenen Funktionsvorschrift berechnet [62][5], an der sich lediglich die konstanten Parameter ändern lassen.

Geometriestufe: Culling und Clipping

Die bis hierhin isoliert voneinander behandelten, aufeinanderfolgenden, transformierten Eckpunkte werden jetzt als zusammengehörige Dreiecke betrachtet. Zunächst erfolgt das *Culling*, das heißt das Weglassen von Dreiecken, die in der x/y-Ebene des kanonischen Sichtvolumens im bzw. gegen den Uhrzeigersinn orientiert sind. Bei einer konsistent orientierten, geschlossenen Geometrie können so die rückseitigen Dreiecke von den vorderseitigen Dreiecken unterschieden und die ohnehin nicht sichtbaren Rückseiten in der Folge vernachlässigt werden. Dies bezeichnet man als *Back-face Culling*, im Gegensatz zu *Front-face Culling*, das für die Implementierung einiger Renderingverfahren ebenfalls erforderlich ist. Das anschließende *Clipping* dient zum Zuschneiden der Dreiecke mit dem Ziel, nur noch Geometrie innerhalb des kanonischen Sichtvolumens weiter zu verarbeiten. Das heißt für das View-Frustum, dass es nach vorne und hinten begrenzt ist und damit Geometrie im Koordinatensystem der Kamera nur zwischen der *Near-clipping Ebene* und der *Far-clipping Ebene* angezeigt wird.

Rasterisierungsstufe: Rasterisierung

Die Dreiecke gelangen nun in die Rasterisierungsstufe. Dort erfolgt zunächst die *Rasterisierung*, das heißt die Aufteilung eines Dreiecks in einzelne *Fragmente*, die jeweils einem darzustellenden Bildpunkt entsprechen. Ein Fragment enthält die folgenden Informationen: einen Farbwert und Texturkoordinaten, die sich durch Interpolation der entsprechenden Werte der Eckpunkte eines Dreiecks ergeben, sowie Tiefe und – implizit – die x/y-Position des Fragments im Framebuffer.

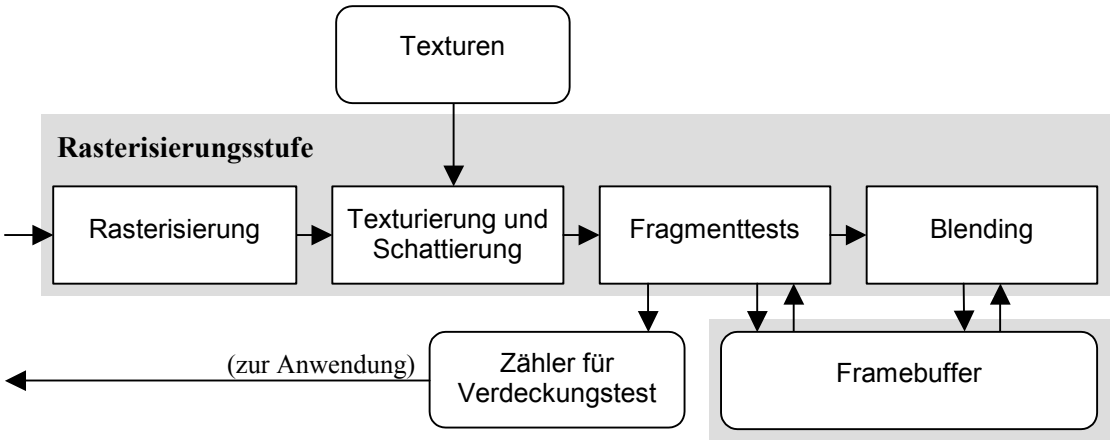

Abbildung 3: Aufbau der Renderingpipeline (Teil 2).

Rasterisierungsstufe: Texturierung und Schattierung

Die Fragmente werden anschließend texturiert und schattiert: Zu der Texturkoordinate wird der entsprechende Texturwert aus einer *Textur* ermittelt und mit dem bisherigen Farbwert des Fragments verknüpft. In der festverdrahteten Renderingpipeline wird aus einer kleinen Anzahl verschiedener Verknüpfungsoperationen ausgewählt, nämlich der Ersetzung des bisherigen Farbwerts mit der Texturwert, der komponentenweisen Multiplikation des Farb- mit dem Texturwert, oder dem Mischen des Farb- und Texturwerts abhängig vom Alpha des Texturwerts.

Mehrfachtexturierung (multi texturing) führt diese Texturierungsoperation für verschiedene Texturkoordinaten und Texturen nacheinander durch. Die Abfolge der Texturen ist dabei in der festverdrahteten Renderingpipeline linear, und die Verknüpfungsoperation kann pro Textur getrennt eingestellt werden.

Das Ergebnis der Texturierung und Schattierung für ein Fragment ist der resultierende Farbwert sowie der Tiefenwert des Fragments. Ob und mit welchem Anteil diese Werte in den Framebuffer übernommen werden, hängt von dessen Einstellungen ab.

2.2 Funktionen des Framebuffers

Der Framebuffer setzt sich zusammen aus *Farb-*, *Tiefen-*, *Stencil-* und *Akkumulationspuffer*. Der Farbpuffer wird zusätzlich unterteilt in einen *RGB-Puffer*, der auf dem Monitor zur Anzeige kommt, und in einen vielfältig nutzbaren, nicht sichtbaren *Alphapuffer*.

Farb-, Tiefen-, und Stencilpuffer sind das Ziel der Renderingpipeline. Der Akkumulationspuffer unterscheidet sich hinsichtlich seiner Integration in die Renderingpipeline grundlegend von den übrigen Teilen des Framebuffers und wird daher in Abschnitt 2.4 behandelt.

Rasterisierungsstufe: Fragmenttests

Der Farb- und der Tiefenwert eines Fragments wird nur dann im Framebuffer abgelegt, wenn eine Reihe von *Fragmenttests* erfüllt sind, nämlich Scissor-, Alpha-, Tiefen-, und Stenciltest. Diese Fragmenttests können einzeln zu- oder abgeschaltet werden.

Durch den *Scissortest* werden Veränderungen des Framebuffers auf einen rechtwinkligen Bereich beschränkt. Nur Fragmente innerhalb dieses Bereichs werden für die weiteren Tests betrachtet. Der folgende *Alphatest* erlaubt es, eingehende Fragmente abhängig von ihrem Alphawert zu verwerfen. Der *Tiefentest* ermöglicht Verdeckungsermittlung (hidden surface removal)

mit Hilfe des Tiefenpuffers und des Z-Buffer-Algorithmus. Anstelle der Standard-Vergleichsfunktion „kleiner-als“, mit der der Tiefenwert eines rasterisierten Fragments mit dem im Tiefenpuffer abgelegten Wert verglichen wird, können auch andere Vergleichsfunktionen verwendet werden. Der *Stenciltest* schließlich erlaubt es, die Übernehme eines Fragments in den Framebuffer zusätzlich abhängig zu machen von einer boolschen Operation eines konstanten Stencilwerts mit dem entsprechenden Wert im Stencilpuffer. Dabei kann der Wert im Stencilpuffer durch eine Anzahl verschiedener Operationen modifiziert werden.

Rasterisierungsstufe: Übernahme in der Framebuffer, Blending

Sind alle Tests geglückt, wird das Fragment in den Framebuffer übernommen, wobei allerdings Schreiben in den Tiefenpuffer, in jede Bitebene des Stencilpuffers und in jeden Kanal des Farbpuffers einzeln maskiert werden kann. Ein Wert im Farbpuffer wird normalerweise einfach mit dem Farbwert des entsprechenden Fragments überschrieben. *Blending* ermöglicht hingegen das Hinzuaddieren bzw. Mischen des Farbwerts eines Fragments mit dem abgelegten Farbwert im Farbpuffer. Das Mischanteile werden dabei durch die *Blendingfunktion* festgelegt und sind meist abhängig von den Alphawerten des Fragments – dann spricht man von *Alpha-Blending* – oder im Alphapuffer.

Rasterisierungsstufe: Verdeckungstests

Verdeckungstests (occlusion queries) dienen zum Erfragen, wie viele Fragmente bei der Verarbeitung einer Menge von Dreiecken in den Framebuffer übernommen wurden. Vor und nach dem Aufruf zur Übertragung der Geometrie wird die Zählung durch entsprechende Funktionen gestartet bzw. wieder gestoppt. Danach kann das Ergebnis abgefragt werden. Zur Unterstützung einer asynchronen Arbeitsteilung der CPU und der Graphikhardware kann außerdem überprüft werden, ob das Ergebnis des Verdeckungstests bereits vorliegt. Falls nicht, muss die CPU nicht auf das Ergebnis des Tests warten, sondern kann unterdessen andere Aufgaben bearbeiten.

Buffer-Region-Erweiterung

Der Framebuffer stellt eine nur einmalig vorhandene Ressource der Renderingpipeline dar, die aber von jedem Renderingverfahren benötigt wird – denn zumindest der Farbpuffer wird ja durch ein Renderingverfahren modifiziert. Mit der *Buffer-Region*-Erweiterung können Farb-, Tiefen- oder Stencilpuffer jeweils mehrmals erzeugt werden, wobei immer eine Instanz dieser Puffer zum Rendern verwendet wird. Damit wird beispielsweise eine Verwaltung der Puffer auf einem Stack ermöglicht.

Die Buffer-Region-Erweiterung ist ein Beispiel einer nicht standardisierten *OpenGL-Erweiterung* (OpenGL extension, [59]). Dies sind herstellerspezifische Ergänzungen von OpenGL, die dem API experimentelle oder nicht-portable Funktionen hinzufügen. Daher wird die Buffer-Region-Erweiterung auf verbreiteter Hardware wie zum Beispiel von ATI in absehbarer Zeit nicht verfügbar sein. Für die Entwicklung portabler Software ist damit derzeit ihre Verwendung nicht möglich.

2.3 Programmierung der Renderingpipeline

Lange Zeit wurden neue Renderingfähigkeiten der Graphikhardware in OpenGL integriert, indem durch eine OpenGL-Erweiterung eine partikulare weitere Berechnungsfunktion in die festverdrahtete Renderingpipeline aufgenommen wurde. Ein typisches Beispiel ist der *texturbasierte Schattentest* zur Unterstützung von Shadow-Mapping (Abschnitt 3.4.2): Dabei wird beim Auslesen des Texturwerts aus einer Tiefentextur die r-Komponente der Texturkoordinate, die die Entfernung eines Fragments zur Lichtquelle enthält, mit dem entsprechenden Wert in der

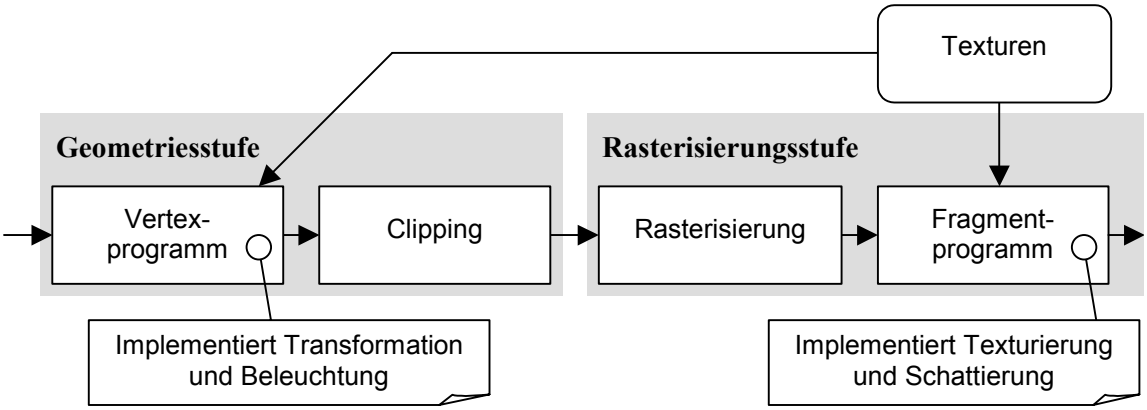

Abbildung 4: Ausschnitt der Renderingpipeline mit programmierbaren Shadern.

Tiefentextur verglichen. Falls die Texturkoordinate kleiner ist, ergibt sich ein Texturwert von 0, ansonsten von 1, wodurch Schatten und beleuchtete Gebiete gekennzeichnet sind.

Programmierbare Shader

Diese und viele andere Erweiterungen bedingen insgesamt ein unübersichtliches Programmiermodell und sind außerdem oft nicht über Herstellergrenzen hinweg portabel. Mit der programmierbaren Renderingpipeline gibt es seit 2001 ein alternatives Programmiermodell. *Vertexprogramme* ersetzen die Transformation und Beleuchtung der festverdrahteten Renderingpipeline durch eine benutzerimplementierte Berechnung [48]. *Fragmentprogramme* ersetzen entsprechend die festverdrahtete Texturierung und Schattierung. Der Anwendungsentwickler implementiert in diesen Programmen die jeweiligen benötigten Funktionen und übergibt sie dem Renderingsystem als String. Dieses kompiliert die Programme, damit die Graphikhardware sie in der Geometrie- bzw. Rasterisierungsstufe ausführen kann.

Vertex- und Fragmentprogramme werden zusammen auch als *programmierbare Shader* bezeichnet. Praktisch werden sie entweder durch assemblerartige Programmiersprachen implementiert, oder neuerdings durch an C angelehnte Hochsprachen wie die *OpenGL Shading Language* (kurz *glsl*, [72]).

Durch programmierbare Shader können grundsätzlich beliebige lokale Beleuchtungsmodelle implementiert werden. Dazu werden frei wählbare Informationen pro Eckpunkt übermittelt und im Vertex- bzw. Fragmentprogramm wie gewünscht verknüpft. Darüber hinaus kann in einem Vertexprogramm die Transformation der Eckpunkte, das heißt die Vorschrift zur Positionierung der Geometrie, frei programmiert werden, wodurch zum Beispiel hierarchisch zusammengesetzte 3D-Geometrie allein durch Verwendung der Graphikhardware animiert werden kann.

Zukünftige Entwicklungen

Die Entwicklung von und mit programmierbarer Graphikhardware ist bis heute bei weitem nicht abgeschlossen. Beispielsweise ist der Zugriff auf Texturen auch in Vertexprogrammen erst seit kurzem mit Graphikhardware der GeForce6000-Serie von NVidia möglich.

Grundlegende Fragestellungen ergeben sich vor allem aus der mangelhaften Modularität programmierbarer Shader. Zwar können mit glsl verschiedene Shader, die unterschiedliche Funktionen definieren, auf der Graphikhardware gelinkt und ausgeführt werden. Praktisch wird diese Möglichkeit aber nur wenig genutzt. Dies ist durch eine hohe Komplexität der Implementierung begründet, weil die Anwendung dafür verantwortlich ist, passende Shader als Strings zu erzeugen und zu verwenden. Zur gemeinsamen Verwendung mehrerer atomarer Renderingfunktio-

nen, wie zum Beispiel zur automatischen Generierung einer Texturkoordinate und zur Beleuchtung des Eckpunkts mit einer Lichtquelle, muss dabei zumindest der String der zentralen `main`-Einsprungsfunktion dynamisch zusammengesetzt werden. Die Anzahl an benötigten Kombinationen kann dabei je nach Anwendung sehr groß sein, da prinzipiell alle atomaren Renderingfunktionen in einem Programm genutzt und miteinander verknüpft werden können [58]. Demgegenüber hat die festverdrahtete Renderingpipeline für den Anwendungsentwickler den großen Vorteil, dass alle Renderingoptionen orthogonal zueinander sind und die Komplexität, einen gültigen Shader zu erzeugen, im Graphiktreiber gekapselt ist.

Mittelfristig zeichnet sich daher die Entwicklung einer Abstraktionsschicht zwischen Renderingsystem und Anwendung ab, mit der atomare Renderingfunktionen kombiniert, komplexe Shader in Einzelteile getrennt, und der Zugang zu verschiedenen Shadersprachen vereinheitlicht wird. Das *Sh* GPU Metaprogramming Toolkit [51] ist ein interessanter Ansatz zur dynamischen Erzeugung von Shadern durch Kapselung atomarer Renderingfunktionen in sogenannten Kernels, und zu deren Aggregation und Konkatenation [52].

Durch die zunehmend freie Programmierbarkeit der Graphikhardware gibt es außerdem viele Ansätze, die Hardware für allgemeine, parallelisierbare, sogenannte *Stream-Computing*-Probleme zu nutzen. Dieses neue Forschungsgebiet wird bezeichnet mit dem Akronym *GPGPU* (general-purpose computation on GPUs, [30]). Beispiele sind die Implementierung computergraphischer Algorithmen wie Ray-Tracing [66], Photon-Mapping [67] oder Radiosity [13] unter Nutzung programmierbarer Shader, aber auch Algorithmen abseits der Computergraphik wie Strömungssimulationen oder numerische Algorithmen. Bibliotheken wie *Brook* [11], die die Leistungsfähigkeit der Graphikhardware für allgemeine Stream-Computing-Probleme über eine allgemeine Programmierschnittstelle zur Verfügung stellen, können für die praktische Implementierung solcher Algorithmen unter Verwendung der Graphikhardware nutzbringend eingesetzt werden.

2.4 Dynamische Erzeugung von Texturen

Die Funktionalität und mögliche Komplexität programmierbarer Shader ist mit fortschreitender Entwicklung der Graphikhardware mittlerweile beträchtlich. Dennoch beruhen sie auf einem eingeschränkten Programmiermodell und können daher nicht zur Implementierung aller graphischen Phänomene eingesetzt werden. Eine grundlegende Einschränkung ist, dass zur Laufzeit eines Vertexprogramms nur Eckpunkt-Informationen für einen Eckpunkt und konstante Informationen des Renderingkontextes zugreifbar sind, nicht aber Informationen über andere Geometrie in der Szene. Solche Informationen können nur durch Hilfskonstrukte zur Verfügung gestellt werden, nämlich durch Texturen, die Informationen über die Szenengeometrie enthalten. In einigen Fällen ist es auch möglich, die benötigten Informationen über die Szene direkt im Framebuffer, zum Beispiel im Stencilpuffer, abzulegen.

Szenengeometrie ist meist animiert, daher müssen Texturen, die Informationen über Szenengeometrie enthalten, in der Regel dynamisch, das heißt für jedes Bild neu erzeugt werden. Die Entwicklung von APIs zur Erzeugung *dynamischer Texturen* ist in OpenGL noch nicht abgeschlossen. Der heutige Weg geht über das Erzeugen eines nicht-sichtbaren Framebuffers, eines sogenannten *P-Buffers*. Der Inhalt des P-Buffers wird dabei mit einer Textur assoziiert, die zum Rendern in andere Framebuffer verwendet werden kann. Nachteile dieses Verfahrens sind, dass 1) das API zum Erzeugen eines P-Buffers betriebssystemabhängig ist und 2) der Wechsel vom Framebuffer in den P-Buffer zum Rendern ein teures und technisch nicht unbedingt notwendiges Umschalten des Renderingkontextes erfordert. Ein aktueller Vorschlag für ein diesbezüglich verbessertes API zur dynamischen Erzeugung von Texturen ist die *Framebuffer-Object*-Erweiterung [59].

Anwendungen für dynamische Texturen

Dynamische Texturen werden zur Implementierung einer Vielzahl von Renderingverfahren eingesetzt, die folgendermaßen eingeteilt werden können:

- **Bildbasierte Renderingverfahren.** Dies sind Renderingverfahren, die im Bildraum auf ein bereits gerendertes Bild angewendet werden. Dazu wird das gerenderte Bild der Szene in eine Textur kopiert. Dann wird ein Rechteck mit dieser Textur über den gesamten Framebuffer gerendert, so dass jedes Pixel im Framebuffer genau dem Texel in der Textur an gleicher Position entspricht. Dabei wird, meist mit einem Fragmentprogramm, die Farbe jedes Fragments unter Verwendung der Texel in seiner Umgebung geeignet berechnet.
- **Verfahren für Beleuchtung erster Ordnung.** Beleuchtungseffekte wie Schatten, Reflexion oder Refraktion werden in der Regel gerendert, indem zur Schattierung von Geometrie Texturen verwendet werden, die Informationen über die sonstige Geometrie in der Szene enthalten. Beispiele für solche Texturen sind Shadow-Maps zur Speicherung von Tiefeninformationen oder dynamische Cubemaps zur Speicherung eines gespiegelten Bilds für Reflexionen. Die Texturen werden dabei im Allgemeinen von einer anderen Kameraposition als von der Position der Szenenkamera aus erstellt.
- **Virtualisierung von programmierbaren Shadern.** Zwischenergebnisse eines Shaders können temporär in Texturen abgespeichert werden. Dies wird zum Beispiel für Shader benötigt, die von der Hardware aufgrund ihrer Komplexität oder ihres Umfangs nicht nativ ausgeführt werden können. So ein Shader wird in mehrere Teile aufgeteilt, und die Zwischenergebnisse werden als Texturen gespeichert und in den nachfolgend ausgeführten Shaderteilen verwendet. Der *F-Buffer* ist ein Ansatz, die Aufteilung eines Shaders und das Anlegen der temporären Texturen durch einige Hardware-Erweiterungen in die Graphiktreiberschicht zu integrieren und damit Anwendungsentwickler nicht mit der Virtualisierung von Shadern per Hand zu belasten [49]. Auf praktischer Hardware hat sich aber bisher der F-Buffer nicht durchgesetzt.
- **Analytische Berechnungen.** Texturen enthalten berechnete Informationen, die keine Eigenschaften rasterisierter Geometrie repräsentieren. Solche dynamischen Texturen werden für GPGPU-Algorithmen benötigt, bei denen in der Regel Texturen die Eingaben und Zwischenergebnisse für die durchgeführten Berechnungen enthalten.

Der Akkumulationspuffer

Bildbasierte Renderingverfahren können auch mit mehreren Ursprungsbildern durchgeführt werden, beispielweise um zwei Bilder ineinander überzublenden. Dazu werden beide gerenderte Bilder in jeweils eine Textur kopiert, und mit einem Fragmentprogramm werden die Texturen gewichtet addiert. Für diese Art der Anwendung gibt es einen weiteren Mechanismus, der dasselbe erlaubt: den *Akkumulationspuffer* [34]. Dessen Funktionsweise ist wie folgt: Der Inhalt des Farbpuffers kann zu beliebigen Zeitpunkten in den Akkumulationspuffer übernommen werden. Dies geschieht im Allgemeinen mehrfach, wobei die Farbwerte mit einstellbaren Gewichten akkumuliert werden. Schließlich wird zur Anzeige der Inhalt des Akkumulationspuffers wieder in den Farbpuffer übernommen.

Der Akkumulationspuffer lässt sich vollständig durch Verwendung dynamisch erzeugter Float-Texturen nachbilden, die zur Kombination additiv verknüpft werden. Float-Texturen sind erforderlich, weil die OpenGL-Spezifikation von einem Akkumulationspuffer eine höhere Genauigkeit zur Speicherung der Farbwerte als für den normalen Farbpuffer erfordert. Daher ist auf Standard-Graphikhardware der Akkumulationspuffer erst seit der Verfügbarkeit von Float-Texturen hardwarebeschleunigt, durch die er intern emuliert wird.

2.5 Multipass-Rendering

Algorithmen, die Multipass-Rendering verwenden, sind einer Obermenge der Algorithmen, die dynamisch erzeugte Texturen verwenden. Im Kontext dieser Arbeit heißt *Multipass-Rendering*, dieselbe Geometrie mehrfach zu rendern, um ein einziges Bild zu erzeugen. Bei jedem Rendern werden im Allgemeinen unterschiedliche Einstellungen des Immediate-Mode Renderingsystems vorgenommen. Außerdem können statt der originalen Geometrie davon abgeleitete Varianten zur Hardware geschickt werden. Verwaltet die Applikationsstufe die Szenenbeschreibung in einer separaten Datenstruktur, werden die darin enthaltenen Geometrie-Objekte für Multipass-Rendering mehrfach traversiert und damit gerendert. Jeder dieser Traversierungen bezeichnet man als einen *Renderingdurchlauf* (rendering pass), kurz Durchlauf.

Anwendungen für Multipass-Verfahren

Multipass-Rendering ist eine häufige Arbeitsweise vieler Renderingverfahren, daher werden diese Verfahren mit dem Begriff *Multipass-Verfahren* zusammengefasst. Sie sind vor allem für Renderingeffekte erforderlich, die zur Schattierung von Geometrie die globale Szenengeometrie berücksichtigen. In den einzelnen Durchläufen können dabei die folgenden temporären Daten erzeugt werden:

- **Temporäre Informationen im Framebuffer.** Es können beispielsweise im Stencilpuffer bestimmte Gebiete maskiert werden, um später eine Aktualisierung nur in diesen Gebieten zu bewirken. Im Tiefenpuffer kann ein Tiefenbild der Szene vorweg abgelegt werden für Renderingverfahren, die Berechnungen aufgrund der Tiefenwerte der Szene durchführen. Im Farbpuffer können Ergebnisse einzelner Renderingdurchläufe durch Blending akkumuliert werden.
- **Dynamische Texturen.** Diese werden in späteren Durchläufen wieder genutzt. Eine Variante davon ist die Verwendung des Akkumulationspuffers.
- **Analytische Daten.** Die Szenenbeschreibung wird analysiert, ohne dass Renderingbefehle an die Graphikhardware gesendet werden. Dies kann zum Beispiel erforderlich sein, um Bounding-Boxen von Geometrie-Objekten in der Szene zu berechnen. Es sich also um eine Traversierung der Szenenbeschreibung zu Analysezwecken.

Globale Multipass-Verfahren

Multipass-Verfahren können anhand des Bereiches der Szenenbeschreibung eingeteilt werden, den sie mehrmals traversieren. Eine solche Einteilung korrespondiert zum Teil mit dem Konzept der *Berechnungsfrequenzen* (computation frequencies), das Grundlage der Stanford Real-Time Shading Language ist [65]. Neben Berechnungen pro Fragment und pro Vertex, die von der Graphikhardware in der Rasterisierungs- und Geometriestufe implementiert werden, definiert diese Sprache außerdem Berechnungen pro geometrisches Objekt und konstante Einstellungen für die gesamte Szene.

Globale Multipass-Verfahren traversieren die gesamte Szenenbeschreibung mehrfach. Zum einen implementieren sie Effekte, die für die gesamte Szene gleichermaßen gelten, im Sinne der Berechnungsfrequenzen also konstante Einstellungen für die Szene. Solche Renderingverfahren nehmen lediglich vor und nach jeder Traversierung der Szenenbeschreibung Renderingeinstellungen vor und wieder zurück, und sie aktualisieren gegebenenfalls dynamische Texturen. Ein Beispiel für so ein Verfahren ist Tiefenunschärfe mit dem Akkumulationspuffer (Abschnitt 3.5).

Globale Multipass-Verfahren mit Bezug auf einzelne Geometrie-Objekte

Andere globale Multipass-Verfahren reagieren auf einzelne Geometrie-Objekte in der Szenenbeschreibung separat: Je nach Renderingdurchlauf werden zum Rendern unterschiedlicher Geometrie-Objekte verschiedene Renderingeinstellungen verwendet. Geometrie kann in unterschiedlichen Durchläufen auch gefiltert, das heißt, nicht gerendert werden. Vor und nach einer Traversierung der Szene können ebenfalls Einstellungen des Renderingkontextes vorgenommen werden.

Globale Multipass-Verfahren mit Bezug auf einzelne Geometrie-Objekte lassen sich nicht zufriedenstellend auf das Konzept der Berechnungsfrequenzen abbilden. Da es eine Reihe derartiger Verfahren gibt, ist dieses Modell an dieser Stelle offenbar nicht ausreichend. Ein typisches Beispiel für ein solches Verfahren ist die Darstellung transparenter Geometrie in einer Szene durch zwei Renderingdurchläufe für opake und transparente Geometrie-Objekte (Abschnitt 3.3.1).

Lokale Multipass-Verfahren

Lokale Multipass-Verfahren traversieren nur einen Teil einer Szenenbeschreibung oder nur ein einzelnes geometrisches Objekt mehrfach, um einen Renderingeffekt zu erzeugen. Dies sind im Sinne der Berechnungsfrequenzen also Berechnungen pro Objekt. In der Regel werden lokale Multipass-Verfahren verwendet, um mangelhafte Shadingfähigkeiten der Renderinghardware nachzubilden, das heißt zur Virtualisierung von programmierbaren Shadern. Jedoch lassen sich bestimmte Verfahren, zum Beispiel zur bildbasierten Kantenhervorhebung [56], auch mit beliebig mächtigen Shadingfähigkeiten ohne lokale Multipass-Verfahren nicht implementieren.

Ein Beispiel für ein lokales Multipass-Verfahren ist der Bumpmapping-Algorithmus von Kilgard zur Beleuchtung von Geometrie mit einer Bumpmap und einer Lichtquelle in drei Renderingdurchläufen [42].

Kapitel 3

KATEGORIEN ECHTZEITFÄHIGER RENDERINGEFFEKTE

Bei der Kombination von Renderingeffekten treten zwei unterschiedliche Fragen auf: 1) Sind verschiedene Renderingeffekte semantisch kombinierbar? 2) Sind verschiedene Renderingverfahren, die die jeweiligen Effekte implementieren, technisch kombinierbar? Zur Beantwortung dieser Fragen werden in diesem Abschnitt wichtige allgemeine Renderingeffekte kategorisiert und, wo erforderlich, Verfahren zu ihrer Darstellung skizzert. Akenine-Möller und Haines geben eine gute Zusammenstellung und Vertiefung der meisten hier beschriebenen und weiterer Renderingverfahren [3].

3.1 Oberflächenschattierungen

Eine *Oberflächenschattierung* befasst sich mit der lokalen Beleuchtung und Schattierung von Oberflächen. Dazu werden virtuelle Lichtquellen und Materialeigenschaften der Oberfläche eingesetzt, es wird aber keine weitere Geometrie in der Szene bei der Beleuchtung oder Schattierung berücksichtigt. Technisch ist das Standardrendering mit der festverdrahteten Renderingpipeline zur normalen Beleuchtung und Schattierung von Geometrie eine einfache, von OpenGL direkt unterstützte Form der Oberflächenschattierung. Durch Verwendung einer anderen Oberflächenschattierung ergeben sich Änderungen im Farbpuffer, nicht aber im Tiefenpuffer, das heißt, die Form und Position von Oberflächen bleibt unverändert. Wir gehen grundsätzlich davon aus, dass nur eine Oberflächenschattierung für jede Geometrie verwendet wird, diese Effekte also nicht automatisch kombiniert werden.

Realistische Oberflächenschattierungen

Realistische Oberflächenschattierungen haben eine möglichst photorealistische Simulation und Darstellung von Oberflächeneigenschaften zum Ziel. Eine mittlerweile veraltete Form eines solchen Effekts ist Beleuchtung durch *Lightmaps*, die die Beleuchtung eine Lichtquelle pixelgenau

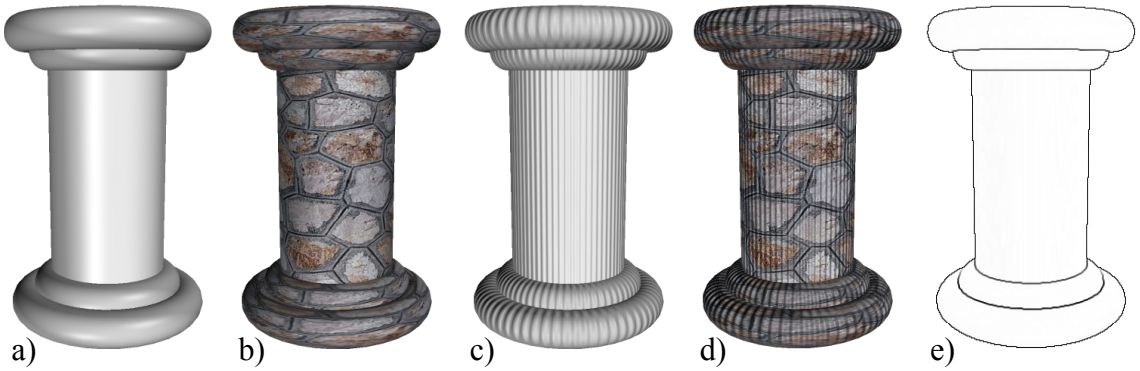

Abbildung 5: Verschiedene Oberflächenschattierungen auf das Modell einer Säule angewendet. Phong-Schattierung (a), zusätzlich mit Textur (b), Bumpmapping (c), zusätzlich mit Textur (d), Kantenhervorhebung (e).

und nicht wie sonst üblich pro Eckpunkt approximieren [2]. Eine Beleuchtungsberechnung pro Fragment kann durch *Phong-Schattierung* realisiert werden [62], bei der das Beleuchtungsmodell für jedes Fragment auf die Interpolation der Normalen der Eckpunkte angewendet wird (Abbildung 5a und, mit Textur und anderen Materialeigenschaften, 5b). Neben dem Blinn- oder Phong-Beleuchtungsmodell werden dabei auch komplexere lokale Beleuchtungsmodelle verwendet, wie zum Beispiel *Bidirektionale Reflexionsverteilungsfunktionen* (bidirectional reflection distribution functions, BRDFs [55]), bei denen die resultierende Helligkeit eine Funktion über alle auftretende Ein- und Ausfallswinkel ist. Dies kann zur verbesserten Simulation unterschiedlicher Materialeigenschaften genutzt werden. Jedes Beleuchtungsmodell lässt sich mit *Bumpmapping* kombinieren [6], bei dem die Normale zusätzlich durch eine texturbasierte feine Störung, die *Bumpmap*, modifiziert wird. Dadurch ergibt sich ein Eindruck von Mikrostrukturen auf der Oberfläche (Abbildung 5c und, mit Textur, 5d).

Nicht-photorealistische Oberflächenschattierungen

Eine andere Kategorie von Oberflächenschattierungen hat das *nicht-photorealistische Rendering* (NPR, [82]) zum Gegenstand. Hier erfolgt die Darstellung nach dem Vorbild von Zeichnungen oder technischen Illustrationen. Beispiele sind Effekte zur *Kantenhervorhebung* [56] (Abbildung 5e), *Toon-Shading* [47], bei dem die Darstellung wie bei Comiczeichnungen auf wenige Farbtöne reduziert wird, *Gooch-Shading* [29], bei dem anstatt eines hell-dunkel Farbverlaufs ein warm-kalt Farbverlauf zur Schattierung verwendet wird, oder *Hatching* [63], bei dem Form und Materialeigenschaften mit Schraffuren dargestellt werden.

Implementierung von Oberflächenschattierungen

Durch die Verfügbarkeit programmierbarer Shader (Abschnitt 2.3) erfolgt die Entwicklung von *Oberflächenshadern*, die neuartige Oberflächenschattierungen implementieren, zur Zeit sehr schnell. Zur Implementierung einiger Oberflächenschattierungen werden darüber hinaus dynamisch erzeugte Texturen benötigt, das heißt lokale Multipass-Verfahren. Beispielsweise können zur Kantenhervorhebung Unstetigkeiten im Tiefenpuffer ausgewertet werden, wozu zunächst eine Tiefentextur der Geometrie erzeugt werden muss [56].

3.2 Markierungseffekte

Ein *Markierungseffekt* ist ein Renderingeffekt zur Hervorhebung von Szenenobjekten. Markierungseffekte werden benötigt, um interaktiv ausgewählte oder andere fokussierte Geometrie-Objekte erkennbar als solche darzustellen. Einige Markierungseffekte sind spezielle Oberflä-

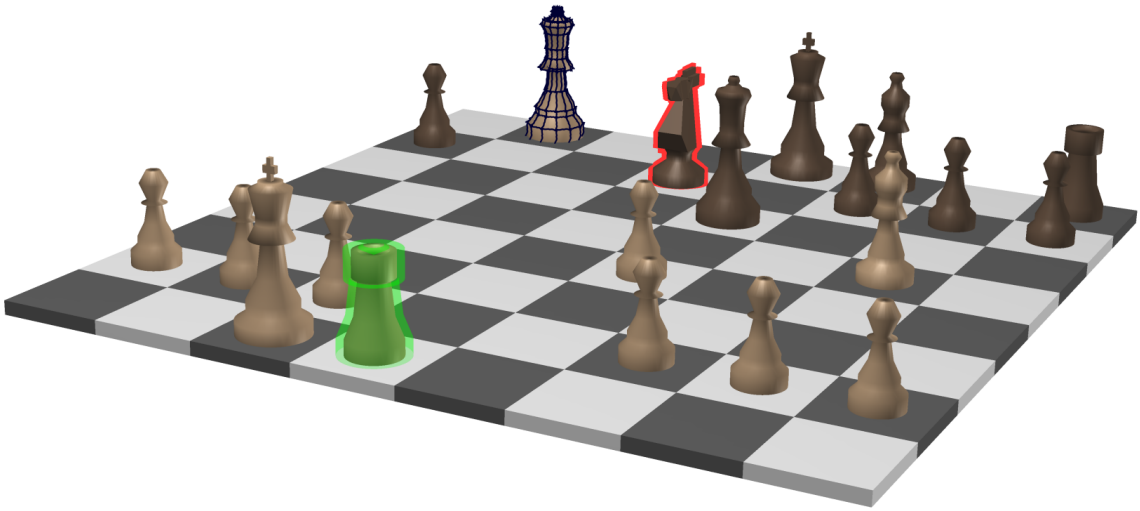

Abbildung 6: Illustrierte Schachpartie, in der zur Hervorhebung von Figuren verschiedene Markierungseffekte verwendet werden: Die schachbietende weiße Dame, hervorgehoben durch texturverstärkte Kanten, der mit Signalfarbe umrandete schwarze Springer, der die Dame schlagen muss, und der weiße, mit Halo dargestellte Turm, der anschließend matt setzen kann.

chenschattierungen. Markierungseffekte bilden jedoch eine eigene Kategorie von Renderingeffekten, weil ihre Wirkung orthogonal zu einer Oberflächenschattierung ist, das heißt, die Materialeigenschaften von Geometrie aufgrund einer Oberflächenschattierung sind mit der hervorgehobenen Darstellung durch einen Markierungseffekt kombinierbar.

Beispiele für Markierungseffekte

Markierungseffekte sind in Hinblick auf ihre visuellen Grundelemente meist einfach (Abbildung 6). Ein Beispiel ist die *Umrandung* (outlining) hervorgehobener Geometrie mit einer konstanten Signalfarbe. Eine ähnlicher Effekt ist das *Halo*, das heißt, das Übermalen von Geometrie mit einer transparenten, konstanten Farbe, meist in einem Bereich, der das eigentliche Szenenobjekt etwas überragt, oder alternativ nur in einem Bereich knapp außerhalb der sichtbaren Geometrie [41].

Kanten von Szenenobjekten können durch zusätzliche Darstellung von texturierten Rechtecken entlang der Kanten hervorgehoben werden. Diese *texturverstärkten Kanten* (stroke textures, [19]) erlauben durch Auswahl verschiedener Alphatexturen sowie durch Störung der Ausrichtung und Länge der Rechtecke viele Varianten in der Darstellung.

Implementierung von Markierungseffekten

Markierungseffekte können durch lokale Multipass-Verfahren implementiert werden. Eine typische Vorgehensweise ist, die betroffene Geometrie mit ihrer eigentlichen Oberflächenschattierung zu rendern und anschließend in einem weiteren Durchlauf den Markierungseffekt darzustellen. Je nach Art des Effekts können aber auch andere Vorgehensweisen erforderlich sein.

3.3 Transparenz

Transparenz bezeichnet die Eigenschaft von 3D-Objekten, ganz oder teilweise lichtdurchlässig zu sein und damit den Blick auf weiter hinten liegende Geometrie zu ermöglichen. In unserer wahrnehmbaren Umwelt ist Transparenz stets mit Refraktion verbunden, das heißt, der Brechung von Lichtstrahlen am Übergang optisch verschieden dichter Medien. Im Echtzeitrendering ist es allerdings üblich, Refraktion als separaten Renderingeffekt aufzufassen, der thema-

tisch dem Bereich Beleuchtung erster Ordnung angehört. Transparenzverfahren ohne Berücksichtigung von Refraktion sind somit eine eigenständige Klasse von Renderingverfahren; sie werden als globale Multipass-Verfahren mit Bezug auf einzelne Geometrie-Objekte, genauer auf deren Transparenzeigenschaft, realisiert.

3.3.1 Grundlegendes Verfahren zur Transparenzdarstellung

Die Darstellung von transparenter Geometrie ohne Refraktion bereitet im Echtzeitrendering, gemessen an der einfachen Problemstellung, überraschende Schwierigkeiten. Grundsätzlich rendert man transparente Geometrie durch Alpha-Blending: Der Alphakanal eines Geometrie-Objekts kodiert den Sichtbarkeitsanteil, das heißt ein Alphawert von 1 bedeutet opak, und ein Alphawert von 0 bedeutet voll transparent, also nicht sichtbar. Bei der Rasterisierung von Geometrie wird mit der Blendingfunktion `glBlendFunc(GL_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` das Fragment entsprechend seinem Alphawert dem Framebuffer hinzugemischt. Dadurch ergibt sich der Transparenzeindruck: Sowohl das alte Bild im Framebuffer als auch neue Fragmente sind sichtbar.

Die grundlegende Schwierigkeit besteht darin, dass durch Transparenz verschiedene Geometrie an einer Position gleichzeitig sichtbar ist, der Tiefenpuffer aber nur einen Tiefenwert für jede Position enthalten kann. Somit reicht der Z-Buffer-Algorithmus für korrekte Verdeckungsermittlung nicht aus, und es kommt zu Renderingartefakten. Transparenzverfahren unterscheiden sich vorwiegend darin, auf welche Weise sie diese Artefakte verhindern oder minimieren.

Der Standardalgorithmus für Transparenz unterteilt die Geometrie-Objekte in opake und transparente Objekte. Zunächst werden mit normalen Einstellungen des Tiefenpuffers alle opaken Objekte gerendert, für die der Z-Buffer-Algorithmus korrekt funktioniert. Danach erfolgt mit Alpha-Blending das Rendern aller transparenten Geometrie-Objekte. Dabei wird der Tiefenpuffer nicht mehr aktualisiert, damit der Tiefentest immer relativ zur opaken Geometrie durchgeführt wird. Opake Geometrie erscheint mit diesem Verfahren also korrekt, aber bei transparenter Geometrie kommt es zu Artefakten (Abbildung 7c und d). Diese können zuverlässig nur verhindert werden, indem alle Dreiecke der transparenten Geometrie-Objekte von hinten nach vorne sortiert gerendert werden. Eine Sortierung auf Dreiecksebene ist in der Praxis aber mit sehr großem Aufwand verbunden, und man beschränkt sich daher meist auf eine Sortierung der Geometrie-Objekte und toleriert die entstehenden Artefakte.

3.3.2 Ordnungsunabhängige Transparenz

Ordnungsunabhängige Transparenz (order-independent transparency, [22]) basiert auf einer *Tiefenschichtenermittlung* (depth peeling), das heißt, einer pixelgenauen Sortierung der *Tiefenschichten* einer Szene von vorne nach hinten. Die einzelnen Tiefenschichten werden dabei jeweils in Tiefentexturen abgelegt, und die jeweils n-te Tiefenschicht wird – mit einer Zweckentfremdung des texturbasierten Schattentests, der als Art zweiter Tiefentest verwendet wird – zur Generierung der n+1-ten Tiefenschicht verwendet: Es werden nur Fragmente in den Framebuffer geschrieben, die sich hinter der n-ten Tiefenschicht befinden, und für die gleichzeitig der gewöhnliche Tiefentest gelingt. Zu jeder Tiefenschicht wird außerdem eine entsprechende *Farbschicht* erzeugt, welche die Schattierung und Transparenz der Fragmente in der Tiefenschicht enthält. Jede Farbschicht wird in einer RGBA-Textur abgelegt.

Wenn n Tiefenschichten ermittelt sind, können alle Fragmente in oder vor der n-ten Schicht artefaktfrei mit Transparenz dargestellt werden: Nacheinander werden schichtweise alle Farbschichten von der n-ten bis zur ersten Schicht mit Alpha-Blending bildbasiert in den Farbbuffer gerendert (Abschnitt 2.4). Durch die Sortierung der Tiefenschichten ist insgesamt die Darstellung der Fragmente von hinten nach vorne sichergestellt.

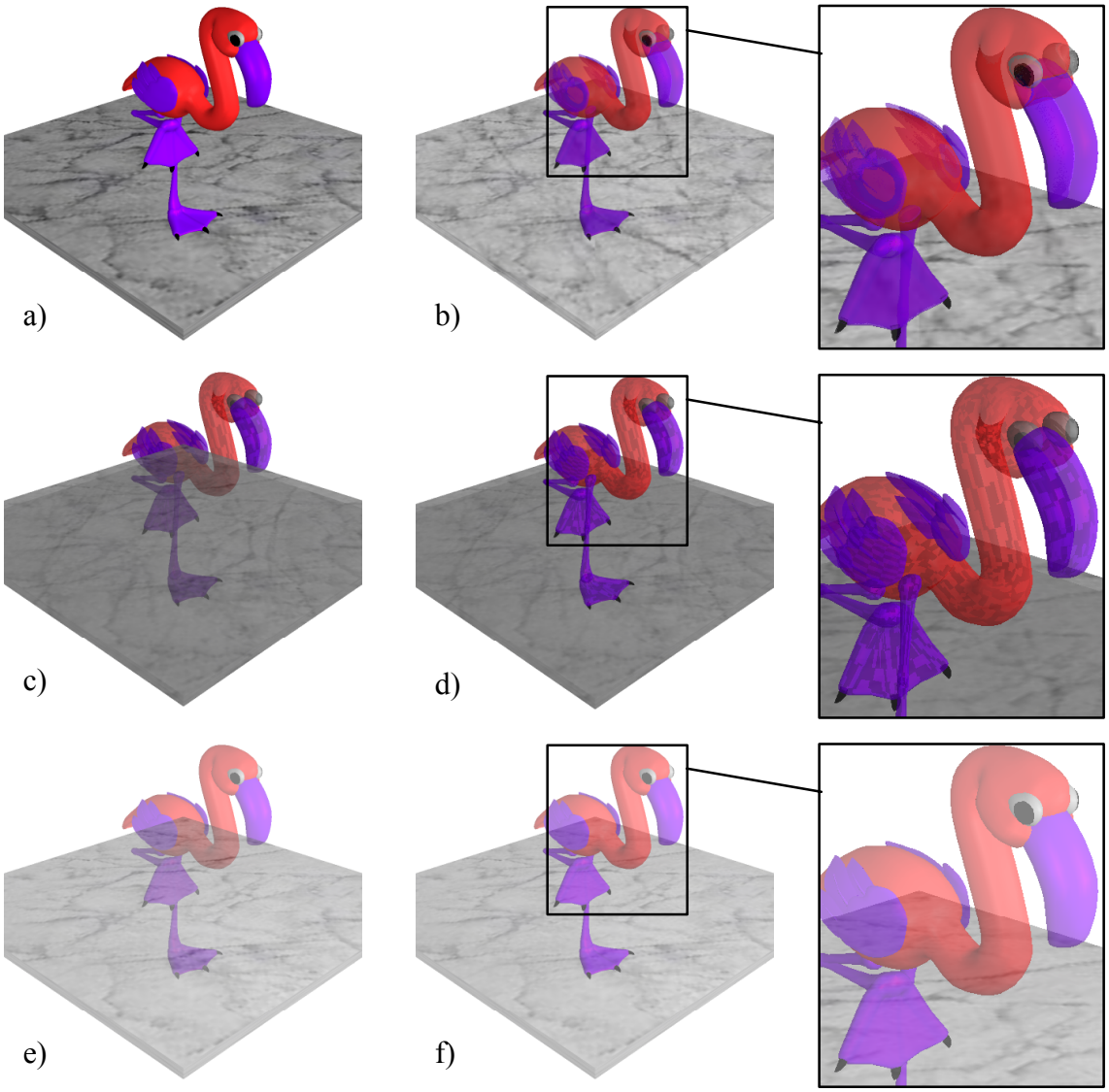

Abbildung 7: Vergleich verschiedener Transparenzverfahren. Sowohl Flamingo als auch Bodengeometrie sind dabei transparent. Vergleichsbild ohne Transparenz (a). Ordnungsunabhängige Transparenz (b). Standardtransparenzverfahren; Renderingreihenfolge erst Flamingo, dann Boden (c) bzw. umgekehrt (d). Transparenzdarstellung von Alternativen; Renderingreihenfolge erst Flamingo, dann Boden (e) bzw. umgekehrt (f).

Ordnungsunabhängige Transparenz ermöglicht ohne objektbasierte Sortierung der Geometrie-Objekte artefaktfreie Bilder (Abbildung 7b). Damit ist das Verfahren aus Sicht der Bildqualität optimal. Der Nachteil des Verfahrens ist die hohe Laufzeit: Zur Ermittlung jeder Tiefenschicht müssen alle Geometrie-Objekte gerendert werden, zudem sind die Speicher- und Laufzeitkosten für das Erzeugen der RGBA- und Tiefentexturen für jede Tiefenschicht erheblich. Ein weiterer Nachteil ist, dass die Graphikhardware für das Verfahren den texturbasierten Schattentest unterstützen, also mindestens der GeForce3-Generation angehören muss.

3.3.3 Transparenzdarstellung von Alternativen

In vielen Anwendungen von Transparenz ist keine Darstellung mehrerer Tiefenschichten eines Geometrie-Objekts erforderlich; stattdessen dient die Transparenz zum Vergleich verschiedener

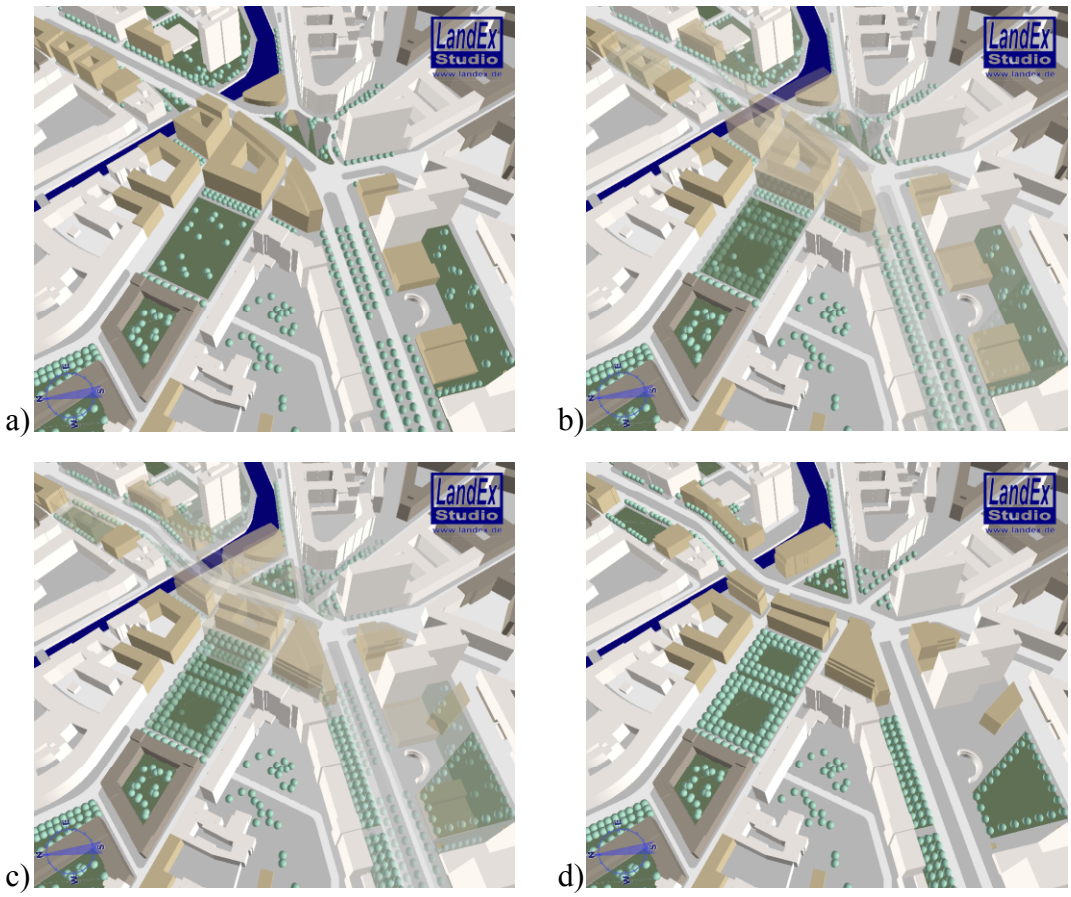

Abbildung 8: Darstellung eines Stadtmodells vom Spittelmarkt in Berlin. Zustand im Jahre 2001 (a). Planung im Jahre 2010 (d). Übergang zwischen den beiden Zuständen (b, c). Alle Bilder wurden mit Hilfe der Darstellung von Alternativen erzeugt.

Objekte in einer Szene, von denen, um eine leichte Wiedererkennung zu ermöglichen, jeweils nur die vorderste Schicht sichtbar sein muss. Ein Beispiel ist die Visualisierung verschiedener Planungsalternativen in interaktiven Stadtmodellen (Abbildung 8): Es werden zwei alternative Bauvorhaben miteinander verglichen, dazu erlaubt eine interaktive Anwendung interaktives, stufenloses Hin- und Herschalten zwischen den beiden verschiedenen transparent dargestellten Alternativen.

Die *Transparenzdarstellung von Alternativen* hat das Ziel, nur die vordersten Fragmente einer transparenten Alternative in den Farbpuffer zu rendern. Dies wird durch das folgende Multipass-Verfahren erreicht: Im ersten Durchlauf wird die opake Szenengeometrie gerendert. Im nächsten Durchlauf wird für jede Alternative berechnet, an welchen Stellen sie sich vor der opaken Szenengeometrie befindet und damit sichtbar sein muss: Dazu wird sowohl in Farb- als auch Tiefenpuffer nicht geschrieben, und der Stenciltest wird so konfiguriert, dass für jedes Fragment einer Alternative vor dem Tiefenpuffer ein Bit im Stencilpuffer gesetzt wird; jede Alternative hat dabei ihre eigene Bit-Ebene im Stencilpuffer. Anschließend werden für jede Alternative zwei weitere Durchläufe durchgeführt: Im ersten wird der Tiefenpuffer gelöscht und an den im Stencilpuffer markierten Stellen die vordere Tiefenschicht der Alternative im Tiefenpuffer bestimmt; Im zweiten werden die Farbwerte der Fragmente einer Alternative, die in der vorderen Tiefenschicht liegen, durch Alpha-Blending in den Farbpuffer hineingemischt.

Transparenzdarstellung von Alternativen vermeidet die hohen Hardwareanforderungen von ordnungsunabhängiger Transparenz, ohne dabei anfällig für Artefakte zu sein: Dadurch, dass von jeder Alternative nur die vorderste Tiefenschicht, das heißt ein einziges Fragment an jeder Position sichtbar ist, erfolgt die Darstellung jeder einzelne Alternative für sich ohne Artefakte. Lediglich die Reihenfolge der Darstellung der verschiedenen Alternativen kann dazu führen, dass zunächst eine vordere und dann eine dahinter liegende Alternative gerendert wird und damit das Blending mathematisch in der falschen Reihenfolge erfolgt (Abbildung 7e). Dies fällt aber in der Praxis im Vergleich zur korrekten Reihenfolge (Abbildung 7f) nur geringfügig auf.

3.4 Beleuchtung erster Ordnung

Beleuchtung erster Ordnung ist die einfachste Form globaler Beleuchtung: Zur Schattierung der Geometrie wird, zusätzlich zur Verwendung eines lokalen Beleuchtungsmodells, die erste Interaktion der Geometrie mit anderer Geometrie in der Szene ausgewertet. Das heißt, Oberflächen können andere Geometrie spekular reflektieren und mit Schatten aufgrund anderer Geometrie in der Szene gerendert werden. Zur Beleuchtung erster Ordnung gehört außerdem die Darstellung von Refraktion im Zusammenhang mit Transparenz. Refraktion wird in dieser Arbeit nicht weiter betrachtet.

Schatten und Reflexionen sind für viele Anwendungsbereiche von großer Wichtigkeit: Zum einen bewirken sie direkt eine Bilddarstellung mit höherer Qualität. Zum anderen setzen sie unterschiedliche Geometrie in einer Szene zueinander in Beziehung und erlauben es dem Betrachter damit, räumliche Relationen der Geometrie wahrzunehmen. Dies wird zum Beispiel in der Abbildung 9a deutlich, in der wegen des Fehlens von Schatten und Bodenreflexion das Auto nicht mit der Bodenplatte verbunden wirkt. Insgesamt ermöglichen Schatten und Reflexionen damit eine effektivere Navigation in und Interaktion mit der Szene [20].

Überblick über Verfahren zur Beleuchtung erster Ordnung

Algorithmen zur Darstellung globaler Beleuchtung beliebiger Ordnung sind Ray-Tracing, Radiosity und Photon-Mapping. Sie werden derzeit aufgrund ihrer hohen Laufzeit nicht für Echtzeitrendering verwendet. Daher hat sich im Bereich des Echtzeitrenderings eine Klasse globaler Multipass-Verfahren gebildet, mit denen Beleuchtung erster Ordnung dargestellt werden kann. Die wichtigsten Beispiele für diese Verfahren sind stencilbasierte [35] und texturbasierte Reflexionen [32] sowie Schattenvolumen [14] und Shadow-Mapping [90]. Schattenverfahren variieren außerdem darin, ob sie harten Schatten erzeugen (hard shadows, Abbildung 9d) oder ob sie weiche Schatten (soft shadows) mit Kernschatten (umbra) sowie Halbschatten (penumbra) berechnen bzw. approximieren.

3.4.1 Reflexionen

Grundsätzlich kann zwischen zwei Arten von Reflexionen unterschieden werden: Eine *planare Reflexion* wird für eine ebene Fläche berechnet, in der sich die übrige Geometrie in der Szene spiegelt (Abbildung 9b). *Environment-Mapping* ermöglicht eine Approximation der Reflexion eines gekrümmten Geometrie-Objekts, das seine gesamte Umgebung reflektiert (Abbildung 9c).

Verfahren zur Darstellung planarer Reflexionen

Zum stencilbasierten Rendering planarer Reflexionen wird zunächst die Szene wie gewöhnlich gerendert, dabei wird die sichtbare Fläche des planaren Reflektors im Stencilpuffer markiert. In diesem Bereich wird anschließend der Farb- und Tiefenpuffer neu initialisiert und dann die Szene von der gespiegelten Kameraposition aus nochmals gerendert. Schließlich wird der Reflektor

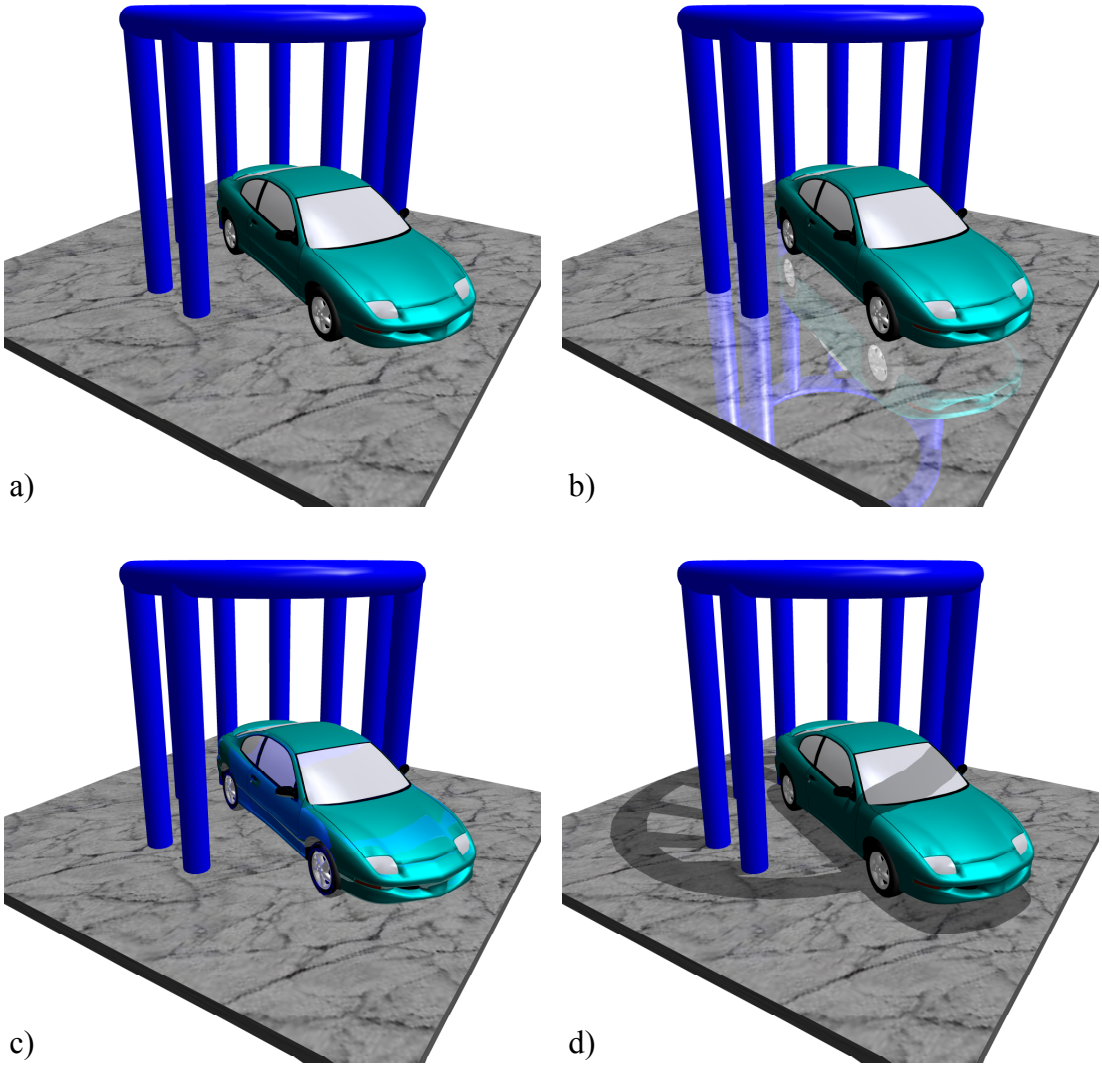

Abbildung 9: Beleuchtungseffekte erster Ordnung. Szene ohne Beleuchtungseffekt (a), mit planarer Reflexion auf der Bodenplatte (b), mit dynamischem Environment-Mapping des Autos (c) und mit harten Schatten (d).

ein weiteres Mal gerendert, wobei durch Blending das Verhältnis der Helligkeit des Reflektors und der gespiegelten Szene festgelegt wird.

Bei texturbasierten planaren Reflexionen wird zunächst die Szene von der gespiegelten Kameraposition in eine dynamische Textur gerendert. Dann erfolgt das Rendern der Szene wie gewöhnlich, außer dass für den planaren Reflektor zusätzlich die dynamische Textur auf den Reflektor projiziert und additiv hinzugefügt wird.

Verfahren zur Darstellung von dynamischem Environment-Mapping

Zur Darstellung von Environment-Mapping wird das Bild der Umgebung, von der reflektierenden Geometrie aus gesehen, in alle sechs Seiten einer dynamische Cubemap gerendert und vorgehalten. Dazu sind sechs Renderingdurchläufe erforderlich. Das Rendern der Szene erfolgt anschließend wie gewöhnlich, außer dass der Reflektor zusätzlich mit der Cubemap texturiert wird und die Texturwerte aus der Cubemap zum Bild addiert werden. Die Auswahl eines Texturwerts

aus der Cubemap erfolgt dabei mit Hilfe der interpolierten Normalen eines Fragments der reflektierenden Geometrie, durch die die Reflexionsrichtung festgelegt wird.

Texturbasierte Reflexionen mit Bumpmap

Bei texturbasierten Reflexionen und Environment-Mapping kann die Reflexion zusätzlich durch eine Bumpmap modifiziert werden. Die Normale, anhand derer der Reflexionsvektor berechnet wird, wird dabei zunächst durch eine Bumpmap modifiziert. Im Allgemeinen nutzt man hierbei dieselbe Bumpmap wie zur Berechnung der lokalen Beleuchtung. Mit stencilbasierten planaren Reflexionen ist eine Modifikation des Reflektionsvektors durch eine Bumpmap hingegen nicht möglich.

3.4.2 Harte Schatten

Schattenvolumen

Schattenvolumen markieren die von einer Lichtquelle beleuchteten Stellen als temporäre Information im Stencilpuffer [38]. Dazu wird zunächst der Tiefenpuffer der Szenengeometrie initialisiert. Dann werden unsichtbare Schattenvolumenpolygone, die die räumliche Berandung der 3D-Regionen in der Szene im Schatten kennzeichnen, in den Stencilpuffer gerendert. Dabei werden unter Verwendung von Back- und Front-face Culling bei vorderseitigen Polygonen vor dem Tiefenpuffer die Stencilwerte erhöht, bei rückseitigen Polygonen vor dem Tiefenpuffer werden sie dagegen erniedrigt. Alle beleuchteten Gebiete sind schließlich im Stencilpuffer mit dem Wert 0 gekennzeichnet, da sich dort vorder- und rückseitige Schattenvolumenpolygone aufheben. An den beleuchteten Stellen wird dann die Beleuchtung der Lichtquelle durch additives Blending hinzugefügt.

Schattenvolumen galten wegen mangelhafter Robustheit lange als unpraktikabel in Anwendungen. Vor allem mussten Schattenvolumenpolygone, die die Near- oder Far-clipping Ebene schneiden, innerhalb des View-Frustums begrenzt werden (das sogenannte *Capping*), eine geometrische Operation, die anfällig für Berechungs- und Rundungsfehler ist. Everitt und Kilgard haben schließlich robuste Techniken für Schattenvolumen beschrieben [23]: Zum einen verschieben sie die Far-clipping Ebene des View-Frustums durch eine spezielle Projektion in das Unendliche, wodurch für diese Ebene das Capping entfällt. Zum anderen formulieren sie die oben genannten Stenciloperationen äquivalent um, dass die Stencilwerte bei Schattenvolumenpolygonen *hinter* dem Tiefenpuffer erhöht oder erniedrigt werden. Dadurch ist Capping für die Near-clipping Ebene nicht mehr erforderlich.

Shadow-Mapping

Ein grundsätzlich anderer Algorithmus zur Berechnung von Schatten ist *Shadow-Mapping*. Shadow-Maps sind dynamische Tiefentexturen, die von der Position der Lichtquelle erzeugt werden. Wenn die Szene danach von der normalen Kameraposition aus gerendert wird, wird zur Schattenberechnung die Position eines Fragments in das Koordinatensystem der Lichtquelle transformiert und die Entfernung zur Lichtquelle mit der in der Tiefentextur abgespeicherten Entfernung verglichen. Stimmen die Werte überein, ist das Fragment von der Lichtquelle beleuchtet und kann entsprechend schattiert werden. Andernfalls muss die Entfernung in der Tiefentextur geringer sein, das heißt, ein Objekt befindet sich zwischen Lichtquelle und Fragment, so dass das Fragment im Schatten ist. Mit projektiven Texturen [74] und dem texturbasierten Schattentest, bzw. mit Fragmentprogrammen, kann diese Operation effizient durch die Graphikhardware erfolgen.

Hauptproblem von Shadow-Mapping ist die bildbasierte Abtastung der Shadow-Map, durch die Ungenauigkeiten in die Schattentestberechnung und dadurch Renderingartefakte in die Darstel-

lung einfließen. Dies fällt vor allem auf bei der – etwas irreführend als solche bezeichneten – *Selbstschattierung*, also dem Schattenwurf von Geometrie auf sich selbst. Diese Artefakte müssen durch eine *Tiefentoleranz* (depth bias) ausgeglichen werden, der für beliebige Szenen aber nur schwierig geeignet gewählt werden kann [88]. Zudem hat eine Shadow-Map nur eine begrenzte Auflösung, die im Bild durch Stufen in der Schattenberandung sichtbar wird. Diese Artefakte können durch Filtertechniken vermindert werden [68].

Eine weitere Schwierigkeit bei Shadow-Mapping ist die Wahl des Koordinatensystems der Lichtquelle. In den letzten Jahren wurde erkannt, dass gewöhnlich die Auflösung der Shadow-Map vor allem für kameranahe Geometrie zu gering ist. Dies führte zu zahlreichen Vorschlägen für eine diesbezüglich verbesserte Wahl des Koordinatensystems der Lichtquelle [77] [50] [91]. Bei Punktlichtquellen, die von allen Seiten von Geometrie umgeben sind, müssen unter Umständen mehrere Shadow-Maps in verschiedene Richtungen erzeugt werden, um die Schattenberechnung in alle Richtungen durchführen zu können. Ein praktikabler Ansatz hierzu ist Dual-Paraboloid Mapping, das mit geringen Verzerrungen die Erzeugung von nur zwei Shadow-Maps erfordert [9].

3.4.3 Weiche Schatten

Sowohl Schattenvolumen als auch Shadow-Mapping erzeugen harte, bei Shadow-Mapping gefilterte Schatten. Sie gehen also letztlich von dem unrealistischen Modell einer punktförmigen Lichtquelle ohne Ausdehnung aus. Das Problem der Darstellung weicher, durch Flächenlichtquellen entstehender Schatten ist in den letzten Jahren ein vielbearbeitetes Forschungsgebiet; einen aktuellen Überblick gibt [36]. Zusammengefasst gibt es derzeit kein Verfahren, das mit einem vertretbaren Aufwand physikalisch korrekte weiche Schatten in beliebigen, dynamischen Szenen erzeugen kann. Schwierig zu behandeln ist insbesondere die Selbstschattierung. Zudem wird die Form der Lichtquelle für weiche Schatten normalerweise eingeschränkt.

Als optische Referenz für weiche Schatten wird vielfach das Verfahren von Heckbert und Herf verwendet [37]. Es verteilt auf einer Flächenlichtquelle verschiedene Punktlichtquellen, führt für jede die Beleuchtungsberechnung durch und kombiniert die Ergebnisse zu Attenuations-Texturen, die für jeden Punkt aller Polygone in der Szene die prozentuale Helligkeitsinformation enthalten, von wie vielen Punktlichtquellen er beleuchtet ist. Für eine stufenlos wirkende Darstellung der Schatten sind sehr viele – mehrere 100 – Punktlichtquellen erforderlich, weshalb das Verfahren für eine dynamische Berechnung von Schatten nicht geeignet ist.

Single Sample Soft Shadows

Eine Klasse von Algorithmen für weiche Schatten sind solche, die den Schatten aufgrund einer Punktlichtquelle berechnen und die genaue Form einer Lichtquelle vernachlässigen. Diese *Single Sample Soft Shadow* Algorithmen verfolgen also nicht das Ziel physikalisch korrekter weicher Schatten, sondern streben lediglich die Darstellung überzeugend wirkender Schatten an. Von Brabec und Seidel stammt ein solches, auf Shadow-Mapping basierendes Verfahren [10]. Nach der Transformation eines Fragments in das Koordinatensystem der Lichtquelle berücksichtigen sie zur Schattenbestimmung auch die Nachbarschaft des transformierten Punkts in der Shadow-Map: Ist der Punkt beleuchtet, in der Nachbarschaft befinden sich aber Stellen im Schatten, wird der Punkt je nach Abstand zur nächsten Stelle im Schatten abgedunkelt. Ist der Punkt im Schatten, in der Nachbarschaft befinden sich aber beleuchtete Stellen, wird der Punkt entsprechend seinem Abstand zur nächsten beleuchteten Stelle aufgehellt. Die Nachbarschaftssuche in der Shadow-Map geschieht durch die CPU; das dazu notwendige Kopieren der Shadow-Map vom Graphik- in den Hauptspeicher ist der wesentliche Nachteil für die Geschwindigkeit des Verfahrens.

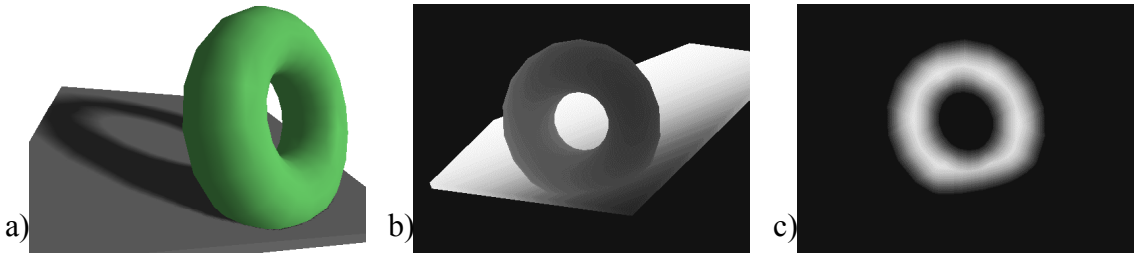

Abbildung 10: Torus mit Single Sample Soft Shadows (a). Zugehörige Shadow-Map (b). Zugehörige Shadow-Width-Map (c).

Implementierung durch die Graphikhardware

Eine Variante dieses Algorithmus kann vollständig auf der Graphikhardware implementiert werden [43]. Die Idee ist, für Punkte, die sich bei Betrachtung der Shadow-Map (Abbildung 10b) im Schatten befinden, den Abstand zur nächsten beleuchteten Stelle aus einer zweiten Textur, der Shadow-Width-Map (Abbildung 10c), auszulesen. Je nach diesem Abstand und je nach Entfernung des Punkts zum Schattenwerfer wird dann in einem Fragmentprogramm die Stärke des Schattens berechnet (Abbildung 11a). Punkte, die bei Betrachtung der Shadow-Map beleuchtet sind, werden immer voll beleuchtet. In der Praxis führt dies zu Schatten, die kleiner als die physikalisch zu erwartenden Schatten sind. Dennoch überzeugen die Schatten optisch (Abbildung 11b).

Die Erzeugung der Shadow-Width-Map durch die Graphikhardware ist die Hauptschwierigkeit bei dem Verfahren. Zu ihrer Erzeugung wird nur schattenwerfende Geometrie herangezogen. Diese wird bei der Erzeugung der Shadow-Map in einen unbenutzten, zunächst leeren Farbkanal mit einer Intensität von eins gerendert. Danach erfolgt eine Reihe von Filteroperationen, bei dem eine Textur, die den Farbkanal enthält, mehrfach auf den Framebuffer projiziert wird. Eine der Projektionen erfolgt dabei ohne Verschiebung oder Veränderung der Textur, die anderen Projektionen dagegen verschieben die Textur in verschiedene Richtungen von der Ursprungsposition weg, und beim Auslesen dieser Texturen wird zu jedem Texturwert ein Differenzbetrag addiert. Die Intensität eines Fragments ergibt sich aus dem Minimum aller Texturwerte. Diese Operation wird mehrfach wiederholt, wobei in jedem Schritt die Texturverschiebung und der addierte Differenzbetrag verdoppelt wird. Der Algorithmus wird beendet, sobald der Differenzbetrag größer als eins ist. Der Farbkanal enthält dann eine gute Approximation der Shadow-Width-Map.

Diskussion des Verfahrens

Durch eine Shadow-Width-Map berechnete weiche Schatten erzielen für einfache geometrische Szenen optisch überzeugende Ergebnisse. Sie sind für beliebig gekrümmte Geometrie im Schatten ohne Einschränkungen nutzbar. Sich im Koordinatensystem der Lichtquelle überlappende, schattenwerfende Geometrie-Objekte führen dagegen in der Regel zu Renderingartefakten in der Schattenberandung. Diese resultieren daher, dass in der Shadow-Map nur ein Wert für mehrere schattenwerfende Geometrie-Objekte abgelegt werden kann und darum in solchen Fällen die Entfernung eines Punkts im Schatten zum Schattenwerfer falsch berechnet wird. Aus dem gleichen Grund sollte Selbstschattierung mit diesem Verfahren vermieden werden.

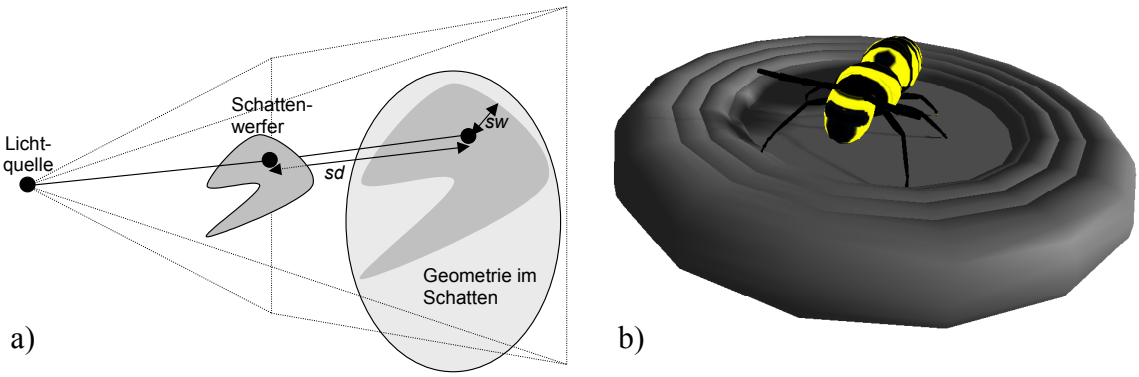

*Abbildung 11: Geometrische Interpretation des Werts aus der Shadow-Width-Map **sw** und des Abstands des Punkts im Schatten zum Schattenwerfer **sd** (a). Aus diesen beiden Werten wird die Stärke des Schattens berechnet. Modell eines Insekts, dargestellt mit Single Sample Soft Shadows (b).*

3.5 Erweiterungen des Kameramodells

Eine Reihe echtzeitfähiger Renderingeffekte befasst sich mit der Erweiterung oder Verbesserung des Kameramodells. Beim normalen Kameramodell wird die Szene von einer punktförmigen Position aus erfasst. Je nach Ausstattung und Leistungsfähigkeit der Hardware können aber komplexere Kameramodelle nötig oder sinnvoll sein.

Ein Beispiel hierfür ist *Stereo-Rendering*: Zur Unterstützung des stereoskopischen Sehens werden zwei Varianten eines Bilds mit unterschiedlichen Kameraeinstellungen für das linke und rechte Auge generiert. Dies erfordert geeignete Ausgabehardware mit getrennten Framebuffern für das linke und rechte Auge.

Der Begriff *Tiefenunschärfe* bezeichnet die Eigenschaft einer photographischen Kamera, nur für eine bestimmte, einstellbare Entfernung ein scharfes Bild zu erzeugen (Abbildung 12). Bilder mit diesem Effekt können mit einem globalen Multipass-Verfahren erzeugt werden, indem mehrere Einzelbilder von verschiedenen leicht verschobenen Kamerapositionen aus gerendert und im Akkumulationspuffer gleichgewichtet aufaddiert werden [34]. Für eine gute Bildqualität sind allerdings viele Einzelbilder erforderlich, wodurch das Verfahren in der Regel zu langsam für Echtzeitdarstellung ist. Daher gibt es bildbasierte Renderingverfahren, die Tiefenunschärfe als Nachbearbeitungseffekt durch eine Analyse des Tiefenpuffers und entsprechendes Anpassen des Farbpuffers approximieren [15].

Ein ähnlicher Effekt ist *Bewegungsunschärfe*: Auf Photos oder in Filmen erscheinen Objekte in Bewegungsrichtung je nach Verschlusszeit der Kamera unscharf. Dieser Effekt kann ebenfalls mit dem Akkumulationspuffer durch Aufaddieren mehrerer Einzelbilder erzeugt werden [34]. Alternativ kann Bewegungsunschärfe für ein Objekt durch zweimaliges Rendern des Objekts dargestellt werden: Zunächst wird das Objekt normal dargestellt, dann wird transparente Geometrie gerendert, die mit einem Vertexprogramm entlang der Bewegungsrichtung des Objekts generiert wird. [92].

Antialiasing bezeichnet die Reduzierung des Treppeneffekts, der bei der Rasterisierung schräger Linien oder Polygonkanten systemimmanent auftritt. Dieser Renderingeffekt kann ebenfalls durch das mehrfache Rendern der Szene, das sogenannte Supersampling, implementiert werden [8]: Die Kamera wird hierzu für jedes Einzelbild um weniger als ein Pixel verschoben und die generierten Bilder im Akkumulationspuffer aufaddiert. Mittlerweile bietet moderne Graphik-

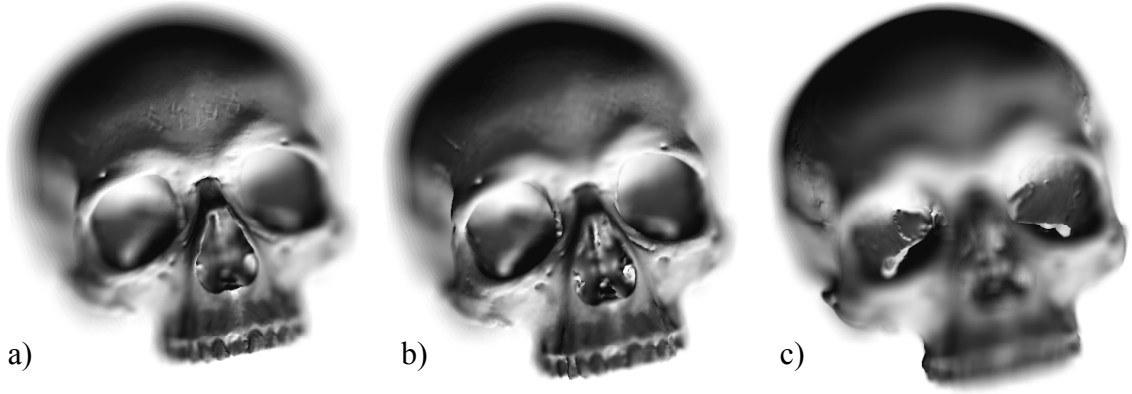

Abbildung 12: Erweiterung des Kameramodells. Ein Schädel mit unterschiedlicher Einstellung der Tiefenunschärfe: Fokus auf Nasenspitze (a), auf Höhe der Nasenhöhle (b), auf Rückseite der Augenhöhle (c).

hardware mit den zugehörigen Treibern die Möglichkeit, diese und andere Arten des Antialiasings applikationsübergreifend und ohne Vorkehrungen des Entwicklers einer Anwendung zu aktivieren. Daher wird Antialiasing heute nur noch selten explizit durch eine Anwendung implementiert.

3.6 Nachbearbeitungseffekte

Echtzeitfähige Renderingverfahren durch *Nachbearbeitung* (post processing) umfassen Verfahren, die bildbasiert auf ein bereits vollständig gerendertes Bild – und unter Umständen dem zugehörigen Tiefenpuffer – angewendet werden. Beispiele für so implementierbare Effekte sind Graustufenfilter, Sepia-Filter, Schärfungs- und Weichzeichnungsfilter [28], oder bildbasierte Kantenhervorhebung aufgrund von Unstetigkeiten im Tiefenpuffer. Der wesentliche Unterschied zu Oberflächenschattierungen ist, dass ein Nachbearbeitungseffekt immer auf das gesamte Bild, eine Oberflächenschattierung dagegen auf einzelne Geometrie in einem Bild wirkt.

Algorithmen für Nachbearbeitungseffekte sind vor allem aus der 2D-Bildverarbeitung bekannt. Durch bildbasierte Renderingverfahren werden sie aber zunehmend durch die Graphikhardware unterstützt und damit echtzeitfähig. Zur Implementierung wird der Framebuffer in eine Textur kopiert, welche dann unter Anwendung eines Fragmentprogramms, das eine Filteroperation implementiert, wieder auf den Framebuffer projiziert wird.

3.7 Anwendungsspezifische Verfahren

Eine große Anzahl von Renderingverfahren sind anwendungsspezifische Verfahren zur Darstellung und Animation einzelner Phänomene oder Stoffe, wie zum Beispiel zur Darstellung von Wasser, Bäumen, Pflanzen oder Fell. Sie werden einerseits benötigt, wenn die visuelle Komplexität eines Phänomens spezielle Renderingalgorithmen erfordert. Ein Beispiel ist die Darstellung von Wasseroberflächen, wo für optisch überzeugende Ergebnisse gleichzeitig Refraktion, Wasserspiegelung und Animation der Wellen simuliert werden muss. Andererseits können anwendungsspezifische Verfahren aus Geschwindigkeitsgründen zur Reduzierung der zur Graphikhardware gesendeten Geometrie nötig sein. Ein Beispiel dafür sind Renderingverfahren zur Echtzeitdarstellung von Bäumen oder Gras.

Einige anwendungsspezifische Renderingverfahren dienen zur Darstellung von 3D-Daten, die nicht in Form polygonaler 3D-Geometrie vorliegen und darum nicht direkt von der Graphikhardware dargestellt werden können. Ein Beispiel hierfür sind Volumendaten, die hardware-

beschleunigt durch texturbasiertes Volumenrendering visualisiert werden [21]. Ein anderes Beispiel ist CSG (Constructive Solid Geometry). Kapitel 7 befasst sich mit CSG und mit bildbasierten Renderingverfahren zur Darstellung von CSG-Modellen.

3.8 Kombination von Renderingeffekten

Bei der Kombination von echtzeitfähigen Renderingeffekten sind einerseits grundlegende technische Fragestellungen bei ihrer Auswertung zu betrachten. Andererseits muss gegebenenfalls zunächst geklärt werden, ob und wie die Kombination mehrerer Renderingeffekte semantisch sinnvoll ist.

3.8.1 Semantik

Für bestimmte Kombinationen von Renderingeffekten ist nicht definiert, wie das Ergebnisbild auszusehen hat. Zwei solche Renderingeffekte gleichzeitig für die gleiche Geometrie anzuwenden ist also semantisch nicht sinnvoll, oder die gemeinsame Wirkung muss zumindest genauer geklärt werden. In der Regel kann für konkrete Einzelfälle von Kombinationen leicht angegeben werden, ob sie semantisch definiert sind oder nicht:

- Die Kombination aller photorealistischen Renderingeffekte ist sinnvoll. Dazu gehören realistische Oberflächenschattierungen, Transparenz, Reflexionen und Schatten. Das zu erwartende Ergebnis wird im Zweifel definiert durch das physikalische Vorbild der realen Welt.
- Für eine NPR-Oberflächenschattierung, die Geometrie ohne Berücksichtigung einer Lichtquelle schattiert, ist die gleichzeitige Darstellung von Schatten nicht sinnvoll, da dieser sich immer auf eine Lichtquelle bezieht.
- Mehrere Oberflächenschattierungen können nicht sinnvoll für die gleiche Geometrie verwendet werden. Dazu ist nämlich zusätzlich eine Vorschrift erforderlich, wie die Farbwerte zweier Oberflächenschattierungen zu verknüpfen sind, wodurch aber technisch eine neue Oberflächenschattierung entsteht.
- Erweiterungen des Kameramodells können mit allen anderen Renderingeffekten kombiniert werden, da eine Kamera Teil jeder Szene ist.
- Nachbearbeitungseffekte können mit allen anderen Renderingeffekten kombiniert werden, weil sie auf dem Ergebnis der anderen Effekte, dem gerenderten Bild, operieren und damit nur 2D-Effekte darstellen. Nachbearbeitungseffekte sind nicht normalerweise nicht kommutativ, die Reihenfolge, in der mehrere Nachbearbeitungseffekte angewendet werden, beeinflusst also das Renderingergebnis.

Zu klären und gegebenenfalls zu definieren ist die Wirkung einer Kombination von Renderingeffekten, wenn ein beteiligter Effekt eine NPR-Oberflächenschattierung oder ein Markierungseffekt ist. Bewirken oder verändern diese Effekte einen Schatten? Kann eine NPR-Oberflächenschattierung spiegeln und wie sieht die Spiegelung aus? Wird ein Markierungseffekt gespiegelt? Arbeiten die Effekte zusammen mit Transparenz? Die Beantwortung dieser Fragen hängt letztlich vom einzelnen Effekt ab.

Renderingeffekt	Nachbearb'gseffekte	Erw. Kameramodelle	Reflexion	Schatten	Transparenz	Markierungseffekte	NPR-Oberflächen	PR-Oberflächen
PR-Oberflächen	+	+	+	+[1]	+	+	–	–
NPR-Oberflächen	+	+	?[2]	– / ?[2]	?[2]	+	–	
Markierungseffekte	+	+	+[3]	?[3]	– / ?[3]	? / +[3]		
Transparenz	+	+	+	+	+[4]			
Schatten	+	+	+	+[1]				
Reflexion	+	+	+[4]					
Erw. Kameramodelle	+	+[5]						
Nachbearb'gseffekte	+							

1) Kombinierbar für verschiedene Lichtquellen.
2) Abhängig vom NPR-Effekt, bei Schatten unter Umständen für nur eine Lichtquelle.
3) Je nach Markierungseffekt.
4) Für verschiedene Geometrie, sonst unklar.
5) Für verschiedene Kameramodell-erweiterungen, sonst nicht sinnvoll.

Abbildung 13: Semantische Kombinationsmatrix für eine Menge verschiedener Rendering-effekte. Die Einträge zeigen an, ob die Kombination zweier Effekte semantisch sinnvoll ist (+), ob dies unklar ist (?), oder ob sie nicht semantisch nicht definiert ist (–).

Zur besseren Übersicht kann in eine *semantische Kombinationsmatrix für Renderingeffekte* eingetragen werden, welche Kombinationen semantisch sinnvoll sind. Eine solche Matrix ist in Abbildung 13 abgebildet. Zu beachten ist bei dieser Darstellung, dass einige Effekte noch genauer unterteilt werden könnten. Bei Beleuchtungseffekten erster Ordnung wäre zusätzlich eine Unterscheidung zwischen Geometrie, die von einem Effekt betroffen ist, und Geometrie, die den Effekt auslöst, sinnvoll (zu dieser Unterscheidung siehe Abschnitt 5.5.1). Beispielsweise erwartet man vom Schatten eines schattenwerfenden Geometrie-Objekts mit NPR-Oberflächenschattierung auf normal beleuchtete Geometrie keine Besonderheiten. Problematisch ist hingegen die Darstellung nicht-photorealistischer Geometrie in einem Schatten.

3.8.2 Technische Restriktionen

Auch wenn viele Kombinationen von Renderingeffekten semantisch definiert sind, heißt dies noch nicht, dass auch alle Renderingverfahren zur Auswertung der Effekte zusammen funktionieren können. Durch technische Restriktionen kann eine gleichzeitige Anwendung mancher Verfahren unmöglich sein.

Objektbasierte Renderingverfahren sind anfällig für solche Inkompatibilitäten. Wie zum Beispiel der Schattenvolumenalgorithmus bei der Berechnung von Schattenvolumenpolygonen werten diese Verfahren die Geometrie von Objekten aus, um einen Renderingeffekt zu erzeugen. Solche Algorithmen können prinzipbedingt nicht mit bildbasierten Renderingverfahren zusammenarbeiten, die nur das Bild, nicht aber die geometrische Objektrepräsentation der dargestellten Geometrie erzeugen. Bildbasiertes CSG-Rendering und texturbasiertes Volumenrendering sind Beispiele für bildbasierte Verfahren, die daher grundsätzlich nicht mit Schattenvolumen kompatibel sind.

Die Auswertung von Transparenz ist im Echtzeitrendering problematisch. Dementsprechend ist die Zusammenarbeit mit anderen Renderingverfahren zum Teil noch ungeklärt. Insbesondere die realistische Behandlung von Schattenwurf transparenter Geometrie ist algorithmisch schwierig zu lösen.

Abbildung 14 zeigt eine *technische Kombinationsmatrix* für eine Reihe verschiedener Renderingverfahren. Insgesamt ist die überwiegende Anzahl der Renderingverfahren miteinander

Renderingeffekt	CSG Rendering	Tiefen-unschärfe (Akkum)	Textur-basierte Reflexion	Stencil-basierte Reflexion	Shadow-Mapping	Schat-tenvo-lumen	Ord-nungsu. Transp.	Bump-mapping
Bumpmapping	+	+	+	+	+	+	+	+[5]
Ordnungsu. Transparenz	+[1]	+	+	?[2]	?[3]	?[3]	+	
Schattenvolumen	–[3]	+	+	+[4]	+[5]	+[4,5]		
Shadow-Mapping	+	+	+	+	+[5]			
Stencilbasierte Reflexion	+[4]	+	+	+[4]				
Texturbasierte Reflexion	+	+	+					
Tiefenunschärfe (Akkum)	+	nicht sinnvoll						
CSG Rendering	+							

1) Technisch komplex, siehe Abschnitt 7.5.
2) Vermutlich möglich, technisch komplex.
3) Siehe Text.
4) Potentieller Ressourcenkonflikt im Stencilpuffer.
5) Kombination für verschiedene Lichtquellen möglich, sonst nicht sinnvoll.

Abbildung 14: Technische Kombinationsmatrix für eine Anzahl verschiedener Renderingverfahren. Die Einträge zeigen an, ob verschiedene Verfahren miteinander kombiniert werden können (+), ob es unklar ist (?), oder ob die Kombination der Verfahren aus grundsätzlichen Erwägungen nicht möglich ist (–).

kombinierbar. Die algorithmischen Probleme, die bei einer praktischen Implementierung gelöst werden müssen, sind komplex; insbesondere treten die folgenden Fragestellungen auf:

- **Ressourcenverwaltung.** Farb-, Tiefen-, und Stencilpuffer sind jeweils nur einmalig vorhanden, werden aber von allen Renderingverfahren benötigt. Deshalb muss sichergestellt werden, dass im Framebuffer abgelegte Zwischenergebnisse eines Renderingverfahrens nicht zwischenzeitlich von anderen Renderingverfahren überschrieben werden.
- **Kombination von Renderingeinstellungen.** Arbeiten mehrere Renderingverfahren in einem Renderingdurchlauf zusammen, beispielsweise ein Schattenverfahren und ein Oberflächenshader, müssen sie verschiedene Einstellungen des Renderingkontextes so vornehmen, dass die Zusammenarbeit korrekt erfolgt und keine Konflikte auftreten.
- **Kombination programmierbarer Shader.** Arbeiten mehrere Renderingverfahren in einem Renderingdurchlauf zusammen, von denen eines in diesem Durchlauf programmierbare Shader verwendet, muss ein neuer programmierbarer Shader konstruiert werden, der die für beide Verfahren nötigen Aufgaben erfüllt. Dies ist derzeit nur durch Handarbeit, nicht aber automatisiert möglich.
- **Integration des Beleuchtungsmodells.** Oberflächenshader, die virtuelle Lichtquellen auswerten, müssen zur Nutzung mit einem Schatten mit dem verwendeten Schattenverfahren zusammenarbeiten können. In gespiegelten Bildern muss die Oberflächenschattierung für die gespiegelte Geometrie ebenfalls genutzt werden.

Die Anzahl der Veröffentlichungen, die sich mit dieser Problematik befassen, ist bislang klein. Von Diefenbach stammt eine der wenigen Arbeiten, die sich mit der Integration einiger Verfahren, nämlich Schattenvolumen, stencilbasierten Reflexionen und Refraktionen in einer 3D-Szene befasst [16]. Er beschränkt sich dabei auf Fragestellungen zur Ressourcenverwaltung und zur Reihenfolge der Renderingdurchläufe.

Kapitel 4

HIERARCHISCHE SZENENBESCHREIBUNGEN

> *„(Scene graphs) allow programmers to shift their attention away from thinking about triangles and vertices to thinking about objects and their arrangement within a scene. They allow programmers to forget about controlling the rendering pipeline and instead think about content and how best to present it."*
>
> Henry Sowizral: „Scene Graphs in the New Millennium", 2000, [75].

Die Entwicklung von computergraphischen Anwendungen erfolgt zunehmend nicht mehr durch direkte Nutzung eines Immediate-Mode Renderingsystems. Ergänzend haben sich Retained-Mode Renderingsysteme entwickelt, die intern eine Repräsentation der dargestellten 3D-Szene vorhalten und deren Darstellung mit einem Immediate-Mode Renderingsystem implementieren.

Der Hauptvorteil der Entwicklung mit einem Retained-Mode Renderingsystem ist die stärkere Abstraktion von der Graphikhardware. Für den Anwendungsentwickler ist eine direkte Auseinandersetzung mit der Konfiguration der Renderingpipeline nicht erforderlich, stattdessen kann er die darzustellende Szene deklarativ festlegen. Dadurch vereinfachen Retained-Mode Renderingsysteme die Anwendungsentwicklung. Andererseits vermindern sie die Flexibilität, weil Algorithmen und Optimierungsstrategien, die nicht direkt unterstützt werden, sich nur mit einem großen Aufwand oder gar nicht integrieren lassen.

Die Datenhaltung der 3D-Szene in einem Retained-Mode Renderingsystem erfolgt in der Regel hierarchisch, also durch einen Baum oder einen gerichteten azyklischen Graphen. Diese zentrale Datenstruktur wird mit dem Begriff *Szenengraph* bezeichnet und ein entsprechendes Retained-Mode Renderingsystem als *Szenengraphsystem*. Die Beschreibung einer Szene durch einen Szenengraph ist die *Szenenbeschreibung*.

4.1 Historische Entwicklung von Szenengraphsystemen

Szenengraphverwandte Datenstrukturen gibt es schon seit den Anfängen der Computergraphik. Bereits im Jahre 1963 enthielt *Sketchpad* [84] eine hierarchische Szenenbeschreibung und konnte, um diese auszuwerten, geometrische Transformationen miteinander zu einer Gesamttransformation verknüpfen.

Aus Softwarearchitektursicht bestand lange Zeit keine Trennung zwischen Immediate-Mode und Retained-Mode Renderingsystemen. Beispielsweise erlaubt der historische 3D-Graphikstandard *PHIGS* [64] den prozeduralen Aufruf von Renderingfunktionen, unterstützt aber als wesentliches Merkmal editierbare, hierarchische *structure stores*. Diese sind Datenstrukturen zur Aufnahme von Geometrie, zugehörigen Transformationen und weiterer structure stores und können so als Vorläufer hierarchischer Knoten in einem modernen Szenengraphsystem angesehen werden.

Szenengraphsysteme

Die eigenständige Modellierung eines Retained-Mode Renderingsystems als Graphiksystem, das ein zugrundeliegendes Immediate-Mode Renderingsystemen für das Rendering nutzt, hat sich mit *Open Inventor* [81] von SGI durchgesetzt. Den grundsätzlichen Aufbau dieses objektorientierten, deklarativen Szenengraphsystems verwenden heute praktisch alle Retained-Mode Renderingsysteme.

Eine Art Industriestandard-Szenengraphsystem hat sich bis heute nicht etabliert. Stattdessen gibt es heute mehrere Open-Source Szenengraphsysteme, unter anderem *OpenSceneGraph* [60], *OpenSG* [69] und *VRS* [18]. Daneben gibt es Systeme wie *Java3D* [76] und *NVSG* [57], deren Quellcode nicht frei verfügbar ist. Java3D nimmt als einziges nicht für C++ entwickelte API eine Sonderstellung ein.

Szenendeklarationssprachen

Szenendeklarationssprachen ermöglichen die Beschreibung einer 3D-Szene in Textform. Das bekannteste Beispiel einer solchen Sprache ist *VRML* [87], das auf einer Weiterentwicklung des externen Dateiformats von Open Inventor für 3D-Szenen beruht. VRML hat sich als ein wichtiges Datenaustauschformat für Szenengraphsysteme und 3D-Anwendungen durchgesetzt. Der Nachfolger von VRML ist das XML-basierte *X3D* [24], welches 2004 den ISO-Standardisierungsprozess durchlaufen hat. X3D unterscheidet sich derzeit von VRML von der Funktionalität her nur unwesentlich, wird aber unabhängig weiterentwickelt.

4.2 Ziele bei der Entwicklung von Szenengraphsystemen

Trotz ihres ähnlichen Aufbaus sind verschiedene Szenengraphsysteme nicht für jede Anwendung gleich gut geeignet. Sie können beispielsweise optimiert sein in Hinblick auf

- einfache und bequeme Verwendung,
- hohe Renderinggeschwindigkeit,
- Erweiterbarkeit mit benutzerdefinierten Graphikelementen,
- Portabilität auf verschiedenen Betriebssystemen,
- Skalierbarkeit auf Mehrprozessormaschinen
- oder Verwendbarkeit mit verschiedenen Immediate-Mode Renderingsystemen.

Weil sich diese Entwicklungsziele in der Regel widersprechen, kann es ein optimales Szenengraphsystem für alle Einsatzzwecke nicht geben. Dadurch erklärt sich auch die Vielzahl der existierenden Szenengraphsysteme.

Einfache und bequeme Benutzung

Eine wesentliche Motivation für die Entwicklung mit einem Szenengraphsystem ist, die Entwicklung von 3D-Graphikanwendungen zu vereinfachen. Dies wird einerseits durch die Möglichkeit der deklarativen Beschreibung einer 3D-Szene erreicht, im Gegensatz zur Verwendung des prozeduralen APIs eines Immediate-Mode Renderingsystemens. Der Aufwand für die Erstellung oder Modifikation eines Teils einer Szenenbeschreibung unterscheidet sich bei verschiedenen Szenengraphsystemen: Einige Systeme erfordern den expliziten Aufruf von Methoden wie `beginEdit()` und `endEdit()`, bevor und nachdem ein Szenenknoten verändert wird. Bei anderen Systemen ist dies nicht erforderlich.

Andererseits implementieren Szenengraphsysteme viele Standardfunktionalitäten der Applikationsstufe, die der Anwendungsentwickler bei direkter Verwendung eines Immediate-Mode Renderingsystems selbst entwickeln müsste. Beispiele für solche Funktionen, die von verschiedenen Szenengraphsystemen unterstützt werden, sind:

- **Analytische Berechnungen** wie Schnittpunktberechnung der Szene mit einem Strahl oder Kollisionsabfragen;
- **Komfortfunktionen**, zum Beispiel zur einfachen, portablen Initialisierung des Immediate-Mode Renderingsystems;
- **Ein- und Ausgabefunktionen**, zum Beispiel zum Laden und Aufbereiten von 3D-Modellen und Bilddaten oder zum einfachen Erzeugen und Speichern von Bildschirm-Snapshots.

Optimierungen der Renderinggeschwindigkeit

Nahezu alle Szenengraphsysteme implementieren eine Vielzahl von Algorithmen zur Steigerung der Renderinggeschwindigkeit, und bei einigen Systemen ist das Erreichen einer größtmöglichen Geschwindigkeit allen anderen Designzielen übergeordnet [71]. Die Renderinggeschwindigkeit kann durch die folgenden Verfahren verbessert werden:

- **View-Frustum Culling**, das heißt die Nichtdarstellung von 3D-Geometrie außerhalb des Sichtbarkeitsbereichs, abhängig von einem objektbasierten Test des Graphiksystems;
- **Occlusion-Culling**, das heißt die Nichtdarstellung von 3D-Geometrie, die von anderer Geometrie verdeckt wird, abhängig von einem objektbasierten Test des Graphiksystems;
- **Multiresolutionsmodellierung** (Level-Of-Detail), das heißt die entfernungsabhängige Auswahl unterschiedlich detaillierter Geometriemodelle eines geometrischen Objekts;
- **State-Sorting**, das heißt die Umordnung der Szenenbeschreibung für das Rendering, so dass die Anzahl der Konfigurationsänderungen des Renderingkontextes zum Rendern eines Bilds minimiert wird;
- **Kompilation statischer Teilgraphen**, durch die statische Teile der Szene in eine Darstellung kompiliert werden, die von der Graphikhardware schneller ausgewertet werden kann, aber keine nachträglichen Änderungen mehr erlaubt.

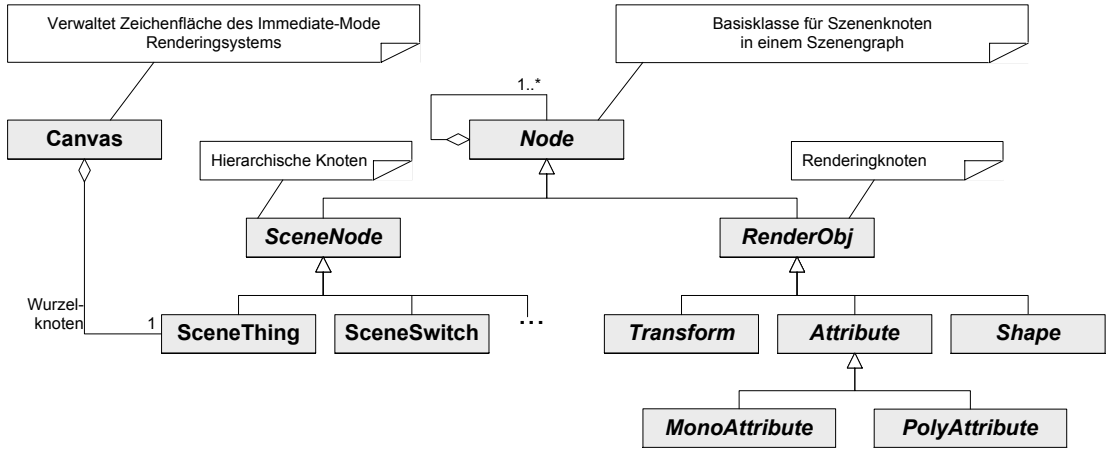

Abbildung 15: Klassendiagramm mit den Grundelementen einer Szenenbeschreibung.

4.3 Grundlegende Elemente einer Szenenbeschreibung

Eine Szenenbeschreibung wird durch eine Anzahl von Grundelementen, den *Szenenknoten*, deklariert, die in diesem Abschnitt vorgestellt werden. Abbildung 15 zeigt ein Klassendiagramm der beteiligten Basisklassen. Die Bezeichnungen der Basisklassen sind dem Szenengraphsystem VRS entnommen, die Konzepte unterscheiden sich aber nicht von anderen Szenengraphsystemen.

Geometrie

Ein *Geometrie*-Objekt, Instanz einer von der Basisklasse `Shape` abgeleiteten Klasse, beinhaltet geometrische Daten eines einzelnen geometrischen Körpers in einer 3D-Szene. Diese Daten können polygonal durch Dreiecksnetze oder analytisch durch eine mathematische Beschreibung des Körpers gegeben sein. Zu den geometrischen Daten gehören neben den Eckpunktpositionen auch die zugehörigen Normalen und Texturkoordinaten. Zur Auswertung echtzeitfähiger Renderingeffekte können auch weitere Arten von Eckpunktinformationen wie Tangenten oder Binormalen benötigt werden.

Transformationen

Durch *geometrische Transformationen* wird die Positionierung der Geometrie-Objekte im Koordinatensystem der Szene festgelegt. Transformationen in einer Szenenbeschreibung werden hierarchisch miteinander verknüpft, das heißt, die Wirkung einer geometrischen Transformation ist relativ zur vorigen Gesamttransformation.

Attribute

Attribut-Objekte beschreiben die Eigenschaften von Geometrie oder auch der gesamten Szene. Typische Beispiele sind Materialeigenschaften, Texturen oder Lichtquellen. Attribute wirken sich nur auf die nachfolgenden Geometrie-Objekte in einer Szenenbeschreibung aus. Verschiedene Attribut-Objekte vom gleichen Typ überschreiben sich dabei in der Regel; solche Attribute heißen *Monoattribute* und sind Instanzen einer von der Basisklasse `MonoAttribute` abgeleiteten Klasse. Instanzen einiger Attribut-Klassen, z.B. für Lichtquellen, sind im Gegensatz dazu parallel aktiv und werden darum als *Polyattribute* bezeichnet [18]. Die entsprechenden Klassen sind von der Basisklasse `PolyAttribute` abgeleitet.

In Hinblick auf die Abbildung eines Attributs auf das Immediate-Mode Renderingsystem kann zwischen verschiedenen Arten von Attributen unterschieden werden: *Konkrete Attribute* haben eine eins-zu-eins Abbildung auf eine Einstellung des Renderingkontextes. Solche Attribute sind daher oft technisch motiviert und haben dann keine anschauliche Bedeutung. *Aggregierende Attribute* werden durch eine eins-zu-n Abbildung auf eine Menge von Einstellungen des Renderingkontextes abgebildet, die zeitgleich aktiviert werden; als Beispiel könnte durch so ein Attribut zugleich eine Textur, die Texturkoordinatengenerierung und der Alphatest eingestellt werden. Solche Attribute können technisch, aber auch aus deklarativen Gründen nötig sein. Technisch vereinfachen sie das State-Sorting durch eine Verminderung der Anzahl der verschiedenen Attributtypen. Die Deklaration einer Szene wird vereinfacht, wenn durch ein aggregierendes Attribut logisch zusammengehörige Einstellungen des Renderingkontextes eingestellt werden.

Die dritte Art von Attributen sind *abstrakte Attribute*. Solche Attribute beschreiben eine Eigenschaft, die auf kompliziertere Art und Weise als das zeitgleiche Aktivieren einer Menge von Einstellungen des Renderingkontextes erzeugt werden muss. In Kapitel 5 wird gezeigt, dass abstrakte Attribute wesentlich für die Deklaration von Renderingeffekten in einer Szenenbeschreibung sind. Abstrakte Attribute erfordern kompliziertere Auswertungsalgorithmen als konkrete und aggregierende Attribute. Mit verschiedenen Ansätzen dazu befasst sich Kapitel 6.

Hierarchische Knoten

Ein *hierarchischer Knoten* strukturiert den Szenengraph hierarchisch, er hält also mehrere Kindknoten. Diese können Geometrie-, Transformations-, oder Attribut-Objekte sein oder wiederum hierarchische Knoten; ein solcher ist dann die Wurzel eines *Teilgraphen*. Attribute und Transformationen in einem hierarchischen Knoten wirken sich nur auf die im Knoten folgenden Geometrie-Objekte und Teilgraphen aus. Attribute und Transformationen innerhalb eines Teilgraphen beeinflussen darüber hinaus andere Teilgraphen nicht. Der Begriff *hierarchische Szenenbeschreibung* illustriert, dass wegen dieser Eigenschaften Teilgraphen in einen bestehenden Szenengraph ohne Seiteneffekte eingefügt oder entfernt werden können.

Hierarchische Knoten haben darüber hinaus oft keine semantische Bedeutung. Solche allgemeinen Knoten sind in VRS Instanzen der Klasse `SceneThing`, und beispielsweise der Wurzelknoten des Szenengraphen ist ein solches Objekt. Ein Beispiel eines hierarchischen Knotens mit zusätzlicher Logik ist ein Verzweigungsknoten der Klasse `SceneSwitch`, bei dem, abhängig von bestimmten Umständen bei der Auswertung der Szenenbeschreibung, nur ein Teilgraph berücksichtigt wird.

4.4 Nicht-abstrakte Deklaration echtzeitfähiger Renderingverfahren

Viele auf C++ basierende Szenengraphsysteme stellen über konkrete Attribute sämtliche Einstellungen des Immediate-Mode Renderingsystems als Szenenknoten zur Verfügung. Damit erlauben sie es grundsätzlich, beliebige Renderingverfahren in einer Szenenbeschreibung anzugeben und damit zu implementieren. Dies bezeichnen wir als *nicht-abstrakte Deklaration echtzeitfähiger Renderingverfahren*. In diesem Abschnitt wird gezeigt, dass in Bezug auf die einfache Nutzung eines Renderingeffekts diese Vorgehensweise keine Vorteile gegenüber der direkten Verwendung des Immediate-Mode Renderingsystems hat.

Beispiel: Stencilbasierte Reflexion

Als Beispiel zur nicht-abstrakten Deklaration von Renderingverfahren dient ein Algorithmus zur Darstellung einer planaren Reflektion mit dem Stencilpuffer. Abbildung 16 zeigt eine vereinfachte Implementierung eines solchen Verfahrens, wie in Kapitel 3.4.1 beschrieben, für

```
glEnable(GL_STENCIL_TEST)

// draw mirroring floor with stencil value 1
glStencilFunc(GL_ALWAYS, 1, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);

drawFloor();

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

// draw the spacecraft model, not its reflection
glStencilFunc(GL_ALWAYS, 0, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

drawSpaceCraft();

// set up the depth to the furthest depth value
glStencilFunc(GL_EQUAL, 1, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);

glDepthRange(1.0, 1.0);
glDepthFunc(GL_ALWAYS);

drawFloor();

glDepthRange(0.0, 1.0);
glDepthFunc(GL_LESS);

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

// draw the reflection
glStencilFunc(GL_EQUAL, 1, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

glPushMatrix();
  glScalef(1.0, -1.0, 1.0);
  drawSpaceCraft();
glPopMatrix();
```

```
// draw the mirror, blending it with reflection
glDepthFunc(GL_ALWAYS);

glStencilFunc(GL_EQUAL, 1, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glColor4f(0.7, 0.7, 0.7, 0.5);

drawFloor();

glDisable(GL_BLEND);
glDepthFunc(GL_LESS);
glDisable(GL_STENCIL_TEST);
```

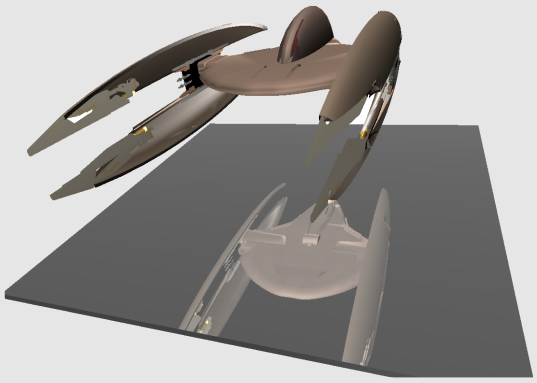

Abbildung 16: Quellcode in OpenGL zum Rendern eines Raumschiffmodells, das in einem ebenen Boden reflektiert wird. Dies ist ein einfaches Beispiel für ein Multipass-Verfahren

OpenGL. Abbildung 17 zeigt in Auszügen eine äquivalente Implementierung mit dem Szenengraphsystem OpenSceneGraph, dabei sind korrespondierende Teile durch gleichlautende Kommentare gekennzeichnet. Insgesamt deklariert OpenSceneGraph damit die in Abbildung 18 abgebildete Szenenbeschreibung.

Der Vergleich zeigt, dass zu jeder prozeduralen Änderung einer Renderingeinstellung in OpenGL ein entsprechendes Attribut-Objekt von OpenSceneGraph erzeugt wird. Durch die Trennung von Erzeugung dieser Objekte und ihrer Anordnung im Szenengraphen ist der Quellcode für den Algorithmus mit OpenSceneGraph umfangreicher. Außerdem fällt auf, dass in OpenSceneGraph explizit die Reihenfolge des Algorithmus durch die Methode `setRenderBinDetails(int binNum, const std::string)` festgelegt wird. Dies verhindert bei der Szenengraphauswertung Umordnungen der Renderreihenfolge aufgrund von State-Sorting. Bei einer prozeduralen Deklaration des Algorithmus mit OpenGL ist dies nicht erforderlich.

Bewertung der nicht-abstrakten Deklaration von Renderingeffekten

Die Probleme, die sich durch die nicht-abstrakte Deklaration von Renderingeffekten in einer hierarchischen Szenenbeschreibung ergeben, sind die folgenden:

- **Komplexe Implementierung.** Nach wie vor sind fundamentale Kenntnisse über die Graphikhardware erforderlich. Die Implementierung eines Renderingverfahrens erfolgt auf der gleichen Abstraktionsebene wie direkt mit einem Immediate-Mode Renderingsystem und

```
osg::MatrixTransform* rootNode = new osg::MatrixTransform;

{ // draw mirroring floor with stencil value 1
    osg::Stencil* stencil = new osg::Stencil;
    stencil->setFunction(osg::Stencil::ALWAYS, 1, ~0);
    stencil->setOperation(osg::Stencil::KEEP, osg::Stencil::KEEP, osg::Stencil::REPLACE);

    osg::ColorMask* colorMask = new osg::ColorMask;
    colorMask->setMask(false, false, false, false);

    osg::StateSet* statesetBin1 = new osg::StateSet();
    statesetBin1->setRenderBinDetails(1, "RenderBin");
    statesetBin1->setAttributeAndModes(stencil, osg::StateAttribute::ON);
    statesetBin1->setAttribute(colorMask);

    osg::Geode* geode = new osg::Geode;
    geode->addDrawable(floorFacetGeometry);
    geode->setStateSet(statesetBin1);

    rootNode->addChild(geode);
}

{ // draw the spacecraft model, not its reflection
    osg::Stencil* stencil = new osg::Stencil;
    stencil->setFunction(osg::Stencil::ALWAYS, 0, ~0);
    stencil->setOperation(osg::Stencil::KEEP, osg::Stencil::KEEP, osg::Stencil::REPLACE);

    osg::StateSet* statesetBin2 = new osg::StateSet();
    statesetBin2->setRenderBinDetails(2, "RenderBin");
    statesetBin2->setAttributeAndModes(stencil,osg::StateAttribute::ON);

    osg::Group* groupBin2 = new osg::Group();
    groupBin2->setStateSet(statesetBin2);
    groupBin2->addChild(spacecraftGeometry);

    rootNode->addChild(groupBin2);
}

  // set up the depth to the furthest depth value
...
  // draw the reflection
...
  // draw the mirror, blending it with reflection
...
```

Abbildung 17: Ausschnitt eines Quellcodes für das Szenengraphsystem OpenSceneGraph zum Rendern eines Modells, das in einer ebenen Bodenfläche reflektiert wird.

erfordert also die gleichen Kenntnisse über den verwendeten Algorithmus und über die Graphikhardware. Es müssen sogar zusätzliche Schwierigkeiten gelöst werden, etwa zur Deaktivierung von internen Optimierungen wie State-Sorting bei der Auswertung des Szenengraphen: Durch diese könnte die Reihenfolge von Geometrie für das Rendering umgestellt werden, wodurch eine korrekte Funktion des Multipass-Verfahrens nicht mehr gewährleistet wäre.

- **Fehlende Ausdruckskraft.** Ohne Kommentare oder genaue Analyse ist dem Quellcode nicht anzusehen, was für einen Renderingeffekt er bewirkt.
- **Schlechte Wartbarkeit.** Dies liegt zum einen an der Komplexität des Algorithmus. Des weiteren wird die Wartbarkeit dadurch vermindert, dass dieselben Geometrie-Objekte sich mehrfach in der Szenenbeschreibung befinden. Im Beispiel ist die `floorFacetGeometry` dreimal im Szenengraph vorhanden und die `spacecraftGeometry` zweimal. Für alle echtzeitfähigen Renderingverfahren, die durch Multipass-Rendering implementiert werden, das heißt Geometrie-Objekte mehrfach rendern, tritt dieses Problem auf.
- **Keine Wiederverwendbarkeit.** Die Geometriedaten und die prozedurale Beschreibung des Renderingverfahrens sind in der Implementierung nicht voneinander getrennt, daher muss der Quellcode kopiert und angepasst werden, um andere Geometrie zu unterstützen.

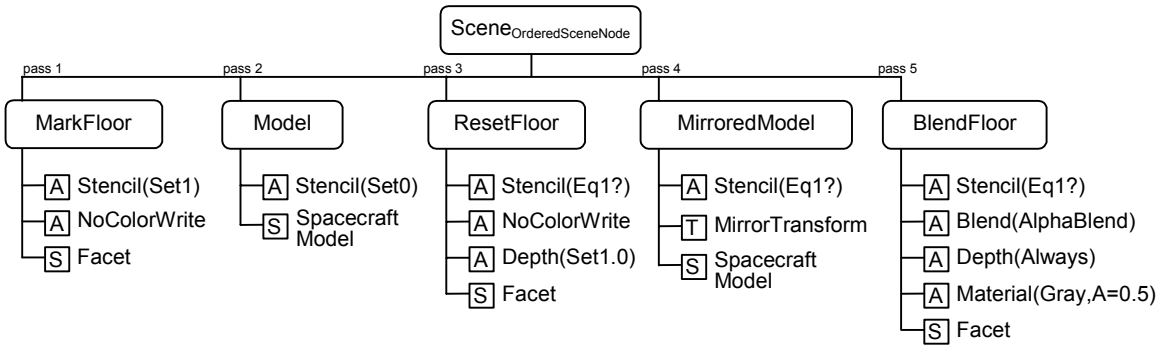

Abbildung 18: Szenengraph zur Deklaration eines stencilbasierten Reflexionsverfahrens.

- **Kombination von Renderingverfahren.** Die technischen Probleme bei der Kombination mehrerer Renderingverfahren (Abschnitt 3.8.2) werden nicht vermindert. Der Anwendungsentwickler muss selber Renderingressourcen aufteilen und die Gesamtreihenfolge der Renderingdurchläufe festlegen. Zum Beispiel muss zur gleichzeitigen Darstellung von zwei planaren Spiegeln mit einem stencilbasierten Reflexionsverfahren für jeden Spiegel ein eigener Stencilwert verwendet werden, und die einzelnen Renderingdurchläufe zum Rendern der Spiegelgeometrie müssen jeweils hintereinander erfolgen. Der Anwendungsentwickler muss dies technisch in der prozeduralen Beschreibung des Renderingverfahrens im Szenengraph umsetzen.

Zusammengefasst erlaubt ein Szenengraphsystem durch nicht-abstrakte Deklaration von Renderingverfahren grundsätzlich die Implementierung von Renderingeffekten. Durch die fehlende Abstraktion muss der Anwendungsentwickler dazu aber dieselben komplexen technischen Fragestellungen wie bei direkter Verwendung eines Immediate-Mode Renderingsystems lösen. Im Widerspruch zum Zitat von Sowrizal zu Beginn dieses Kapitels muss sich der Anwendungsentwickler insbesondere explizit um die Kontrolle der Renderingpipeline kümmern.

Kapitel 5

ABSTRAKTE DEKLARATION ECHTZEITFÄHIGER RENDERINGEFFEKTE

Die nicht-abstrakte Deklaration von Renderingverfahren (Abschnitt 4.4) ist trotz ihrer Nachteile zur Implementierung von Renderingeffekten mit Szenengraphsystemen häufig anzutreffen. In gängigen Systemen ist für die meisten Renderingeffekte die einzige Alternative der Verzicht auf die Verwendung des Effekts. VRML, X3D oder Java3D erlauben keinen direkten Zugriff auf alle Einstellungen der Renderingpipeline, und daher ist mit ihnen die Verwendung vieler Renderingeffekte, die im Echtzeitrendering mittlerweile wesentlich für qualitativ hochwertige Graphik erforderlich sind, nicht möglich.

Zur erleichterten Verwendung von Renderingeffekten sind Elemente in einer Szenenbeschreibung erforderlich, die es ermöglichen, einen Renderingeffekt ohne Bezugnahme auf direkte Einstellungen der Renderingpipeline zu seiner Auswertung zu deklarieren. Die Wirkung eines Effekts auf die Szene, bzw. auf einzelne Geometrie in der Szene, wird demnach abstrahiert von der Graphikhardware angegeben. Eine zentrale Rolle haben hierbei abstrakte Attribute, die in diesem Kapitel zur Deklaration der im Kapitel 3 vorgestellten Renderingeffekte verwendet werden. So können Anwendungsentwickler diese Effekte auch ohne Kenntnisse der jeweiligen Renderingverfahren zu ihrer Auswertung in einer Szenenbeschreibung angeben. Mit Verfahren in einem Szenengraphsystem zur Auswertung abstrakter Attribute befasst sich Kapitel 6.

5.1 Anforderungen

Eine Lösung zur Deklaration von Renderingeffekten in einer Szenenbeschreibung muss die folgenden Eigenschaften erfüllen:

- **Abstrakte Deklaration von Renderingeffekten.** Renderingeffekte sind als Teil der Szenenbeschreibung deklariert, das heißt, es wird angegeben, welcher Renderingeffekt welche Geometrie in der Szene beeinflusst. Die Implementierung der zugehörigen Renderingverfahren dagegen befindet sich nicht in die Szenenbeschreibung, sondern sie wird intern vom Szenengraphsystem vorgehalten und gegebenenfalls verwendet.
- **Granulare Auswahl der Geometrie.** Ein Renderingeffekt muss für einzelne Geometrie mit der gleichen Flexibilität wie mit einem Immediate-Mode Renderingsystem einstellbar sein. Dies ist für Beleuchtung erster Ordnung eine schwierig zu erfüllende Anforderung. In Abschnitt 5.5.1 wird dieser Sachverhalt erörtert.
- **Redundanzfreiheit.** Geometrie, die nur einmal in einer Szene vorhanden ist, darf nur einmal in der Szenenbeschreibung vorkommen.
- **Kombination von Renderingeffekten.** Die gleichzeitige Deklaration mehrerer semantisch definierter Renderingeffekte für dieselbe Geometrie muss möglich und nachvollziehbar sein.
- **Unterstützung verschiedener Renderingverfahren.** Die Szenengraphauswertung soll in der Lage sein, aus verschiedenen Verfahren zur Darstellung eines Renderingeffekts ein geeignetes auszuwählen, auf der anderen Seite soll die Auswahl des Renderingverfahrens auch in der Szenenbeschreibung oder durch eine andere Konfigurationsmöglichkeit einstellbar sein.

5.2 Deklaration von Oberflächenschattierungen

Eine Abstraktion von Oberflächenschattierungen von beliebigen Anwendungen aus erlauben Bibliotheken wie *D3DX Effects* [61] oder *CgFX* [12]. Beide erlauben skriptbasiert die Benennung einer Oberflächenschattierung und die Implementierung von verschiedenen zugehörigen Oberflächenshadern, auch als lokale Multipass-Verfahren, je nach Fähigkeiten der unterstützten Graphikhardware. Die APIs dieser Bibliotheken unterstützen die folgenden Funktionen:

- Bestimmen der am Besten geeigneten Technik zur Auswertung einer Oberflächenschattierung, abhängig von der installierten Hardware;
- Iterieren über alle von der Technik definierten Durchläufe;
- Setzen der skriptdefinierten Einstellungen des Renderingkontextes für einen Durchlauf;
- Zurücksetzen dieser Einstellungen nach Durchführung aller Durchläufe.

Wegen der expliziten Formulierung der Schleife wird also keine vollständige Trennung von Deklaration und Auswertung einer Oberflächenschattierung erreicht, aber zumindest wird Anwendungsentwickler nicht direkt mit einzelnen Einstellungen der Renderingpipeline konfrontiert.

Shaderattribute

Hierarchische Szenenbeschreibungen können leicht um abstrakte Attribute erweitert werden, die den Namen einer Oberflächenschattierung enthalten. Ein solches *Shaderattribut* spezifiziert, dass die darauffolgende Geometrie mit der Oberflächenschattierung dargestellt wird. Zu beachten ist, dass nur ein einziges Shaderattribut gleichzeitig auf eine Geometrie wirkt, das heißt, es handelt sich um Monoattribute, die sich einander überschreiben. Szenengraphsysteme mit Unterstützung von Shaderattributen sind NVSG, bei dem die Umsetzung auf Basis von CgFX erfolgt, OpenSceneGraph mit einem ähnlichen, *osgFX* genannten Mechanismus, und VRS.

Ein Beispiel für ein Shaderattribut in VRS ist das Attribut `Bumpmap` zur Darstellung von Geometrie mit Bumpmapping. Zusätzlich zur Bumpmap-Textur zur Störung der Normalen enthält ein solches Attribut-Objekt Verweise auf die Lichtquellen in der Szenenbeschreibung, die für das Bumpmapping bei der Beleuchtungsberechnung berücksichtigt werden. Weitere Shaderattribute in VRS dienen zur Deklaration von NPR-Oberflächenschattierungen.

Zusammenarbeit mit anderen Renderingeffekten

Shaderattribute in VRS wirken ähnlich wie eine Materialeigenschaft. Bei der gleichzeitigen Verwendung von Beleuchtung erster Ordnung in einer Szenenbeschreibung wirken sich Shaderattribute auch auf gespiegelte Geometrie aus, das heißt, die in einem Spiegel sichtbare Geometrie wird ebenfalls mit der entsprechenden Oberflächenschattierung dargestellt. Ähnlich werden photorealistische Oberflächenschattierungen zusammen mit Schatten behandelt: Verwenden sie, wie Bumpmapping, virtuelle Lichtquellen aus der Szenenbeschreibung, wird ein zugehöriger Schatten auf der jeweiligen Geometrie ebenfalls dargestellt, die Oberflächenschattierung also vom Schatten beeinflusst. Mit einer Oberflächenschattierung für NPR-Darstellung sind dagegen Schatten oft semantisch nicht definiert, und sie werden daher ignoriert.

5.3 Deklaration von Markierungseffekten

Ein Abstraktion von Markierungseffekten als eigenständige Graphikelemente von Szenengraphsystemen gibt es derzeit allenfalls durch Auffassen dieser Effekte als Spezialfälle von Oberflächenshadern. Eine Verwendung gleichzeitig mit anderen Oberflächenschattierungen in Systemen wie NVSG oder OpenSceneGraph ist allerdings so nicht möglich; damit ist diese Lösung nicht ausreichend.

Markierungsattribute

In hierarchischen Szenenbeschreibungen können Markierungseffekte, ähnlich wie Shaderattribute, durch abstrakte Monoattribute deklariert werden, die sich auf die nachfolgende Geometrie auswirken. Im Unterschied zu Shaderattributen können mehrere *Markierungsattribute* unterschiedlichen Typs gleichzeitig aktiv sein. Markierungsattribute des gleichen Typs überschreiben sich hingegen im Regelfall.

In VRS gibt es die folgenden Markierungsattribute: Ein `TextureEdges`-Attribut spezifiziert, dass die folgende Geometrie mit texturverstärkten Kanten dargestellt wird. Ein `Halo`-Attribut bewirkt die Hervorhebung von Geometrie durch ein Halo. Ein `Silhouette`-Attribut hebt Geometrie durch Umrandung mit einer Signalfarbe hervor.

Zusammenarbeit mit anderen Renderingeffekten

Die Zusammenarbeit von Markierungseffekten mit Beleuchtung erster Ordnung kann nur für jeden Einzelfall festgelegt werden (Abschnitt 3.8.1). Die Darstellung des Schattens eines Halos oder einer Umrandung kann nicht sinnvoll definiert werden. Im Unterschied dazu wäre der Schatten von Geometrie mit texturverstärkten Kanten entsprechend verstärkt darstellbar, wodurch eine visuelle Zuordnung von Geometrie und jeweiligem Schatten erleichtert würde. Dennoch werden in VRS Schatten und Markierungseffekte immer ohne gegenseitige Beeinflussung dargestellt. Bei Reflexionen ist die Übereinstimmung der direkt sichtbaren Geometrie mit dem Spiegelbild weit wichtiger. Oft ist gespiegelte Geometrie nur als solche erkennbar, wenn sie in allen visuellen Attributen mit der Originalgeometrie übereinstimmt. Daher werden Markierungsattribute auch auf in einem Spiegel sichtbare Geometrie angewendet.

In der Frage, wie Einstellungen zur Transparenz mit Markierungseffekten zusammenwirken, gibt es zwei verschiedene Möglichkeiten: Entweder der Markierungseffekt wird mit der glei-

chen Transparenz wie die markierte Geometrie angezeigt. Dies ist allerdings problematisch für Markierungseffekte, die selber Transparenz beinhalten. Oder der Markierungseffekt wird unverändert dargestellt, das heißt, als wäre die markierte Geometrie nicht transparent. In VRS wird die zweite Variante genutzt.

Logische und physische Markierungsattribute

Markierungseffekte werden im Allgemeinen nicht zur direkten Verbesserung des Realismus einer Graphikdarstellung verwendet. Stattdessen dienen sie zum verbesserten Verständnis einer Szene und zur Vereinfachung der Interaktion mit einer Szene, dadurch, dass sie Geometrie-Objekte mit einer bestimmten Eigenschaft markieren oder hervorheben. Damit werden Markierungseffekte normalerweise zur logischen Strukturierung von Geometrie und nicht zur direkten graphischen Attributierung eingesetzt.

Hier kann eine Analogie zu HTML als Hypertext-Beschreibungssprache gezogen werden. HTML unterscheidet „*logische und physische Elemente zur Auszeichnung von Text. Physische Textauszeichnungen haben Bedeutungen wie "fett" oder "kursiv", stellen also direkte Angaben zur gewünschten Schriftformatierung dar. [...] Logische Textauszeichnungen haben Bedeutungen wie "betont" oder "emphatisch"*“ [54]. In einer 3D-Szenenbeschreibung kann für Markierungseffekte ähnlich zwischen der physischen, unmittelbaren Auswirkung eines Effekts und seiner logischen Bedeutung unterschieden werden. Daher ist es sinnvoll, Markierungseffekte zusätzlich durch *logische Attribute* deklarieren zu können, die Bedeutungen wie „betont“, „ausgewählt“ oder „geplant“ transportieren. Die interne Abbildung auf *physische Attribute* erfolgt dann intern durch die Szenengraphauswertung. Bei auf X3D basierenden Szenenbeschreibungen ist hier zusätzlich, analog zu HTML, ein auf CSS basierender Ansatz denkbar zur Definition, wie logische Attribute durch physische Attribute repräsentiert werden.

5.4 Deklaration von Transparenz

Transparenz kann als einer von wenigen Renderingeffekten schon seit langem in den meisten Szenengraphsystemen abstrakt deklariert und ausgewertet werden: Die Transparenz ist Instanzvariable eines Material-Attributs und somit Teil der Materialeigenschaften von Geometrie. In VRS heißt das entsprechende Monoattribut `ShapeMaterialGL`; es erlaubt die Angabe der Transparenz über den Alphawert der Materialfarbe.

Bei der Zusammenarbeit von Transparenz mit anderen Renderingeffekten ist zu beachten, dass die jeweiligen Renderingverfahren meist nur schlecht mit dem Standard-Transparenzverfahren zusammenarbeiten. Daher wird für andere Renderingeffekte in VRS die Entstellungen der Transparenz ignoriert, auch wenn aus Sicht der Klarheit der Szenendeklaration eine andere Behandlung mitunter wünschenswert wäre.

Logische und physische Transparenz

Transparenz kann eine physische, aber auch eine logische Bedeutung haben. Manche 3D-Objekte in der Realität sind schlicht transparent, so dass die Deklaration von Transparenz zur Verbesserung der Realitätstreue einer 3D-Szene dient. Transparenz kann aber auch zur logischen Einteilung von Geometrie-Objekten genutzt werden, etwa zur Visualisierung von verschiedenen möglichen Planungsalternativen durch transparente Darstellung der verschiedenen Alternativen (Abschnitt 3.3.3). Zur logischen Deklaration verschiedener Alternativen enthält VRS das Monoattribut `Alternative`, das eine ID zur Unterscheidung verschiedener Alternativen in einer Szenenbeschreibung und die Stärke der Transparenz enthält. Sämtliche Geometrie-Objekte mit einem `Alternative`-Attribut mit derselben ID werden als eine zusammengehörige

Planungsalternative aufgefasst, die mit dem Verfahren zur Transparenzdarstellung von Alternativen aus Abschnitt 3.3.3 transparent dargestellt wird.

5.5 Deklaration von Beleuchtung erster Ordnung

Die wenigen bisherigen Ansätze zur Deklaration von Beleuchtung erster Ordnung in hierarchischen Szenenbeschreibungen verwenden hierarchische Szenenknoten mit spezieller Semantik. Die Verwendung abstrakter Attribute hat diesem Ansatz gegenüber den Vorteil, die abstrakte und redundanzfreie Deklaration von Beleuchtung erster Ordnung bei einer granularen Auswahl der Geometrie und für mehrere Beleuchtungseffekte gleichzeitig zu ermöglichen.

5.5.1 Granulare Auswahl von Geometrie.

Die meisten Renderingeffekte, wie Oberflächenschattierungen, Markierungseffekte oder Transparenz, können durch jeweils ein einziges Attribut pro Geometrie deklariert werden. Der Einflussbereich von einem Beleuchtungseffekt erster Ordnung lässt sich hingegen nicht präzise durch eine einzige Einstellung in einer Szenenbeschreibung angeben. Dies ist eine für Lichtquellen bereits bekannte Problematik [1]: die Position eines Lichtquellenknotens im Szenengraph kann einerseits die Transformation der Lichtquelle anzeigen. Zum anderen kann sie auch die Geometrie kennzeichnen, die im Einflussbereich der Lichtquelle ist, dadurch, dass die Geometrie im Szenengraph unterhalb der Lichtquelle eingehängt wird. Das Szenengraphsystem OpenSG erlaubt die Auflösung dieser Mehrdeutigkeit, indem eine Lichtquelle einen Verweis auf einen anderen, sogenannten *beacon*-Knoten enthalten kann, welcher die Transformationsmatrix der Lichtquelle definiert.

Wirkungseigenschaften und Ursacheneigenschaften

Bei Beleuchtungseffekten erster Ordnung muss zwischen zwei unterschiedlichen Arten von Eigenschaften unterschieden werden: Eine *Wirkungseigenschaft* bewirkt die direkte Veränderung der Schattierung von Geometrie, für die sie aktiv ist; die entsprechende Geometrie heißt *betroffen*. Eine *Ursacheneigenschaft* bewirkt, dass mit ihr markierte, sogenannte *handelnde* Geometrie die Schattierung von anderer Geometrie in der Szene beeinflusst. Diese Eigenschaften sind voneinander unabhängig: Geometrie kann sowohl betroffen und handelnd, nur betroffen, nur handelnd, oder keins von beiden sein.

Im Beispiel von Schatten wird betroffene Geometrie mit einem Schatten dargestellt. Dieser Schatten wird durch andere, handelnde Geometrie zwischen Lichtquelle und betroffener Geometrie verursacht. Bei einer Reflexion ist die betroffene Geometrie spiegelnd, während handelnde Geometrie in dem Spiegel reflektiert wird.

Mit Immediate-Mode Renderingsystemen ist die Trennung von betroffener und handelnder Geometrie einfach zu implementieren: In den entsprechenden Renderingdurchläufen wird jeweils nur betroffene bzw. handelnde Geometrie gerendert, was implizit durch die prozedurale Implementierung des Renderingverfahrens geschieht. Dabei ist eine solche Trennung von betroffener und handelnder Geometrie im Echtzeitrendering häufig anzutreffen und geschieht aus den folgenden Gründen:

- **Teilweise statische Beleuchtung**. In vielen Anwendungen wird für statische Teile einer Szene die Beleuchtung mit Schatten vorberechnet. Für diese Teile der Szene darf die Beleuchtung nicht nochmals durch einen dynamischen Schattenalgorithmus moduliert werden, denn der Schatten würde zweimal und damit falsch angewendet.

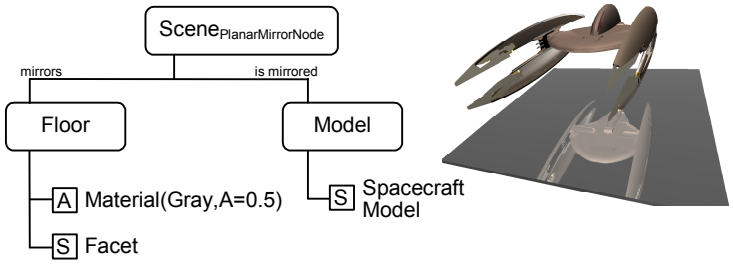

Abbildung 19: Deklaration einer planaren Reflexion durch einen hierarchische Szenenknoten.

- **Perzeptionshinweise durch Beleuchtung**. Eine dem nicht-photorealistischen Rendering entsprungene Idee ist, die Aufmerksamkeit des Betrachters auf die wesentlichen Teile einer Szene zu lenken, indem nur diese mit einem hohen Detailgrad dargestellt werden [82]. Zum besseren Verständnis von Abbildungen kann es analog vorteilhaft sein, nur die entscheidenden Bildelemente mit Schatten oder Reflexionen darzustellen.
- **Verhinderung von Artefakten**. Trotz technischer Fortschritte sind eine Reihe wichtiger Renderingverfahren wie zum Beispiel Shadow-Mapping anfällig für Artefakte (Abschnitt 3.4.2). Insbesondere Artefakte durch Selbstschattierung können auf einfache Weise verhindert werden, indem bei den Eigenschaften von Geometrie darauf geachtet wird, dass keine Geometrie einen Schatten auf sich selbst wirft.
- **Renderinggeschwindigkeit**. Die Darstellung eines Beleuchtungseffekts erster Ordnung ist teuer, und die Graphikhardware ist darum unter Umständen nicht leistungsfähig genug, den Effekt für die gesamte Szene darzustellen. Dagegen kann durch explizite Angabe betroffener und handelnder Geometrie unter Umständen ein Beleuchtungseffekt für ausgewählte, wichtige Teile der Szene mit ausreichender Geschwindigkeit dargestellt werden.

Die ersten beiden Punkte betreffen ausschließlich die Deklaration einer Szene, die anderen beiden Punkte betreffen zwar Probleme bei der Auswertung einer Szene, spielen aber zum Erzielen einer korrekten und flüssigen Darstellung eine wesentliche Rolle und sind darum auch für die Szenenmodellierung wichtig. Zur Anpassung aller genannten Punkte ist die getrennte Angabe von betroffener und handelnder Geometrie in einer Szenenbeschreibung zwingend erforderlich.

5.5.2 Hierarchische Szenenknoten

Ein Ansatz zur abstrakten Deklaration von Beleuchtung erster Ordnung ist, die Nutzung des Beleuchtungseffekts für den gesamten Szenengraphen, bzw. für einen Teilgraphen anzugeben. Dazu kann ein spezieller hierarchischer Szenenknoten anzeigen, dass seine Kinder beispielsweise einen Schatten werfen und mit Schatten gerendert werden. Offenbar ist so aber keine Trennung von betroffener und handelnder Geometrie, das heißt keine granulare Auswahl von Geometrie möglich. Ein weitergehender Ansatz ist deshalb, zur Deklaration eines Renderingeffekts hierarchische Szenenknoten mit zwei Teilgraphen zu verwenden, von denen der eine Teilgraph die handelnde und der andere die betroffene Geometrie enthält. Abbildung 19 zeigt ein entsprechendes Beispiel zur Deklaration einer planaren Reflexion. Die Firma OpenWorlds hat diesen Ansatz für Schatten als eine Erweiterung ihres VRML-Browsers spezifiziert und implementiert [17]. Ein ähnlicher Vorschlag stammt von Sudarsky [83]. In beiden Fällen ist die Geometrie im Schatten aufgrund des verwendeten Renderingverfahrens zur Auswertung eines solchen Knotens auf planare Flächen beschränkt.

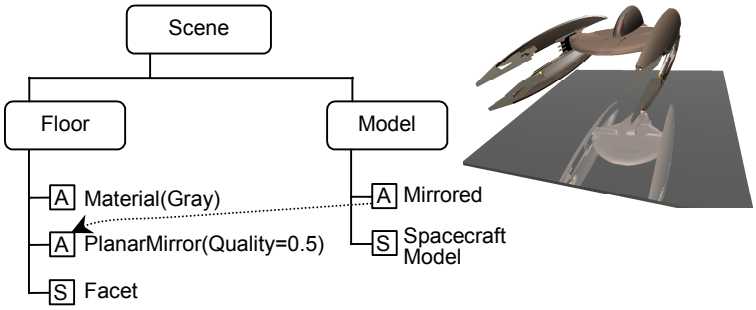

Abbildung 20: Deklaration einer planaren Reflexion durch abstrakte Attribute.

Bewertung in Bezug auf die Anforderungen

Angenommen, man lässt mit dem Ansatz der hierarchischen Szenenknoten beliebige betroffene und handelnde Geometrie zu. Dann ist eine redundanzfreie Deklaration von Beleuchtung erster Ordnung nicht möglich, denn Geometrie, die gleichzeitig betroffen und handelnd ist, muss in beiden Kindknoten eines hierarchischen Szenenknotens vorhanden sein. Dies hat außerdem zur Folge, dass nicht offensichtlich ist, wie mehrere Beleuchtungseffekte für dieselbe Geometrie zu deklarieren sind. Hierarchische Szenenknoten für Beleuchtung erster Ordnung könnten zwar geschachtelt werden, aber dabei wäre nicht klar, ob ein geschachtelter Beleuchtungsknoten in beiden Teilgraphen des übergeordneten Beleuchtungsknotens deklariert werden müsste.

Im Beispiel von Abbildung 19 müssten also zur Deklaration von zwei Spiegeln in einer Szene zwei entsprechende hierarchische Szenenknoten vorhanden sein. Soll aber das Raumschiff-Objekt in beiden Spiegeln gespiegelt werden, müsste es in beiden Teilgraphen für die gespiegelten Geometrie-Objekte deklariert werden. Demnach befänden sich zwei ungespiegelte Raumschiff-Objekte in der Szene, was nicht das angestrebte Ziel war.

Hierarchische Szenenknoten erlauben also eine abstrakte, granulare Deklaration von Beleuchtungseffekten, aber sie erfordern redundante Geometrie, und die kombinierte Deklaration mehrerer Beleuchtungseffekte ist mit ihnen nicht möglich.

5.5.3 Abstrakte Attribute

Ein anderer Ansatz zur Deklaration von Beleuchtung erster Ordnung sind abstrakte Attribute. Er vermeidet die Nachteile von hierarchischen Szenenknoten für diesen Zweck und wird darum von VRS verwendet. Abstrakte Attribute geben getrennt für einen Renderingeffekt handelnde und betroffene Geometrie an.

Planare Reflexionen

Eine spiegelnde ebene Fläche wird durch das Monoattribut `PlanarMirror` deklariert. Dieses Attribut wirkt sich nur auf `Facet`-Objekte aus, die in VRS allgemeine planare Geometrie enthalten. Für alle anderen Geometrie-Objekte wird es ignoriert. Die lokale Beleuchtung von spiegelnden Flächen bleibt durch das `PlanarMirror`-Attribut unverändert, aber es wird die Reflexion der gespiegelten Geometrie-Objekte hinzuaddiert, multipliziert mit einem Faktor des Attributs, der die Qualität, also die Helligkeit der Reflexion angibt. Außerdem enthält das `PlanarMirror`-Attribut die ambiente Farbe der Hintergrundreflexion an den Stellen des Spiegels, an denen keine gespiegelte Geometrie sichtbar ist.

Gespiegelte Geometrie wird durch das Attribut `Mirrored` deklariert. Ein `Mirrored`-Attribut enthält eine Referenz auf das Spiegel-Attribut, und Geometrie wird nur in einem planaren Spiegel reflektiert, wenn ein `Mirrored`-Attribut mit Referenz auf das entsprechende `PlanarMirror`-

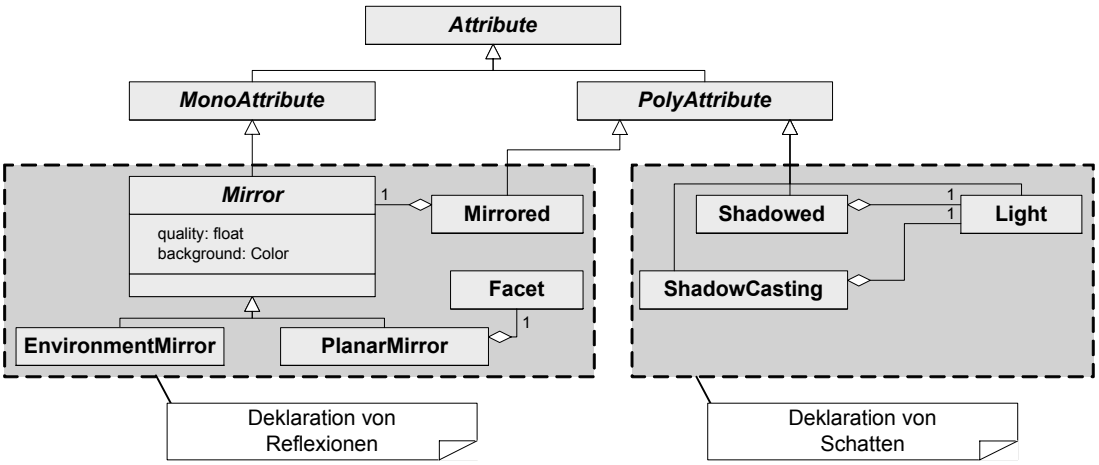

Abbildung 21: Klassendiagramm für Attribute, die zur Deklaration von Beleuchtung erster Ordnung dienen.

Attribut für die Geometrie aktiv ist. Abbildung 20 illustriert die Deklaration einer planaren Reflexion durch diese Attribute. Geometrie kann von mehreren Spiegeln reflektiert werden, daher handelt es sich bei der Klasse `Mirrored` um ein Polyattribut.

Reflexionen durch Environment-Mapping

Diese werden durch das Monoattribut `EnvironmentMirror` deklariert. Es hat die gleichen Eigenschaften wie das `PlanarMirror`-Attribut, ermöglicht aber das Spiegeln von beliebigen 3D-Objekten durch Environment-Mapping. Die `PlanarMirror`- und `EnvironmentMirror`-Klassen sind von der Basisklasse `Mirror` abgeleitet, die die Eigenschaften Qualität und Farbe der Hintergrundreflexion kapselt. Gespiegelte Geometrie wird, wie bei planaren Reflexionen, ebenfalls durch `Mirrored`-Attribute mit Referenz auf ein `Mirror`-Attribut deklariert.

Schatten

Zur Deklaration handelnder, das heißt schattenwerfender Geometrie dient das Attribut `ShadowCasting`, und die Deklaration betroffener, also im Schatten liegender Geometrie erfolgt durch das Attribut `Shadowed`. Diese beiden Attribute enthalten jeweils eine Referenz auf die Lichtquelle in der Szenenbeschreibung, auf die sie sich beziehen. Damit der Schatten einer Lichtquelle auf Geometrie dargestellt wird, müssen zwei Bedingungen erfüllt sein: 1. Für die Geometrie muss ein `Shadowed`-Attribut mit Referenz auf die Lichtquelle aktiv sein. 2. Zwischen der Geometrie mit dem `Shadowed`-Attribut und der Lichtquelle muss sich Geometrie mit einem `ShadowCasting`-Attribut und mit Referenz auf dieselbe Lichtquelle befinden. Nur solche Geometrie wirft einen Schatten. Selbstschattierung ist dabei möglich, das heißt schattenwerfende und im Schatten liegende Geometrie kann identisch sein. In diesem Fall sind sowohl ein `Shadowed`-Attribut als auch ein `ShadowCasting`-Attribut für die Geometrie aktiv.

`ShadowCasting`- und `Shadowed`-Attribute sind Polyattribute, es können also mehrere Attribute dieser Arten für eine Geometrie gleichzeitig aktiv sein. Dies wird zur Deklaration von Schatten verschiedener Lichtquellen in der Szene für dieselbe Geometrie genutzt. Geometrie mit mehreren aktiven `ShadowCasting`-Attributen wirft Schatten von verschiedenen Lichtquellen, die jeweils in den einzelnen Attributen referenziert werden. Auf Geometrie mit mehreren aktiven `Shadowed`-Attributen wird der Schatten von verschiedenen referenzierten Lichtquellen dargestellt.

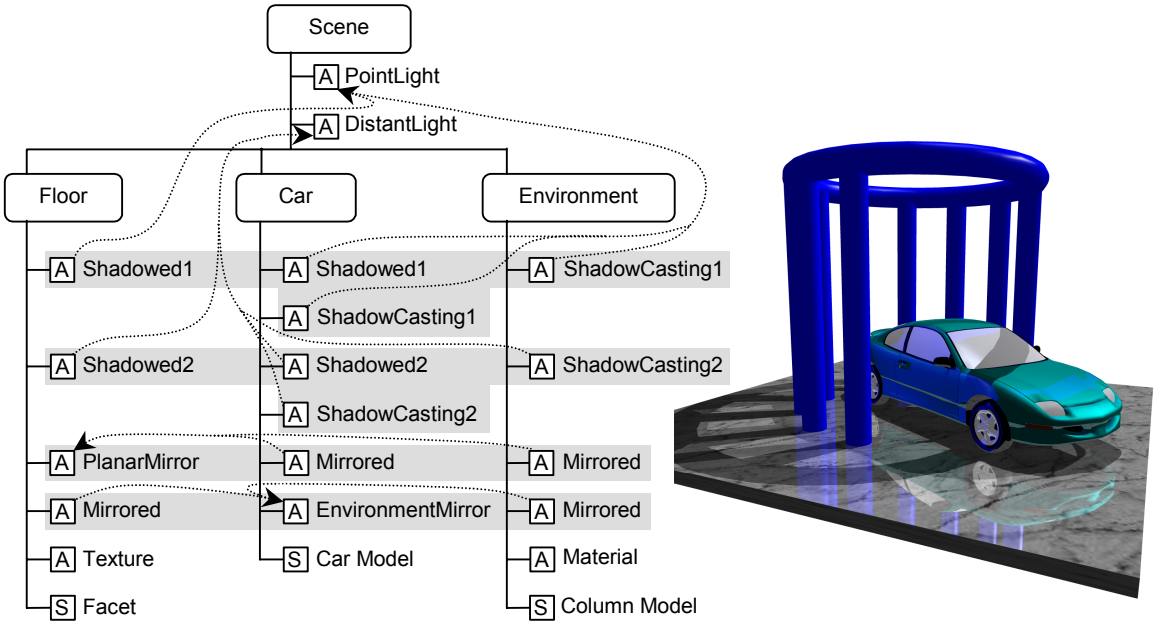

Abbildung 22: Deklaration von mehreren Beleuchtungseffekten durch abstrakte Attribute in einer Szene. Attribute, die zusammen einen Effekt beschreiben, sind dabei grau hinterlegt.

Bewertung in Bezug auf die Anforderungen

Einen Gesamtüberblick über die abstrakten Attribute zur Deklaration von Beleuchtung erster Ordnung gibt Abbildung 21. Insgesamt können durch diesen Ansatz Beleuchtungseffekte in einer Szenenbeschreibung auf konsistente und nachvollziehbare Weise deklariert werden. Dabei gibt es keine Redundanzen durch mehrfach vorhandene Geometrie in der Szenenbeschreibung, und es können auf eindeutige Weise verschiedene Beleuchtungseffekte für dieselbe Geometrie deklariert werden. Da die Attribute zudem getrennt handelnde und betroffene Geometrie deklarieren und somit eine granulare Auswahl von Geometrie ermöglichen, erfüllen abstrakte Attribute die Anforderungen für die Deklaration von Beleuchtungseffekten aus Abschnitt 5.1.

Abbildung 22 zeigt das Beispiel eines Szenengraphen, in dem zwei Schatten, eine planare Reflexion und eine Reflexion durch Environment-Mapping deklariert sind. Selbst bei einer solchen Anzahl von Effekten bleibt die Szenenbeschreibung noch überschaubar und verständlich.

5.6 Deklaration von globalen Renderingeffekten

Erweiterungen des Kameramodells betreffen die gesamte Szene. Das gleiche gilt für Nachbearbeitungseffekte. Damit sind diese Effekte recht einfach zu deklarieren, denn einzelne Geometrie-Objekte spielen für die Art der Deklaration dieser globalen Renderingeffekte keine Rolle. Daher gibt es mehrere vergleichbare Möglichkeiten zur Deklaration eines globalen Renderingeffekts in einer Szenenbeschreibung:

- Er kann als optionale Eigenschaft des (einzigen) Kamera-Attributs in der Szene spezifiziert werden,
- die Deklaration erfolgt durch ein Monoattribut im Wurzelknoten des Szenengraphen,
- oder die Deklaration erfolgt durch eine Konfigurationsoption des gesamten Szenengraphen, bzw. durch die Auswahl eines Traversierungsalgorithmus für die Szenengraphauswertung.

Hierbei sprechen wir von einem *Szenenkonfigurations-Attribut*, auch wenn es sich technisch nicht um ein gewöhnliches Attribut in einer Szenenbeschreibung handelt.

Mehrere globale Renderingeffekte sind im Allgemeinen miteinander kombinierbar. Da die Reihenfolge, in der sie angewendet werden, wichtig für das Renderingergebnis ist, erfolgt die Deklaration der Effekte am Besten als geordnete Liste aufeinanderfolgender Szenenkonfigurations-Attribute des Szenengraphen. Dies erlaubt eine explizite Steuerung, in welcher Reihenfolge die globalen Effekte angewendet werden, die durch die anderen Deklarationsweisen nicht in gleicher Klarheit möglich ist.

In VRS ist es durch Austausch eines globalen Objekts zur Szenengraphtraversierung möglich, Tiefenunschärfe oder Stereo-Rendering als eine Eigenschaft der virtuellen Zeichenfläche, die den Wurzelknoten des Szenengraphen enthält, zu deklarieren. Mehrere globale Renderingeffekte gleichzeitig können derzeit noch nicht spezifiziert werden.

Neben VRS bieten auch einige andere Szenengraphsysteme die Möglichkeit der Deklaration von erweiterten Kameramodellen an: in NVSG gibt es beispielsweise eine Stereo-Kamera und eine Kamera für Software-Antialiasing als optionale Eigenschaft des Kamera-Attributs in der Szene.

Kapitel 6

SZENENGRAPHBASIERTE AUSWERTUNG ECHTZEITFÄHIGER RENDERINGEFFEKTE

Dieses Kapitel befasst sich mit der Auswertung, das heißt dem Rendering von Renderingeffekten, die in einer Szenenbeschreibung spezifiziert sind. Dabei wird vorausgesetzt, dass die Spezifikation der Renderingeffekte durch abstrakte Attribute erfolgt. Während die Auswertung einer Szenenbeschreibung mit ausschließlich konkreten oder aggregierenden Attributen durch eine einmalige Traversierung des Szenengraphen erfolgen kann, muss zur Auswertung abstrakter Attribute der Szenengraph mehrfach traversiert werden können, weil sie in der Regel Multipass-Rendering erfordert. Dazu wird in diesem Kapitel das Konzept der *Renderingtechnik* entwickelt. Eine Renderingtechnik implementiert ein Multipass-Verfahren und kapselt es als eine Klasse. Die Implementierung wertet dabei in der Beschreibung einer Szene vorhandene abstrakte Attribute aus.

Ein Spezialfall sind Szenenkonfigurations-Attribute. Weil diese nicht Bezug auf einzelne Geometrie-Objekte in der Szene nehmen, können sie den Algorithmus zu ihrer Auswertung selbst implementieren.

Zur Auswertung sonstiger abstrakter Attribute werden zum einen global wirkende Attribute betrachtet, die Bezug auf einzelne Geometrie-Objekte in der Szene nehmen, deren Auswirkung aber auch von anderer Geometrie in der Szene abhängt. Zur Auswertung solcher Attribute muss die gesamte Szenenbeschreibung mehrfach traversiert werden, um Informationen über die Geometrie zwischenspeichern zu können. Bei jeder Traversierung sind unterschiedliche Einstellungen der Renderingpipeline pro Geometrie nötig. Zusätzliche Schwierigkeiten liegen darin, dass mehrere solche Attribute für unterschiedliche Renderingeffekte gleichzeitig für dieselbe Geo-

metrie aktiv sein können, die entsprechenden Renderingverfahren also kombiniert werden müssen.

Abstrakte Attribute, die nur lokale Eigenschaften von Geometrie verändern, können durch mehrfache Traversierung des Teilgraphen hinter dem Attribut ausgewertet werden. Attribute zur Deklaration von Markierungseffekten werden davon abweichend mit dem Mechanismus für global wirkende Attribute ausgewertet, obwohl sie nur lokale Eigenschaften der Geometrie beeinflussen. Der Vorteil ist, dass damit nur Monoattribute zur Einstellung der Oberflächenschattierung übrigbleiben, von denen immer nur eines zur gleichen Zeit aktiv ist. Von dieser Tatsache wird beim Algorithmus zur Auswertung dieser Attribute profitiert.

6.1 Szenengraphauswertung ohne abstrakte Attribute

Szenengraphsysteme implementieren den Algorithmus zum Rendern einer hierarchischen Szenenbeschreibung ohne abstrakte Attribute durch eine einmalige Traversierung des Szenengraphen. Der Algorithmus wird als *Depth-First* Traversierung bezeichnet, das heißt, sowohl beim Erreichen als auch beim Verlassen eines Szenenknotens wird eine Aktion durchgeführt, dazwischen werden rekursiv die nachfolgenden Szenenknoten bzw. Teilgraphen traversiert. Die Behandlung der einzelnen Szenenknoten ist dabei wie folgt [18]:

Geometrie-Objekte werden beim Erreichen gerendert. Beim Verlassen werden sie ignoriert, weil sie den Renderingkontext in der jeweiligen Render-Methode nicht verändern.

Transformationen werden beim Erreichen mit der vorherigen Transformation verknüpft, beim Verlassen wird die vorherige Transformation wiederhergestellt. Nachfolgende Geometrie wird also mit der veränderten Transformation gerendert. Die aktuelle Transformation ist zu jeder Zeit der Traversierung vom Szenengraphsystem abfragbar.

Konkrete Attribut-Objekte werden beim Erreichen aktiviert, das heißt, im Renderingsystem wird die durch das Objekt bezeichnete Einstellung des Renderingkontextes vorgenommen. Beim Verlassen des Objekts wird die vorhergehende Einstellung wieder aktiviert. Konzeptionell wird hier zwischen Mono- und Polyattributen unterschieden: Ein Polyattribut-Objekt wird beim Erreichen zusätzlich zu den bisherigen Polyattributen aktiviert, ein Monoattribut-Objekt überschreibt beim Erreichen das bisher aktive Attribut des gleichen Typs. Während der Traversierung kann vom Szenengraphsystem jederzeit die Menge der zur Zeit aktiven Attribut-Objekte erfragt werden.

Aggregierende Attribut-Objekte werden beim Erreichen und Verlassen wie mehrere aufeinanderfolgende konkrete Attribut-Objekte behandelt.

Hierarchische Knoten werden beim Erreichen ausgewertet, das heißt, die Kinder des Knotens werden nacheinander traversiert. Beim Verlassen werden hierarchische Knoten ignoriert.

Diese Behandlung der Szenenknoten stellt für das Rendering effektiv sicher, dass Attribute und Transformationen in einem Teilgraph sich in anderen Teilgraphen nicht auswirken. Im Gegensatz dazu verwenden frühe Szenengraphsysteme wie Open Inventor zum Teil eine einfache Prä-Order Traversierung für das Rendering [81]. Damit werden Attribute und geometrische Transformationen in einem Teilgraph nicht mehr zurückgenommen und wirken sich somit auf andere Teilgraphen aus, erzeugen also möglicherweise unerwünschte Seiteneffekte. Die Depth-First Traversierung hat sich deswegen in allen modernen Szenengraphsystemen durchgesetzt.

Das Szenengraphsystem verwaltet während der Traversierung den *Evaluationskontext*, einen Speicher, der den Zustand aller aktiven Attribute sowie geometrischer Transformationen bereithält. So kann beim Erreichen eines Geometrie-Objekts abgefragt werden, welche Attribute zur

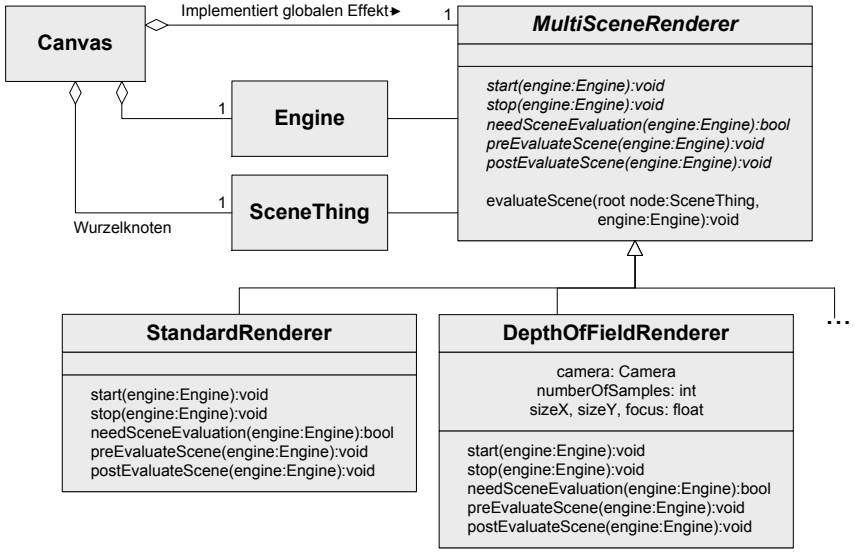

Abbildung 23: Abstraktes Interface der `MultiSceneRenderer`-Klasse, Implementierung des Interfaces in abgeleiteten Klassen.

Zeit aktiv sind, und darauf gegebenenfalls reagiert werden. Als Datenstruktur ist dabei für jeden Typ von Monoattribut ein Stack nötig, auf dem entsprechende Attribut-Objekte beim Erreichen bzw. Verlassen gelegt und wieder entfernt werden. Für Polyattribute wird entsprechend ein Set verwendet, in das Objekte eingefügt bzw. wieder entfernt werden. In VRS wird der Evaluationskontext von einer Instanz der Klasse `Engine` bereitgestellt.

6.2 Globale Multipass-Verfahren

Globale Multipass-Verfahren, die nicht auf einzelne Geometrie-Objekte in der Szene Bezug nehmen, werden durch eine mehrfache vollständige Szenengraphauswertung implementiert. Dazu dient das Interface der abstrakten Klasse `MultiSceneRenderer`, die den Wurzelknoten eines Szenengraphen enthält, diesen mehrfach zum Rendern auswertet, und vor und nach jeder Auswertung Akkumulations-, Nachbearbeitungs- oder sonstige Operationen durchführen kann. Eine Instanz einer `MultiSceneRenderer`-Klasse ist gleichzeitig ein Szenenkonfigurations-Attribut. Eine Trennung zwischen Attribut zur Deklaration eines globalen Renderingeffekts und separater Klasse zur Implementierung der Auswertung des Effekts hätte hier keine Vorteile.

Von der `MultiSceneRenderer`-Klasse abgeleitet sind zum Beispiel die Klassen `StandardRenderer` zum gewöhnlichen Auswerten der Szenenbeschreibung, `StereoRenderer` zur Unterstützung von Stereo-Rendering, oder `DepthOfFieldRenderer` zur Darstellung von Tiefenunschärfe (Abschnitt 3.5). Andere Anwendungen sind Verfahren für Nachbearbeitungseffekte (Abschnitt 3.6), wie zum Beispiel Graustufenfilter durch ein Objekt der Klasse `GrayScaleRenderer`.

In Abbildung 23 wird das Interface der Klasse `MultiSceneRenderer` und von einigen abgeleiteten konkreten Klassen dargestellt. Die Szenengraphauswertung wird durch die Methode `evaluateScene()` vom `MultiSceneRenderer` durchgeführt, diese ruft dabei die abstrakten, von den Kindklassen implementierten Methoden auf. In der `StandardRenderer`-Klasse sind diese Methoden leer implementiert. Die `DepthOfFieldRenderer`-Klasse implementiert in ihnen den Tiefenunschärfealgorithmus: Die Methode `preEvaluateScene()` setzt die Kamera in der Szene auf eine verschobene Position, `postEvaluateScene()` addiert das im letzten Durchlauf gerenderte Bild in den Akkumulationspuffer, und `stop()` kopiert das fertige Bild vom Akkumulationspuffer in den Framebuffer zurück. Die Szene wird genau `numberOfSamples` mal traversiert.

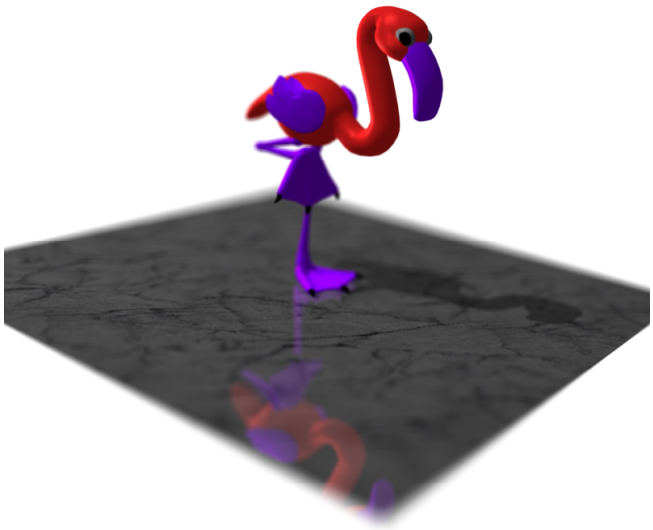

Abbildung 24: Tiefenunschärfe zusammen mit Schatten und Reflexion.

Zur Kombination von Szenenkonfigurations-Attributen mit anderen Renderingeffekten zur Deklaration globaler Multipass-Verfahren mit Bezug auf einzelne Geometrie-Objekte oder lokaler Multipass-Verfahren sind keine weiteren Vorkehrungen zu treffen, sobald in der Szenengraphauswertung solche Renderingeffekte ausgewertet werden können und dabei keine Ressourcenkonflikte auftreten. Abbildung 24 zeigt die Anwendung von Tiefenunschärfe zusammen mit einem Schatten und einer planaren Reflexion. Die Bilderzeugungszeit für dieses Bild liegt aufgrund des aufwändigen Tiefenunschärfealgorithmus etwa bei einer Sekunde.

Eine Kombination verschiedener globaler Renderingeffekte durch Schachtelung der entsprechenden globalen Multipass-Verfahren wäre grundsätzlich einfach zu realisieren, zumindest wenn die Verfahren unterschiedliche Ressourcen der Graphikhardware verwenden. Im Wesentlichen wäre hierzu die Möglichkeit einer rekursiven Schachtelung verschiedener `MultiSceneRenderer`-Objekte erforderlich.

6.3 Globale Multipass-Verfahren mit Bezug auf einzelne Geometrie

Die Integration von globalen Multipass-Verfahren mit Bezug auf einzelne Geometrie-Objekte in eine Szenengraphauswertung ist technisch vergleichsweise kompliziert. Jedoch kann sie evolutionär in bestehende Szenengraphsysteme integriert werden. Darüber hinaus sind Anpassungen an bekannten Algorithmen zur Optimierung der Szenengraphauswertung und neue Optimierungen nötig bzw. sinnvoll.

6.3.1 Überblick über die softwaretechnische Umsetzung

Die folgenden Erweiterungen sind für eine szenengraphbasierte Auswertung von globalen Multipass-Verfahren mit Bezug auf einzelne Geometrie-Objekte erforderlich:

- **Mehrfache Szenendurchläufe**. Als Grundlage kann hierfür eine ähnliche Softwarekomponente wie für ein globales Multipass-Verfahren im Abschnitt 6.2 verwendet werden. Dies ist bei der Umsetzung mit VRS die Klasse `TechniqueProcessor`. Aufgabe der Komponente ist die Auswahl der Renderingverfahren, die zur Synthese der Renderingeffekte in der Szenenbeschreibung nötig sind, und deren Ansteuerung. Jedes Renderingverfahren definiert einen oder mehrere Renderingdurchläufe, das heißt, Szenengraphtraversierungen, die von dem `TechniqueProcessor` in einer Prioritätswarteschlange angeordnet und der Reihe nach

durchgeführt werden. Die Ordnung der Durchläufe ist durch eine konstante numerische ID für jeden Durchlauf festgelegt.

- **Abstrakte Attribute** werden wie konkrete Attribute bei Erreichen in den Evaluationskontext aufgenommen und bei Verlassen wieder entfernt. Sie sind aber direkt nicht auf einzelne Einstellungen des Renderingsystems abbildbar, daher ändern sie direkt den Zustand der Renderingpipeline nicht.
- **Selektive Geometrie-Filterung**. In jedem Renderingdurchlauf können anhand verschiedener Kriterien Geometrie-Objekte gefiltert, das heißt ihr Rendering verhindert werden. Im Allgemeinen hängt die Filterung von dem Zustand des Evaluationskontextes, vor allem der abstrakten Attribute, zum Zeitpunkt der Traversierung des Geometrie-Objektes ab. Beispielsweise wird in einem Renderingdurchlauf zur Erzeugung einer Shadow-Map für eine Lichtquelle nur solche Geometrie gerendert, für die ein entsprechendes `ShadowCasting`-Attribut aktiv ist.
- **Anwendung zusätzlicher Attribute**. Für jeden Renderingdurchlauf können verschiedene konkrete Attribute angewendet werden, die nicht in der eigentlichen Szenenbeschreibung enthalten sind. Diese Attribute können für den gesamten Durchlauf gültig sein, dann werden sie zu Beginn des Durchlaufs aktiviert und danach wieder entfernt. Falls nötig können zusätzliche Attribute auch für einzelne Geometrie-Objekte aktiviert und wieder deaktiviert werden.
- **Selektive Filterung von Attributen**. In jedem Renderingdurchlauf können bestimmte Typen konkreter Attribute, die sich in der Szenenbeschreibung befinden, gefiltert und damit ignoriert werden. Beispielsweise kann zu Beginn des Durchlaufs eines Renderingverfahrens für Shadow-Mapping eine Shadow-Map-Textur aktiviert werden. Diese Shadow-Map ist ein internes Detail des Renderingverfahrens und als solches nicht in der Szenenbeschreibung enthalten. Daher muss sichergestellt werden, dass sie während des Durchlaufs nicht von einer anderen Textur in der Szenenbeschreibung überschrieben wird.
- **Separate Traversierungen für Lichtquellen**. Die Behandlung von Lichtquellen, die mit einem Schatten gerendert werden, wird vollständig umgestellt. Solche Lichtquellen werden im *Haupt-Renderingdurchlauf* deaktiviert, damit in separaten Renderingdurchläufen ihr Lichtanteil, von der Schattenberechnung beeinflusst, zu der Szene hinzuaddiert werden kann.
- **Vor-Inspektion des Szenengraphen.** Renderingverfahren benötigen während der Szenengraphtraversierung bzw. vor dem Start eines Renderingdurchlaufs vielfach globale Informationen über den Szenengraphen, die zu diesen Zeitpunkten nicht über den Evaluationskontext verfügbar sind. Schattenverfahren müssen beispielsweise die geometrische Transformation einer Lichtquelle, die von einem Schatten modifiziert wird, zu Beginn eines Renderingdurchlaufs ermitteln können. Verfahren für planare Reflexionen benötigen analog die Position und Ausrichtung der spiegelnden Geometrie. Solche Informationen werden in einer Inspektion des Szenengraphen, das heißt in einem Analysedurchlauf vor allen anderen Renderingdurchläufen, generiert und als globale Informationen über den Szenengraphen abgespeichert. Damit stehen sie in allen folgenden Durchläufen zur Verfügung. Nach vollendeter Synthese des Bilds werden diese Informationen in einem letzten Durchlauf wieder entfernt.

Eine aus softwaretechnischer Sicht wesentliche Entscheidung ist, jedes Renderingverfahren in einer einzelnen Klasse kapseln zu können. Insbesondere implementiert also der `TechniqueProcessor` Renderingverfahren nicht selbst. Dies wird durch *globale Renderingtechniken* erreicht, die in VRS durch die Basisklasse `Technique` realisiert sind und jeweils die Implementierung eines Renderingverfahrens kapseln. Die Auswahl eines Renderingverfahrens beschränkt sich dann auf die Suche nach einem passenden `Technique`-Objekt für einen Renderingeffekt.

Diese Vorgehensweise hat den Vorteil, dass Renderingtechniken getrennt voneinander implementiert und damit technisch überschaubar sind. Auch verschiedene Renderingtechniken für den gleichen Renderingeffekt, wie zum Beispiel stencilbasierte und texturbasierte planare Reflexionen, können so separat entwickelt und verwendet werden. Dies ist auch in Hinblick auf unterschiedliche Graphikhardware mit verschiedenen Fähigkeiten nützlich.

6.3.2 Implementierung

Die `Engine` als zentrale Instanz zur Verwaltung des Evaluationskontextes ist mit einem Objekt der Klasse `TechniqueProcessor` assoziiert, der die Logik zum Auswerten aller `Technique`-Objekte bereitstellt (Abbildung 25). In der Methode `MultiSceneRenderer::evaluateScene()` (Abschnitt 6.2) wird die Vorschrift zum mehrfachen Durchlauf der Szene durch den `TechniqueProcessor` implementiert (Abbildung 26).

Verwaltung von globalen Renderingtechniken und ihrer Durchläufe

Die Durchläufe, die der `TechniqueProcessor` verwaltet und durchführt, werden von den `Technique`-Objekten, die bei die Szenengraphauswertung berücksichtigt werden, definiert. Diese globalen Renderingtechniken befinden sich als eigenständige Objekte im Szenengraph und werden bei der Vor-Inspektion des Szenengraphen gefunden. Zusätzlich gibt es die Vor- und die Nach-Inspektionstechnik sowie die Haupttechnik, die implizit im `TechniqueProcessor` vorhanden sind. Speziell die Vor-Inspektionstechnik ist hier zu nennen, weil sie die Vor-Inspektion des Szenengraphen implementiert und damit insbesondere die Suche nach den weiteren Techniken im Szenengraph durchführt. Die Haupttechnik implementiert den Haupt-Rendering-

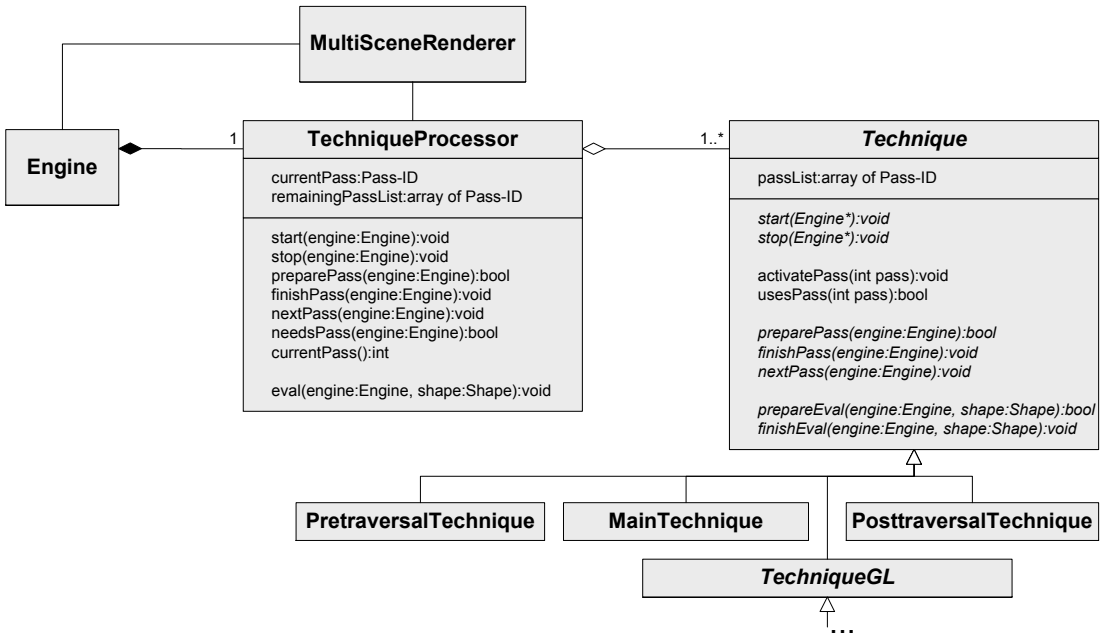

Abbildung 25: Klassendiagramm von `TechniqueProcessor` und `Technique`.

```
void MultiSceneRenderer::evaluateScene(SceneThing thing, Engine engine) {

    // Loop to handle the global passes of the MultiSceneRenderer
    {
        // Loop body
        TechniqueProcessor processor ← engine.getTechniqueProcessor();
        processor.start(engine);
        while (processor.needsPass(engine)) {
            if(processor.preparePass(engine)) {
                thing.evaluate(engine); // traverses the scene graph
            }
            processor.finishPass(engine);
            processor.nextPass(engine);
        }
        processor.stop(engine);
    }
    // End of loop to handle the global passes of the MultiSceneRenderer
}
```

Abbildung 26: Nutzung vom `TechniqueProcessor` zur mehrfachen Traversierung der Szene.

durchlauf zum einmaligen Rendern der Szene ohne spezielles Renderingverfahren, und die Nach-Inspektionstechnik zerstört temporär angelegte Daten aus dem Vor-Inspektionsdurchlauf.

Alle Durchläufe werden vom `TechniqueProcessor` in einer Prioritätswarteschlange verwaltet und der Reihe nach abgearbeitet. In der Regel ist für jeden Durchlauf ein `Technique`-Objekt verantwortlich, das in diesem Zeitraum als *aktiv* bezeichnet wird. Zu Beginn eines Durchlaufs wird die `activatePass()`-Methode der aktiven Technik aufgerufen, wo ihr die ID des Durchlaufs mitgeteilt wird. Die Methoden `preparePass()`, `finishPass()`, und `nextPass()` leitet der `TechniqueProcessor` jeweils zur aktiven Technik weiter. Diese kann dort Einstellungen des Rendering- oder Evaluationskontextes für den Durchlauf setzen und wieder zurücknehmen. Der Rückgabewert von `preparePass()` steuert, ob ein Renderingdurchlauf durchgeführt wird oder nicht. Zum Ende eines Durchlaufs können in `nextPass()` zusätzliche Durchläufe beim `TechniqueProcessor` angefordert und damit in die Prioritätswarteschlange eingefügt werden.

Auswertung einzelner Geometrie-Objekte

Wird während der Szenengraphtraversierung ein Geometrie-Objekt angetroffen, wird die Methode `TechniqueProcessor::eval()` aufgerufen. Diese nutzt die Methode `prepareEval()` der aktiven Technik, in der anhand der Einstellungen des Evaluationskontextes überprüft wird, ob das Geometrie-Objekt gefiltert werden soll. Falls ja, ist der Rückgabewert der Methode `false`. Ansonsten können weitere Einstellungen des Rendering- oder Evaluationskontextes für das Rendern der Geometrie vorgenommen werden, bevor `true` zurückgegeben wird. Der `TechniqueProcessor` kümmert sich dann gegebenenfalls um den Aufruf der Render-Methoden für das Geometrie-Objekt, bevor er die Methode `finishEval()` der aktiven Technik aufruft, in der die in `prepareEval()` veränderten Einstellungen wieder zurückgenommen werden (Abbildung 30).

Kombinierte Renderingdurchläufe

In Spezialfällen können für einen Durchlauf auch mehrere Techniken gleichzeitig aktiv sein. Es werden dann die entsprechenden Methoden aller aktiven Techniken nacheinander aufgerufen. Die eventuellen Rückgabewerte der Methoden werden wie folgt kombiniert: 1) Es genügt die Rückgabe `true` einer `preparePass()`-Methode der aktiven Techniken, damit die Szene in dem Durchlauf traversiert wird, das heißt die Rückgabewerte werden durch logisches „Oder“ verknüpft. 2) Es genügt eine Rückgabe `false` der `prepareEval()`-Methoden der aktiven Techniken, damit ein Geometrie-Objekt gefiltert wird, das heißt diese Rückgabewerte werden durch logisches „Und“ verknüpft. Mehrere aktive Techniken gleichzeitig werden vor allem bei der Kombination von Reflexionstechniken mit Renderingtechniken für Markierungseffekte eingesetzt (Abschnitt 6.3.4).

6.3.3 Beispielhafte Implementierung einer stencilbasierten Reflexionstechnik

Abbildung 27 zeigt den Pseudocode einer vereinfachten Technik zur Darstellung planarer Reflexionen. Die Technik setzt voraus, dass die Reflexionen in der Szenenbeschreibung durch die abstrakten Attribute `PlanarMirror` und `Mirrored` deklariert sind (Abschnitt 5.5.3). Das eigentliche Renderingverfahren ist identisch zu dem im Abschnitt 4.4 beschriebenen, kann allerdings auch mit mehreren Reflexionen in einer Szene umgehen. Aus Platzgründen ist die Prüfung, ob es sich bei spiegelnder Geometrie um planare `Facet`-Objekte handelt, nicht dargestellt.

Die Technik hat drei verschiedene Durchläufe. Der erste, `MAIN`, dient zum normalen Rendern der Szene, und wird in jedem Fall einmal durchgeführt, also auch, wenn sich keine `PlanarMirror`-Attribute in der Szene befinden. Daher wird dieser Durchlauf auch bereits in der Methode `start()` der Technik im `TechniqueProcessor` registriert. Während der Szenengraphtraversierung in `MAIN` erfüllt die Technik in der Methode `prepareEval()` zwei Aufgaben: Die `PlanarMirror`-Attribute in der Szene werden separat in einer Liste gespeichert, und die Geometrie-Objekte, für die ein solches Attribut aktiv ist, werden mit einem eigenen Stencilwert in den Stencilpuffer gerendert. Zum Ende des Durchlaufs sind also die spiegelnden Gebiete im Stencilpuffer markiert. In diesen Gebieten werden dann in `finishPass()` Farb- und Tiefenwerte wieder zurückgesetzt.

Die anderen Durchläufe werden nur ausgeführt, wenn die Liste der `PlanarMirror`-Attribute nicht leer ist, das heißt spiegelnde Geometrie in der Szene vorhanden ist. Der Durchlauf `MIRRORED_SCENE` wird dabei für jedes solche Attribut einmal durchgeführt, der Durchlauf `MIRROR_BLEND` hingegen nur einmal zum Schluss. Dafür sorgt die Implementierung der Methode `nextPass()`, wo die Durchläufe entsprechend im `TechniqueProcessor` registriert werden.

Der Durchlauf `MIRRORED_SCENE` rendert die in spiegelnder Geometrie sichtbaren Objekte. Dazu wird in dem Durchlauf durch `preparePass()` die Kameraposition gespiegelt und es werden nur Fragmente geschrieben, wenn das entsprechende Bit im Stencilpuffer gesetzt ist. `prepareEval()` sorgt dafür, dass nur Geometrie-Objekte gerendert werden, für die ein `Mirrored`-Attribut aktiv ist, welches das in dem Durchlauf bearbeitete `PlanarMirror`-Attribut referenziert.

Der Durchlauf `MIRROR_BLEND` rendert die spiegelnden Geometrie-Objekte nochmals, um die Helligkeit der gespiegelten Objekte und der Spiegeloberfläche richtig darzustellen. Dazu wird die Helligkeit eines aktiven `PlanarMirror`-Attributs als Alphawert benutzt, mit dem durch Blending die bereits im Framebuffer befindliche, gespiegelte Szene multipliziert wird.

6.3.4 Globale Renderingtechniken in VRS

VRS enthält die folgenden Renderingtechniken zum Rendern von Schatten, die durch abstrakte Schattenattribute (Abschnitt 5.5.3) deklariert sind:

- Die `DepthTextureShadowTechniqueGL`, die Shadow-Mapping durch Tiefentexturen implementiert [74];
- Die `AlphaTextureShadowTechniqueGL`, die Shadow-Mapping durch Alphatexturen implementiert [39], zur Verwendung des Verfahrens mit verminderter Qualität auf älterer Hardware;
- Die `ShadowVolumeTechniqueGL`, die Schattenvolumen nach [23] implementiert;

```
class PlanarMirrorTechniqueGL : public TechniqueGL {
    MAIN, MIRRORED_SCENE, MIRROR_BLEND: Pass-ID; // with  MAIN < MIRRORED_SCENE < MIRROR_BLEND
    MirrorList: Array of Mirror-Attributes;
    MirrorIndex: Index into MirrorList;

public:
    bool PlanarMirrorTechniqueGL::start(Engine e) {
        e.getTechniqueProcessor().addPass(MAIN);
    }

    bool PlanarMirrorTechniqueGL::preparePass(Engine e) {
        if (pass == MAIN) {
            clearStencilBuffer();
            MirrorList ← ∅;
        } else if (pass == MIRRORED_SCENE) {
            stencilBit ← getBit(MirrorList[MirrorIndex]);
            e.push(StencilOp(DRAW_IF_SET, stencilBit));
            e.push(mirroredCamera);
        } else if (pass == MIRROR_BLEND) {
            e.push(DepthOp(DRAW_ALWAYS));
        }
        return true;
    }

    void PlanarMirrorTechniqueGL::finishPass(Engine e) {
        if (pass == MAIN) {
            resetDepthAndColorBufferWhereStencilIsNotZero();
        } else if (pass == MIRRORED_SCENE || pass == MIRROR_BLEND) {
            // pop all attributes that have been pushed in preparePass() before
        }
    }

    void PlanarMirrorTechniqueGL::nextPass(Engine e) {
        if (pass == MAIN || pass == MIRRORED_SCENE) {
            if (pass == MAIN) {
                MirrorIndex = 0;
                if (!MirrorList.empty())
                    e.getTechniqueProcessor().addPass(MIRROR_BLEND);
            } else if (pass == MIRRORED_SCENE) {
                MirrorIndex++;
            }
            if (MirrorIndex < MirrorList.size())
                e.getTechniqueProcessor().addPass(MIRRORED_SCENE);
        }
    }

    bool PlanarMirrorTechniqueGL::prepareEval(Engine e, Shape s){
        PlanarMirror mirror ← e.getAttribute(PlanarMirror);
        if (pass == MAIN)
            if (mirror) {
                if (mirror ∉ MirrorList) MirrorList.append(mirror);
                stencilBit ← getBit(mirror);
                e.push(StencilOp(SET_ALWAYS, stencilBit));
            } else {
                e.push(StencilOp(SET_ALWAYS, 0));
            }
            return true;
        } else if (pass == MIRRORED_SCENE) {
            MirroredIterator iter ← e.getAttributeIterator(Mirrored, on);
            for each(MirroredAttribute mirrored in iter)
                if (mirrored.references(MirrorList[MirrorIndex]))
                    return true;
        } else if (pass == MIRROR_BLEND) {
            if (mirror) {
                stencilBit ← getBit(mirror);
                e.push(StencilOp(DRAW_IF_SET, stencilBit));
                e.push(AlphaValue(mirror.getBrightness()));
                e.push(BlendFunc(ONE, SRC_ALPHA));
                return true;
            }
        }
        return false;
    }

    bool PlanarMirrorTechniqueGL::finishEval(Engine e, Shape s){
        // pop all attributes that have been pushed in prepareEval() before
    }
};
```

Abbildung 27: Pseudocode einer stencilbasierten, planaren Reflexionstechnik.

- Die `TwoSidedShadowVolumeTechniqueGL`, die unter Verwendung der OpenGL-Erweiterung `GL_EXT_stencil_two_side` die Beleuchtungsberechnung durch die Schattenvolumenpolygone in einem statt in zwei Renderingdurchläufen durchführt;
- Die `ShadowTechniqueGL`, die abhängig von den Fähigkeiten der Graphikhardware und von Hinweisen des Entwicklers eine der obigen Schattentechniken zum Rendering von Schatten auswählt.

Weitere Schattenverfahren, wie zum Beispiel der Algorithmus für Single Sample Soft Shadows (Abschnitt 3.4.3) zur Darstellung weicher Schatten, sind ebenfalls prototypisch als globale Renderingtechniken mit VRS entwickelt worden.

Zum Rendern von Reflexionen dienen die folgenden Techniken:

- Die `StenciledMirrorTechniqueGL`, für planare, stencilbasierte Reflexionen, deren Implementierung dem Pseudocode aus Abbildung 27 ähnelt, die darüber hinaus aber die Funktionalität enthält, mit Markierungstechniken zusammenzuarbeiten;
- Die `TexturedMirrorTechniqueGL`, für planare, texturbasierte Reflexionen;
- Die `EnvironmentMirrorTechniqueGL`, für dynamisches Environment-Mapping, die es zudem erlaubt, die Reflexionsrichtung durch eine Bumpmap zu modifizieren.

Zur Darstellung von Transparenz enthält VRS die `TransparencyTechniqueGL`, die transparente Geometrie durch zweifache Traversierung des Szenengraphen zur Implementierung des Verfahrens aus Abschnitt 3.3.1 rendert. Die `AlternativeTechniqueGL` dient zur Transparenzdarstellung von Alternativen wie im Abschnitt 3.3.3 beschrieben.

Weitere globale Renderingtechniken in VRS dienen zur Darstellung von Markierungseffekten, zum Beispiel die `TextureEdgesTechniqueGL` zur Hervorhebung von Geometrie mit einem `TextureEdges`-Attribut durch texturverstärkte Kanten, die `HaloTechniqueGL` zur Darstellung von Halos, oder die `SilhouetteTechniqueGL` zur bildbasierten Umrandung von Geometrie, die mit einem `Silhouette`-Attribut markiert ist. Markierungseffekte könnten auch durch lokale Renderingtechniken (Abschnitt 6.4) ausgewertet werden, da sie eigentlich nicht abhängig von anderer Szenengeometrie sind. Die Kombination eines Markierungseffekts mit Oberflächenschattierungen wäre aber so nur schwierig realisierbar.

6.3.5 Anordnung der Renderingdurchläufe

Eine grundsätzliche Schwierigkeit ist, die einzelnen Renderingdurchläufe von verschiedenen globalen Renderingtechniken so anzuordnen, dass sie zusammen das erwartete Ergebnis liefern. Eine solche, allgemeine Anordnung lässt sich für beliebige Renderingtechniken nicht finden, geschweige denn automatisch erzeugen. Der Grund hierfür ist, dass verschiedene Techniken Renderingeffekte implementieren können, deren Kombination semantisch nicht definiert ist, oder die aus technischen Gründen grundsätzlich nicht zusammenarbeiten (Abschnitt 3.8). Die Anordnung der Renderingdurchläufe verschiedener Techniken kann darum nur für bestimmte Kombinationen von Techniken korrekt sichergestellt werden.

Durchläufe der Techniken zur Beleuchtung erster Ordnung

In VRS sind alle Schatten- und Reflexionstechniken aufeinander abgestimmt und liefern kombiniert die erwarteten Ergebnisse. Dies wird durch die folgende Abfolge erreicht:

- Zunächst werden alle Renderingdurchläufe zur Erzeugung dynamischer Texturen durchgeführt. Da diese den Framebuffer nicht verändern, ist ihre Reihenfolge untereinander nicht weiter relevant.

- Es folgt der Haupt-Renderingdurchlauf, in dem die Szene mit der Beleuchtung aller nicht durch einen Schatten beeinflusster Lichtquellen gerendert wird. Gegebenenfalls erfolgen die für Schatten benötigten Einstellungen der Lichtquellen durch eine Schattentechnik.
- Die stencilbasierte planare Reflexionstechnik markiert im nächsten Durchlauf spiegelnde Flächen in der Szene im Stencilpuffer. Anschließend folgt für jeden planaren Spiegel ein Renderingdurchlauf zum Rendern der gespiegelten Geometrie an den markierten Stellen. In einem weiteren Renderingdurchlauf wird die Färbung der Spiegelgeometrie an den markierten Stellen hinzuaddiert.
- Die texturbasierten Reflexionstechniken für planare Reflexionen und Environment-Mapping addieren in jeweils einem Renderingdurchlauf die anfangs in den dynamischen Texturen abgelegten gespiegelten Bilder auf die spiegelnde Geometrie.
- Schließlich fügen die Schattentechniken für jede Lichtquelle mit Schatten die Beleuchtung in extra Renderingdurchläufen hinzu. Für Schattenvolumen sind hierbei pro Lichtquelle zwei Durchläufe nötig: Im ersten werden die Schattenvolumenpolygone zur Markierung der beleuchteten Gebiete in den Stencilpuffer gerendert, im zweiten wird der Lichtanteil des Lichts an diesen Stellen hinzuaddiert. Bei der Renderingtechnik für Shadow-Mapping geschieht das Hinzufügen des Lichts an den beleuchteten Stellen analog in einem oder zwei Renderingdurchläufen.

Durchläufe der Transparenztechniken

Durchläufe zur Darstellung transparenter Geometrie-Objekte in Transparenztechniken werden nach Durchführung aller Durchläufe zur Berechnung von Beleuchtung erster Ordnung durchgeführt. Dadurch können Transparenztechniken mit Techniken für Beleuchtung erster Ordnung in einer Szene verwendet werden. Dabei werden aber bei transparenter Geometrie alle Attribute für Beleuchtung, die eine Wirkungseigenschaft bezeichnen, ignoriert. Transparente Objekte werden also nicht gleichzeitig mit Schatten oder als Spiegel dargestellt. Transparente Geometrie, für die eine Ursacheneigenschaft aktiv ist, beeinflusst die von ihr betroffene Geometrie so, als wäre die Geometrie nicht transparent. Beispielsweise wird in der Schattentechnik ignoriert, wenn ein schattenwerfendes Objekt teilweise lichtdurchlässig ist, und das Objekt stattdessen als lichtundurchlässig betrachtet.

Durchläufe der Markierungstechniken

Die Durchläufe von Markierungstechniken werden in zusätzlichen Renderingdurchläufen nach Beendigung alle anderen Durchläufe hinzugefügt. Die Markierungstechniken sind auf diese Weise mit allen anderen Techniken zusammen kombinierbar. Die zusätzlich dargestellten Effekte werden damit weder gespiegelt noch mit Schatten dargestellt.

Wie in Abschnitt 5.3 dargelegt, ist eine gespiegelte Darstellung von Markierungseffekten aber wünschenswert. Dazu kann eine Markierungstechnik sich zusätzlich für den Durchlauf zur gespiegelten Darstellung der Szene durch eine Reflexionstechnik registrieren. In diesem Durchlauf nimmt die Markierungstechnik keine Renderingeinstellungen vor, aber sie registriert einen weiteren, direkt folgenden Durchlauf, der eine gespiegelte Darstellung des Markierungseffekts zusätzlich zu dem bisherigen gespiegelten Bild bewirkt. Die Reflexionstechnik definiert dazu ein Intervall aufeinanderfolgender Renderingdurchläufe, die nach dem Spiegeldurchlauf durchgeführt werden können. Diese Durchläufe werden von ihr selbst im `TechniqueProcessor` nicht registriert, doch falls eine Markierungstechnik einen solchen Durchlauf registriert, ist in dem Durchlauf auch die Reflexionstechnik aktiv: Sie spiegelt in `preparePass()` die Kameraposition und rendert während der Szenengraphtraversierung gespiegelte Geometrie-Objekte. Die Markierungstechnik nimmt während des Durchlaufs die Renderingeinstellungen zur Darstellung der

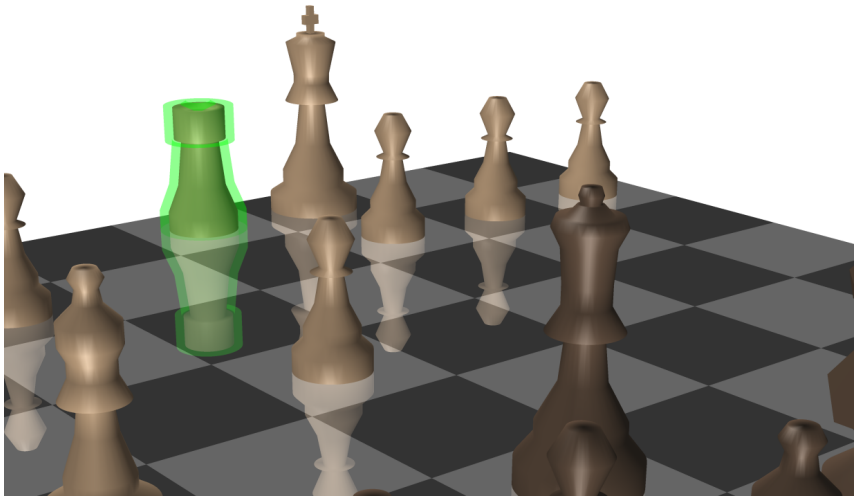

Abbildung 28: Spiegelndes Schachbrett: Der Markierungseffekt für den Turm wird ebenfalls gespiegelt dargestellt.

Markierungen vor. Insgesamt ergibt sich so die gespiegelte Darstellung eines Markierungseffekts (Abbildung 28).

6.4 Lokale Multipass-Verfahren

Eine weitere Variante von Auswertungsalgorithmen für den Szenengraph in VRS sind *lokale Renderingtechniken*, die nur einen Teil des Szenengraphen mehrfach durchlaufen und so lokale Multipass-Verfahren implementieren. Diese Techniken werden zur Auswertung von Shaderattributen, das heißt für Oberflächenschattierungen verwendet. Da immer nur eine Oberflächenschattierung pro Geometrie gleichzeitig aktiv ist, wird beim Hintereinanderauftreten mehrerer Shaderattribute eine rekursive Schachtelung der lokalen Traversierungen verhindert.

6.4.1 Überblick über die softwaretechnische Umsetzung

Lokale Renderingtechniken können evolutionär in bestehende Szenengraphsysteme integriert werden. Dazu sind die folgenden Erweiterungen erforderlich:

- **Gültigkeitsbereich einer lokalen Traversierung.** Eine lokale Traversierung eines Teils des Szenengraphen wird von lokalen Renderingtechniken während der Szenengraphauswertung durchgeführt: Bei Erreichen eines abstrakten Attributs wird überprüft, ob es sich bei ihm um ein Shaderattribut handelt, das heißt technisch, ob für die Auswertung des Attributs eine lokale Renderingtechnik registriert ist. Falls ja, wird die bis dahin verwendete lokale Renderingtechnik unterbrochen und die neue Technik gestartet. Nach Beendigung der Technik wird das Shaderattribut im Szenengraph temporär mit einer Markierung versehen, so dass eine rekursive Traversierung durch die neue Technik nur im ersten Durchlauf der vorigen lokalen Renderingtechnik durchgeführt wird, in deren weiteren Durchläufen aber der Teilgraph ignoriert wird. Auf diese Art ist für jeden Teil des Szenengraphen nur eine lokale Renderingtechnik zuständig.
- **Durchführung der lokalen Traversierungen.** Eine lokale Renderingtechnik definiert mehrere Renderingdurchläufe. Zu beachten ist hier der erste Durchlauf, der als einziger von anderen lokalen Renderingtechniken unterbrochen werden kann. Um auf eine solche Unterbrechung reagieren zu können, stellt eine lokale Renderingtechnik Methoden bereit, die beim Unterbrechen und Wiederaufnehmen des Durchlaufs aufgerufen werden.

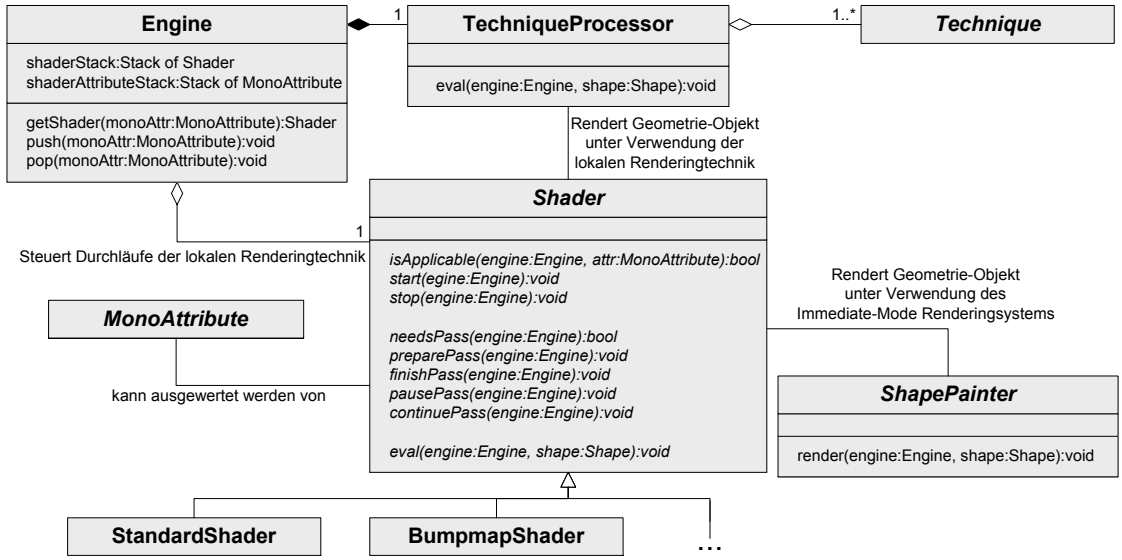

Abbildung 29: Klassendiagramm für lokale Renderingtechniken durch `Shader`.

- **Selektive Geometrie-Filterung und Anwendung zusätzlicher Attribute.** Diese Punkte sind die gleichen wie bei globalen Renderingverfahren mit Bezug auf einzelne Geometrie-Objekte (Abschnitt 6.3.1). Zu Beginn jedes Durchlaufs und für jedes einzelne Geometrie-Objekt können Attribute aktiviert und zum Ende wieder deaktiviert werden. Zudem kann es nötig sein, Geometrie in bestimmten Durchläufen der lokalen Renderingtechnik zu filtern.

6.4.2 Implementierung

Lokale Renderingtechniken werden durch Ableitung von der abstrakten Basisklasse `Shader` realisiert. Diese Klasse definiert ein Interface zum mehrfachen Iterieren über einen Teil des Szenengraphen, das von lokalen Renderingtechniken zur Auswertung eines Shaderattributs implementiert wird (Abbildung 29).

Verwaltung von lokalen Renderingtechniken und ihrer Durchläufe

Der Start eines `Shader`-Objekts geschieht, wenn ein Monoattribut in der Engine ausgewertet wird, für das dort eine entsprechende `Shader`-Klasse zur Auswertung des Attributs registriert ist. Dann wird die bisher verwendete lokale Renderingtechnik unterbrochen (`pausePass()`) und die neue gestartet (`start()`). Im weiteren Verlauf wird der folgende Teilgraph mehrfach traversiert. Zu den entsprechenden Zeitpunkten werden dabei die Methoden `preparePass()`, `finishPass()`, und `needsPass()` des `Shader`-Objekts aufgerufen, in denen Attribute für einen Renderingdurchlauf aktiviert bzw. wieder deaktiviert werden können, und wo überprüft wird, ob noch ein weiterer Renderingdurchlauf erfolgen muss. Zu jeder Zeit kann vom Evaluationskontext das aktive `Shader`-Objekt und das zugehörige Attribut erfragt werden, durch einen Stack für diese beiden Klassen in der `Engine`.

Zu Beginn der Szenengraphauswertung ist eine lokale Renderingtechnik der Klasse `StandardShader` aktiv. Diese beinhaltet das normale OpenGL-Beleuchtungsmodell, das heißt, sie definiert einen einzigen Durchlauf, in dem keine spezifischen Renderingeinstellungen vorgenommen werden. Es sind also alle Methoden der Basisklasse `Shader` bis auf die `eval()`-Methode zur Auswertung eines Geometrie-Objekts leer implementiert.

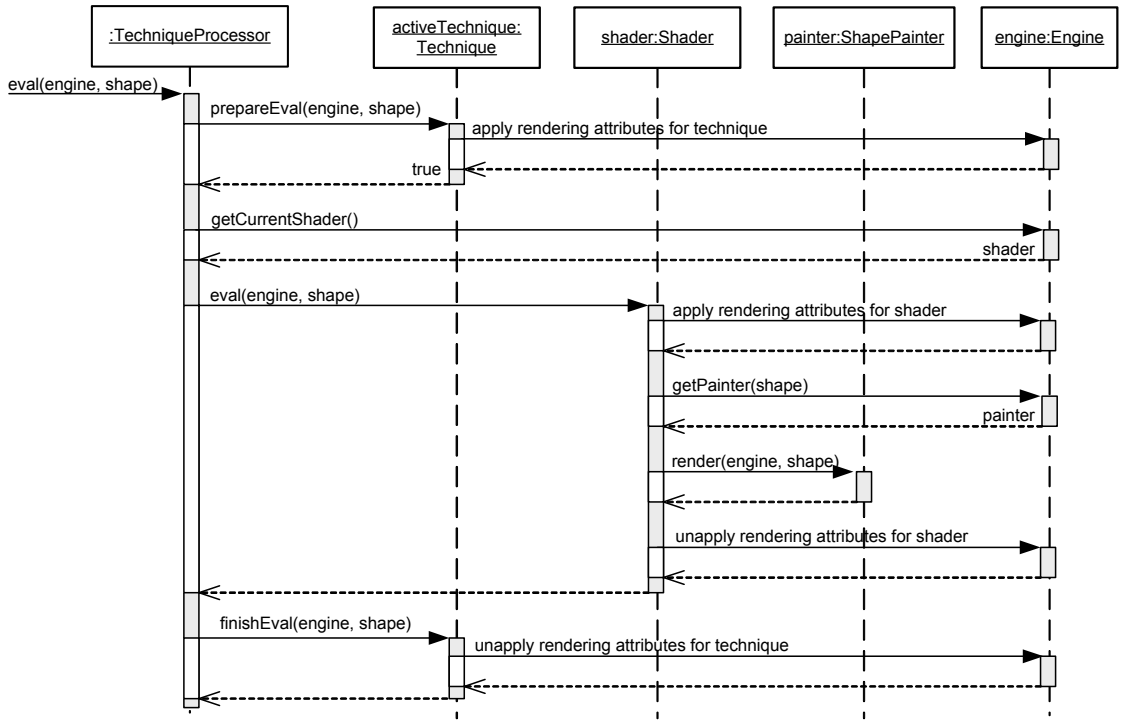

Abbildung 30: Sequenzdiagramm zur Auswertung eines Geometrie-Objekts durch `TechniqueProcessor` *und* `Shader`*.*

Auswertung einzelner Geometrie-Objekte

Wird während der Szenengraphtraversierung ein Geometrie-Objekt angetroffen, wird die Methode `TechniqueProcessor::eval()` aufgerufen. Falls diese anhand des aktiven `Technique`-Objekts entscheidet, die Geometrie zu rendern, ruft sie die Methode `eval()` des zur Zeit aktiven `Shader`-Objekts auf. Primäre Aufgabe dieser Methode ist der Aufruf einer Funktion zum Rendern des Geometrie-Objekts durch das Immediate-Mode Renderingsystem. Zusätzlich kann die Methode vor und nach diesem Aufruf, je nach Zustand des Evaluationskontextes, weitere Attribute aktivieren und wieder deaktivieren. Der Ablauf zur Auswertung eines Geometrie-Objekts wird in Abbildung 30 illustriert.

Eine Filterung von Geometrie-Objekten durch einen `Shader` ist möglich, indem die `eval()`-Methode den Aufruf zum Rendern eines Geometrie-Objekts nicht durchführt.

6.4.3 Zusammenarbeit globaler und lokaler Renderingtechniken

Lokale Renderingtechniken können grundsätzlich in jedem Renderingdurchlauf einer globalen Renderingtechnik durchgeführt werden. Die Ausführung ist aber je nach Art des Durchlaufs der globalen Technik manchmal nicht gewünscht. Deshalb wird bei globalen Renderingtechniken zwischen vier verschiedenen Typen von Durchläufen unterschieden, anhand derer ein `Shader`-Objekt entscheidet, ob es in dem Durchlauf benutzt wird oder stattdessen der `StandardShader` verwendet wird. Diese Entscheidung wird in der Methode `isApplicable()` des `Shader`-Objekts getroffen, die noch vor dessen Start aufgerufen wird. Zu Beginn des Durchlaufs einer globalen Renderingtechnik wird der Typ des Durchlaufs auf den Evaluationskontext abgelegt, damit ein `Shader`-Objekt auf diese Information Zugriff hat.

Typen von globalen Renderingdurchläufen

Vollschattierungsdurchläufe werden zur Darstellung der Geometrie mit allen aktiven Lichtquellen bzw. mit allen anwendbaren Oberflächenschattierungen verwendet. In diesen Durchläufen werden sämtliche lokale Renderingtechniken genutzt. Der Haupt-Renderingdurchlauf zur eigentlichen Darstellung der Szene und Durchläufe von Reflexionstechniken zur gespiegelten Darstellung der Szene sind Beispiele für Vollschattierungsdurchläufe.

Einzellichtdurchläufe werden zur Darstellung der Geometrie mit nur einer Lichtquelle verwendet. Sie werden von Schattentechniken genutzt, um den Lichtanteil einer Lichtquelle, von der Schattenberechnung beeinflusst, zu der Szene hinzuzuaddieren. Lokale Renderingtechniken können in Einzellichtdurchläufen genutzt werden, wenn sie die Beleuchtung bezüglich einer entsprechenden Lichtquelle berechnen sollen und können. Renderingtechniken für photorealistische Renderingeffekte werden im Allgemeinen so implementiert, dass sie dazu in der Lage sind, beispielsweise die Bumpmappingtechnik. Nicht-photorealistische Renderingeffekte hingegen, die unter Umständen keinen Bezug auf eine Lichtquelle nehmen, werden in Einzellichtdurchläufen ignoriert.

Geometriedurchläufe rendern Geometrie nicht in den Farb-, sondern nur in den Tiefenpuffer. Beispielsweise definiert die globale Renderingtechnik für Shadow-Mapping einen Geometriedurchlauf zur Erzeugung der Shadow-Map. Lokale Renderingtechniken zur Darstellung von Oberflächenschattierungen, die den Tiefenpuffer nicht beeinflussen, werden in diesen Durchläufen ignoriert.

In *Nichtschattierungsdurchläufen* wird die Ausführung von `Shader`-Objekten grundsätzlich verhindert. Solche Durchläufe werden verwendet für Analysedurchläufe, in denen überhaupt keine Geometrie gerendert wird. Auch Durchläufe von Markierungstechniken, in denen die Markierungen gerendert werden, sind in der Regel Nichtschattierungsdurchläufe: Die Markierungen werden von Oberflächenschattierungen normalerweise nicht beeinflusst und müssen damit unabhängig von den entsprechenden lokalen Renderingtechniken behandelt werden.

Diskussion

Diese Einteilung und Nutzung verschiedener Arten von Durchläufen löst die meisten Probleme bei der Zusammenarbeit globaler und lokaler Renderingtechniken. Spiegelbilder erscheinen mit der erwarteten Oberflächenschattierung. Zur Berechnung einer Shadow-Map werden nur Tiefenwerte berechnet, die unter Umständen komplexe und unnötige Farbschattierung aber nicht durchgeführt. Schattenwurf auf Geometrie kann auch dann berechnet werden, wenn die Geometrie mit einer Oberflächenschattierung dargestellt wird (Abbildung 31a). Markierungseffekte und Oberflächenschattierungen sind parallel einsetzbar.

Als Nachteil der Herangehensweise aus softwaretechnischer Sicht ist zu sehen, dass lokale Renderingtechniken für photorealistische Effekte zum einen auf Beleuchtung mit mehreren Lichtquellen für Vollschattierungsdurchläufe, zum anderen auf Beleuchtung mit nur einer Lichtquelle für Einzellichtdurchläufe ausgelegt sein müssen. Andere Ansätze könnten es erlauben, die Beleuchtungsberechnung nur für eine einzelne Lichtquelle zu spezifizieren und den Fall mehrerer Lichtquellen darauf zurückzuführen.

Eine weitere, allerdings kaum zu vermeidende Problematik ist, dass die Einstellungen der Renderingpipeline durch eine lokale Renderingtechnik während eines Einzellichtdurchlaufs kompatibel zu den verwendeten Schattentechniken gehalten werden müssen. Das heißt in der Praxis vor allem, dass nur additives Blending verwendet werden darf. Dies führt bei Bumpmappingtechniken für ältere Graphikhardware, die durch komplexe Blendingfunktionen und mehrere Durchläufe realisiert sind [42], zur Inkompatibilität mit den Schattentechniken. In Einzellicht-

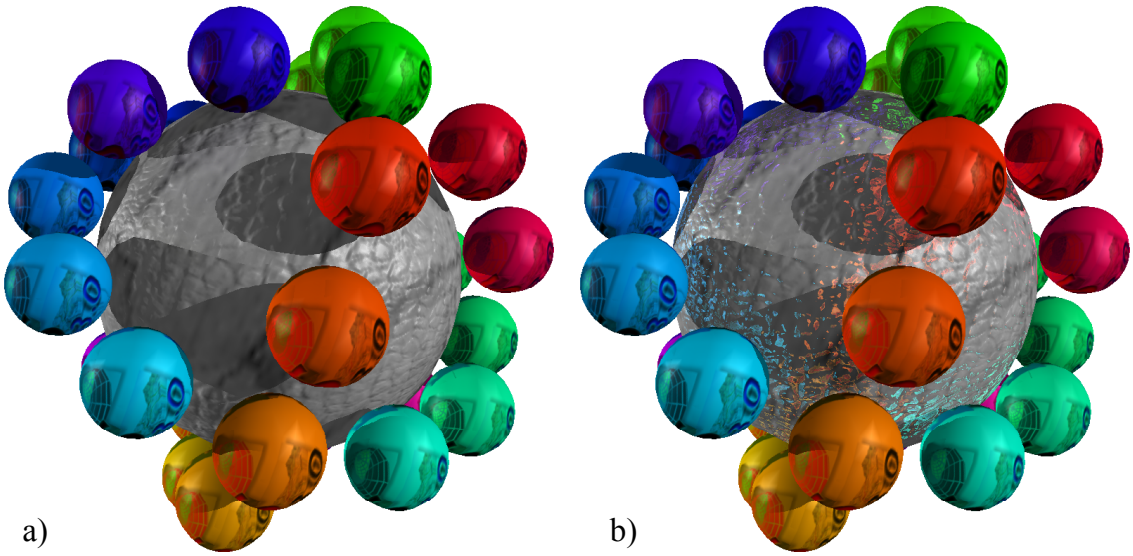

Abbildung 31: Kombination von Schatten mit Bumpmapping (a). Zusätzliche Reflexion durch Environment-Mapping mit Störung der Reflexionsrichtung durch die Bumpmap (b).

durchläufen wird daher solch eine Bumpmappingtechnik ignoriert. Mit neuerer Graphikhardware, die Bumpmapping für eine Lichtquelle in einem Renderingdurchlauf berechnen kann, taucht diese Schwierigkeit nicht auf. Diese Schwierigkeit ist daher nicht überzubewerten: Letztlich ist klar, dass ältere im Vergleich zu neuerer Hardware Einschränkungen hat, die sich auf die Möglichkeiten beim Rendering niederschlagen.

6.4.4 Beispielhafte Implementierung einer lokalen Bumpmappingtechnik

Abbildung 32 zeigt den Pseudocode einer vereinfachten lokalen Renderingtechnik zur Darstellung von Bumpmapping. Die Technik wird für `Bumpmap`-Attribute ausgewertet, die eine Liste der Lichtquellen enthalten, auf welche der Bumpmappingeffekt angewendet wird (Abschnitt 5.2). Alle anderen Lichtquellen werden zur Vereinfachung von der beschriebenen Technik ignoriert. Die Technik wird in Vollschattierungsdurchläufen und in Einzellichtdurchläufen verwendet; sichergestellt wird dies durch die Implementierung der Methode `isApplicable()`.

Die Methode `start()` stellt fest, welche Lichtquellen von der Technik bearbeitet werden sollen. In Vollschattierungsdurchläufen sind dies alle Lichtquellen, die in dem `Bumpmap`-Attribut enthalten sind. In Einzellichtdurchläufen dagegen wird nur die Lichtquelle betrachtet, die gerade von der globalen Schattentechnik bearbeitet wird, bzw. falls diese Lichtquelle nicht im `Bumpmap`-Attribut enthalten ist, ist die Liste der zu bearbeitenden Lichtquellen leer. Für jede Lichtquelle führt die Bumpmappingtechnik einen eigenen Renderingdurchlauf durch; die Schleife hierzu wird von der Methode `nextPass()` implementiert.

In `preparePass()` werden alle notwendigen Attribute zur Synthese des Bumpmappingeffekts aktiviert. Ist die Geometrie bereits einmal gerendert worden, das heißt bei Vollschattierungsdurchläufen ab der zweiten Lichtquelle, oder aber in Einzellichtdurchläufen, wird insbesondere additives Blending aktiviert, damit die Beleuchtungsberechnung für die Lichtquelle zu dem bisherigen Renderingergebnis hinzuaddiert wird. In `eval()` erfolgt der eigentliche Aufruf zum Rendering eines Geometrie-Objekts, allerdings nur, wenn die behandelte Lichtquelle sich an dieser Stelle im Szenengraph befindet und aktiv ist – dies könnte im Vollschattierungsdurchlauf wegen einer möglichen Schattenberechnung nicht der Fall sein. In `finishPass()` schließlich werden alle Attribute, die für den Durchlauf aktiviert wurden, wieder deaktiviert.

```
class BumpmapShader : public Shader {
    LightList: Array of Lights;        // Lights that are all rendered with bumpmap
    LightIndex: Index into MirrorList; // Index to LightList, corresponds to number of passes
    RequireBlending: bool;             // Different lights are rendered in several passes, and for
                                       // all but the first light, we require blending

    bool BumpmapShader::isApplicable(Engine e, Bumpmap bump) {
        PassHint hint ← e.getPassHint();
        if (hint.is(FULL_SHADING_PASS) || hint.is(SINGLE_LIGHT_SHADING_PASS)) {
            return true;
        } else {
            return false;
        }
    }

    void BumpmapShader::start(Engine e) {
        PassHint hint ← e.getPassHint();
        Bumpmap bump ← e.getShaderAttribute();
        if (hint.is(SINGLE_LIGHT_SHADING_PASS)) {
            if (hint.getLight() ∈ bump.getLight()) {
                LightList ← Array(hint.getLight());
            } else {
                LightList ← ∅;
            }
            RequireBlending ← True;
        } else {
            LightList ← bump.getLights();
            RequireBlending ← False;
        }
        LightIndex ← 0;
    }

    void BumpmapShader::stop(Engine) { }

    void BumpmapShader::preparePass(Engine e) {
        if (RequireBlending) {
            e.push(BlendFunc(ONE, ONE));
            e.push(DepthOp(DRAW IF EQUAL));
        }
        // push all textures, attributes, vertex, and fragment programs that enable bumpmap effect
        // with respect to LightList[LightIndex], i.e., the light handled in the upcoming pass
    }

    void BumpmapShader::finishPass(Engine e) {
        // pop all attributes that have been pushed in preparePass() before
    }

    bool BumpmapShader::needsPass(Engine e) {
        LightIndex++;
        return LightIndex < LightList.size();
    }

    void BumpmapShader::eval(Engine e, Shape shape) {
        Light light ← LightList[LightIndex];
        if (e.isActive(light)) { // The light object might be not active in this part of the scene
                                 // or it could be inactive due to the shadow calculation.
            ShapePainter painter ← e.getPainterFor(shape);
            painter.render(e, shape);
            RequireBlending ← True;
        }
    }
};
```

Abbildung 32: Pseudocode einer beispielhaften lokalen Bumpmappingtechnik.

6.4.5 Lokale Renderingtechniken in VRS

Zur Darstellung von Bumpmapping enthält VRS eine Reihe lokaler Renderingtechniken. All diese Techniken stellen die Beleuchtung von Lichtquellen, die nicht im `Bumpmap`-Attribut enthalten sind, mit normaler OpenGL-Beleuchtung dar – im Gegensatz zum exemplarischen `BumpmapShader` aus Abbildung 32, der diese Beleuchtung vernachlässigt. Zur Berechnung der Beleuchtung werten die Techniken die Materialeigenschaften der Geometrie im Szenengraphen

und die Oberflächentextur aus, um eine Gesamthelligkeit zu erhalten, die sich möglichst wenig vom normalen OpenGL-Beleuchtungsmodell unterscheidet.

Die verschiedenen Bumpmappingtechniken unterscheiden sich durch die verwendeten Basistechniken des Immediate-Mode Renderingsystems und, davon abhängig, in ihren Fähigkeiten: Insbesondere können einige Techniken für ältere Graphikhardware Bumpmapping nur mit einer Lichtquelle darstellen, alle weiteren Lichtquellen im `Bumpmap`-Attribut werden dann mit normaler OpenGL-Beleuchtung dargestellt.

- Die `BumpmapShaderARBFP`-Technik implementiert Bumpmapping auf Basis von Vertex- und Fragmentprogrammen. Sie kann beliebig viele Lichtquellen bearbeiten und unterstützt die Darstellung zusammen mit Schatten.
- Die `BumpmapShaderGF3`-Technik implementiert Bumpmapping auf Basis von Vertexprogrammen und der OpenGL-Erweiterung `GL_NV_register_combiners` [42]. Sie funktioniert damit nur auf Graphikhardware von NVidia ab der GeForce3. Sie kann ebenfalls beliebig viele Lichtquellen zusammen mit Schatten bearbeiten.
- Die `BumpmapShaderGF1`-Technik implementiert Bumpmapping auf Basis von Vertexprogrammen und der OpenGL-Erweiterung `GL_NV_register_combiners`, verwendet aber statt vier nur zwei Texturen zugleich und ist damit bereits mit GeForce256-Hardware lauffähig. Die Technik berücksichtigt nur die erste Lichtquelle in einem `Bumpmap`-Attribut und bei dieser auch keine Schatten. Entsprechend wird sie in Einzellichtdurchläufen ignoriert.
- Die `BumpmapShaderARB`-Technik implementiert Bumpmapping mit der OpenGL-Erweiterung `GL_ARB_texture_env_dot3`. Der Bumpmappingeffekt wird mit dem verwendeten Multipass-Verfahren nur mit diffuser und nicht mit spekularer Beleuchtung berechnet, ebenfalls bezüglich nur einer Lichtquelle und ohne Schatten. Der Vorteil des Verfahrens ist, dass es nur standardisierte OpenGL-Erweiterungen benutzt und damit auch auf älterer, nicht von NVidia stammender Hardware funktioniert.
- Die `BumpmapShaderCombine`-Technik implementiert Bumpmapping mit der OpenGL-Erweiterung `GL_ARB_texture_env_combine`. Das verwendete lokale Multipass-Verfahren verwendet damit als einziges kein Skalarprodukt pro Fragment zur Berechnung der Beleuchtung, sondern nutzt ein anderes Beleuchtungsmodell [53]. Die Renderingqualität des Verfahrens ist schlechter als mit den anderen Verfahren, aber dafür hat es die geringsten Anforderungen an die Graphikhardware.
- Die `BumpmapShaderGL`-Technik wählt anhand der vorhandenen OpenGL-Erweiterungen eine der obigen Techniken aus und verwendet diese zur Darstellung des Bumpmappingeffekts. Im Allgemeinen verwendet man als Nutzer von VRS lediglich diese Technik.

Abbildung 33 zeigt die Ausgabe der verschiedenen Bumpmappingtechniken bei Verwendung einer Lichtquelle. Weil die `BumpmapShaderARBFP`-, `BumpmapShaderGF3`- und bei einer Lichtquelle die `BumpmapShaderGF1`-Technik nahezu dasselbe Beleuchtungsmodell implementieren, sind zwischen ihren Bildern kaum Unterschiede zu erkennen. Bei der `BumpmapShaderARB`-Technik fehlt im Vergleich die spekulare Beleuchtung; bei der `BumpmapShaderCombine`-Technik ist spekulare Beleuchtung zu erkennen, die aber von der Bumpmap nicht beeinflusst ist.

Zum Vergleich ist ein Bild der Geometrie ohne Bumpmapping abgebildet. Dabei ist erkennbar, dass die Grundhelligkeit der Geometrie durch das Bumpmapping nicht erkennbar verändert wird, zumindest verglichen mit den qualitativ besseren Varianten wie zum Beispiel der `BumpmapShaderARBFP`-Technik. Dies ermöglicht das Umschalten zwischen einer Darstellung mit

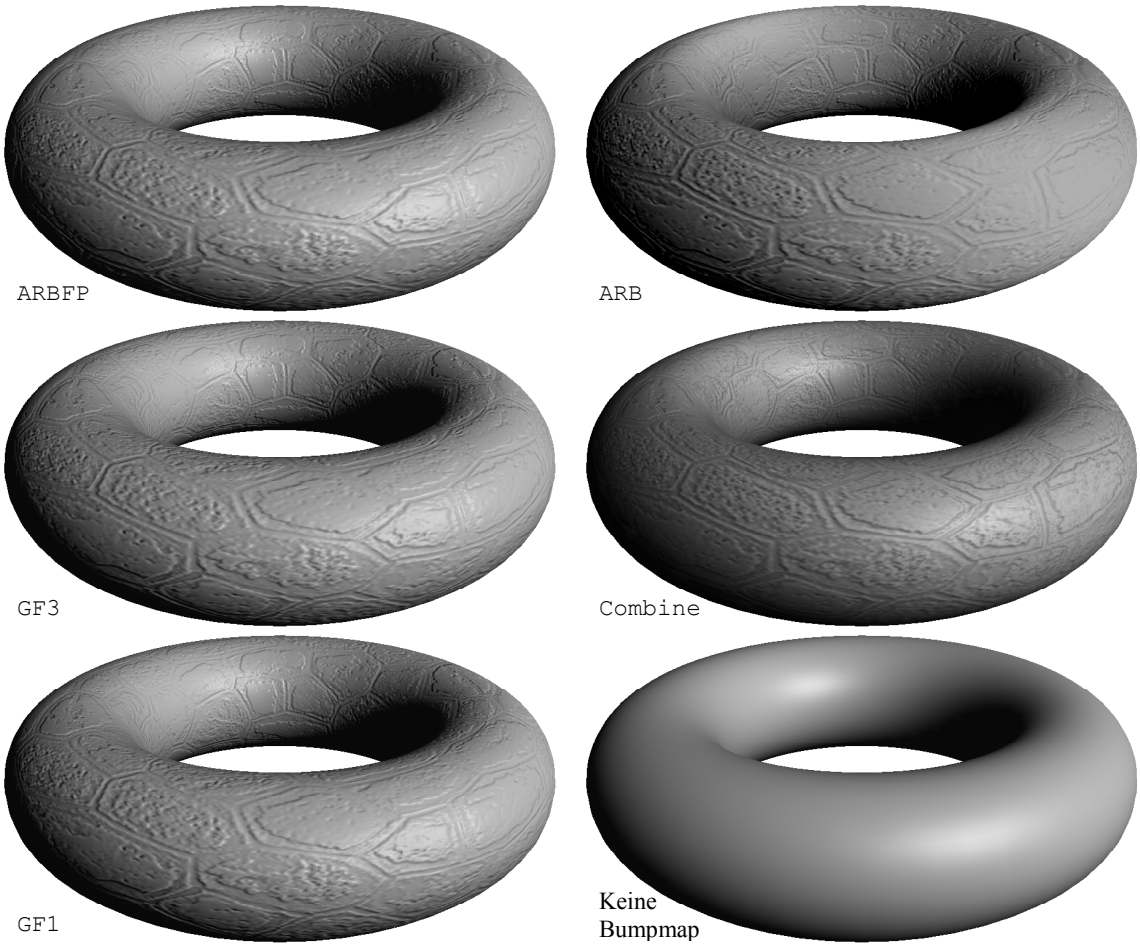

Abbildung 33: Ausgabe verschiedener Bumpmappingtechniken in VRS. Rechts unten ein Vergleichsbild ohne Bumpmapping.

und ohne Bumpmapping bei geringen Intensitätsunterschieden und somit ohne Änderungen sonstiger Lichtquellen- und Materialeigenschaften in der Szene.

Andere Shaderattribute zur Deklaration von NPR-Oberflächenschattierungen in VRS werden ebenfalls durch entsprechende lokale Renderingtechniken ausgewertet. Diese Techniken werden in dieser Arbeit nicht besprochen.

6.5 Optimierungen der Renderinggeschwindigkeit

Die meisten Optimierungen zur Steigerung der Renderinggeschwindigkeit bei der Szenengraphauswertung müssen angepasst werden, um mit Renderingtechniken zur Auswertung von abstrakten Attributen zusammen zu funktionieren. Weiterhin gibt es eine Reihe neuer Ansatzpunkte zur Steigerung der Geschwindigkeit bei der Implementierung von Renderingtechniken.

Anpassung bekannter Szenengraphoptimierungen

View-Frustum Culling, das heißt die Nichtdarstellung von 3D-Geometrie außerhalb des Sichtbarkeitsbereichs durch einen objektbasierten Test des Szenengraphsystems. Vor allem texturbasierte Verfahren zur Beleuchtung erster Ordnung verwenden bei der Erstellung einer dynamischen Textur eine veränderte Kameraposition. In so einem Durchlauf muss diese veränderte

Kameraposition für das View-Frustum Culling verwendet werden. In Szenengraphsystemen, bei denen View-Frustum Culling als Vorverarbeitungsschritt vor der Szenengraphauswertung durchgeführt wird, bedeutet das, dass dieser Schritt vor jedem Durchlauf einer globalen Renderingtechnik durchgeführt werden muss.

Occlusion-Culling, das heißt die Nichtdarstellung von 3D-Geometrie, die von anderer Geometrie verdeckt wird, durch einen objektbasierten Test des Szenengraphsystems. Auch hierbei muss eine veränderte Kameraposition in einem Renderingdurchlauf berücksichtigt werden. Weil die verdeckenden Polygone normalerweise separat in Hinblick auf die Position der Szenenkamera deklariert werden, wird der Verdeckungstest mit einer anderen Kameraposition leicht fehlschlagen, so dass die Geometrie dennoch gerendert werden muss. Daher sind die zu erwartenden Geschwindigkeitsvorteile von Occlusion-Culling zusammen mit Beleuchtung erster Ordnung kleiner als ohne.

Multiresolutionsmodellierung, also die entfernungsabhängige Darstellung unterschiedlich detaillierter Geometriemodelle, muss ebenfalls eine veränderte Kameraposition in einem Durchlauf zur Erzeugung dynamischer Texturen berücksichtigen.

State-Sorting, das heißt die interne Umordnung der Szenenbeschreibung, so dass die Anzahl der Konfigurationsänderungen des Renderingkontextes zum Rendern eines Bilds minimiert wird. Weil abstrakte Attribute nicht direkt auf Einstellungen des Renderingkontextes abgebildet werden, wird auch das State-Sorting sehr viel komplizierter. Letztlich muss die Sortierung für jeden Durchlauf einer globalen Renderingtechnik separat durchgeführt werden, zusätzlich aber auch für alle Durchlaufe von lokalen Renderingtechniken. Besonders schwierig ist dies, weil sogar für einzelne Geometrie-Objekte je nach Zustand des Evaluationskontextes konkrete Attribute aktiviert werden können. Es ist davon auszugehen, dass zur Implementierung eines robusten und allgemeinen Algorithmus für State-Sortung in Kombination mit globalen und lokalen Renderingtechniken grundlegende algorithmische Veränderungen und Erweiterungen erforderlich wären, die den Rahmen dieser Arbeit sprengen würden.

Kompilation statischer Teilgraphen. In der Regel erfolgt die Kompilation statischer Teile von Szenengraphen durch die Erzeugung von Display-Listen, welche die Renderingbefehle zur Auswertung eines Teilgraphen zwischenspeichern. Attribute in einem statischen Teilgraph, die durch lokale Renderingtechniken ausgewertet werden, können mit dieser Heransgehensweise ohne Probleme ausgewertet werden, weil ein solches Attribut nur lokale Auswirkungen in dem Teilgraphen hat. Für abstrakte Attribute, die mit globalen Renderingtechniken ausgewertet werden, gilt dies nicht. Dort müsste für jeden Renderingdurchlauf eine eigene Display-Liste abgespeichert werden. Dieser Ansatz ist aufgrund der hohen Speicherplatzanforderungen von Display-Listen nur für kleine Szenen anwendbar.

Zwischenspeicherung dynamischer Texturen

Alle globalen Renderingtechniken, die dynamische Texturen verwenden, gewinnen an Geschwindigkeit, wenn dynamische Texturen, die sich im Vergleich zu einem früheren Bild nicht verändert haben, wiederverwendet werden. In diesem Fall müssen Shadow-Maps, Cubemaps für Environment-Mapping oder Texturen für planare Reflexionen nicht neu erzeugt werden, und die Renderingdurchläufe dafür müssen nicht durchgeführt werden.

Die folgenden Fälle erfordern die Neuberechnung dynamischer Texturen: Shadow-Maps und Cubemaps für Environment-Mapping müssen neu erzeugt werden, wenn die Position der Lichtquelle bzw. der spiegelnden Geometrie sich verändert, oder wenn handelnde Geometrie-Objekte in der Szene verändert, neu hinzugefügt oder entfernt werden. Texturen für planare Reflexionen

müssen in diesen Fällen ebenfalls neu erzeugt werden, zusätzlich aber auch, wenn die Kameraposition sich verändert.

Ein wichtiger Fall, bei dem für Shadow-Mapping und Environment-Mapping eine Neuerzeugung dynamischer Texturen nicht erforderlich ist, sind virtuelle Spaziergänge (walkthroughs), beispielsweise in Stadtmodellen. Hier ist die dargestellte Szene statisch, Geometrie und Lichtquellen in der Szene ändern sich also nicht, und nur die Kameraposition wird variiert.

Eine automatische Erkennung, wann dynamische Texturen neu berechnet werden müssen, ist aufwändig zu implementieren: Änderungen im gesamten Szenengraph können, aber müssen nicht unbedingt zur Notwendigkeit der Neuberechnung einer dynamischen Textur führen. Beispielsweise können geometrische Transformationen, die sich auf handelnde Geometrie auswirken, die Neuberechnung erfordern. Die Überwachung solcher Änderungen des Szenengraphen ist aber sehr aufwändig. Als pragmatische Lösung stellt VRS die Möglichkeit bereit, die Neuberechnung einer dynamischen Textur manuell auszulösen. Dies geschieht durch eine Methode in einem `ShadowCasting`- oder `Mirrored`-Attribut, durch die die entsprechende dynamische Textur in der entsprechenden globalen Renderingtechnik als ungültig markiert wird.

Adaptiver Transfer von Vertex-Daten

Geometrie in einer Szenenbeschreibung enthält gewöhnlich eine Menge von Vertex-Daten wie Eckpunktpositionen, Normalen, Texturkoordinaten, daneben auch weitere wie beispielsweise Tangenten oder Binormalen. Diese Vertex-Daten werden aber nicht in jedem Durchlauf einer Renderingtechnik benötigt. Um die Menge von Daten, die für ein Geometrie-Objekt durch die Renderingpipeline geschickt wird, zu minimieren, ist es daher wichtig, nur die wirklich erforderlichen Daten zu übermitteln.

Dazu dienen in VRS Attribute der Klasse `VertexAttributeMap`. Eine Instanz dieser Klasse enthält eine Menge von Zuordnungen, die jeweils eine abstrakte Beschreibung von Vertex-Daten einem OpenGL-Register für Vertex-Daten zuordnen. Beispielsweise könnte eine abstrakte Beschreibung die Information enthalten, dass Binormalen zur Durchführung einer Renderingtechnik benötigt werden, und das entsprechende OpenGL-Register, dass die Binormalen im Register für die zweite Texturkoordinate erwartet werden. Ein `VertexAttributeMap`-Objekt wird beim Rendern aller Geometrie-Objekte ausgewertet, und nur die darin referenzierten OpenGL-Register werden mit Vertex-Daten gefüllt. In der Praxis wird die Auswertung jedes Geometrie-Objekts in VRS auf die Auswertung eines Objekts der Klasse `MappedVertexAttributeShapeGL` zurückgeführt, welches grundlegende polygonale OpenGL-Geometrie enthält. Daher muss die Auswertung des aktiven `VertexAttributeMap`-Objekts nur an einer zentralen Stelle, nämlich in der Funktion zum Rendern von `MappedVertexAttributeShapeGL`-Objekten, vorgenommen werden.

`VertexAttributeMap`-Objekte werden fast immer von Renderingtechniken selbst aktiviert und wieder deaktiviert, obwohl sie als Attribut auch Element einer Szenenbeschreibung sein könnten. Beispielsweise werden durch diesen Mechanismus im Renderingdurchlauf zum Erzeugen einer Shadow-Map ausschließlich Eckpunktinformationen zur Graphikhardware geschickt. Ebenso genügt diese kleine Anzahl von Vertex-Daten in Durchläufen zur Darstellung einer einfarbigen Markierung durch eine Markierungstechnik. Zur Verwendung einer Renderingtechnik für Bumpmapping sind hingegen in der Regel Eckpunktinformationen, Normalen, Texturkoordinaten und Tangenten zugleich erforderlich, und auf Graphikhardware, die keine Vertexprogramme unterstützt, müssen unter Umständen noch weitere aus diesen Daten durch die CPU berechnete Informationen übertragen werden.

Erweitertes View-Frustum und Occlusion Culling

Gewöhnliches View-Frustum bzw. Occlusion Culling filtert Geometrie, die im endgültigen Bild nicht sichtbar ist. Wenn von Beleuchtung erster Ordnung betroffene Geometrie-Objekte durch diese Optimierungen gefiltert werden, sind Renderingdurchläufe von Schatten- und Reflexionstechniken, die nur zur Schattierung dieser Geometrie durchgeführt würden, ebenfalls nicht mehr erforderlich. Die Szenengraphauswertung kann insgesamt erfolgen, als befänden sich die Attribute zur Deklaration der entsprechenden Wirkungs- und Ursacheneigenschaften nicht im Szenengraph.

Verminderung der Anzahl von Renderingdurchläufen

Eine geringe Anzahl von Renderingdurchläufen ist im Allgemeinen notwendig, um eine bestmögliche Geschwindigkeit bei der Szenengraphauswertung zu erhalten. Mehrere Durchläufe einer einzigen Renderingtechnik zusammenzufassen, ist dabei ein bekanntes Optimierungsproblem in der Computergraphik und, falls in aufeinanderfolgenden Durchläufen dieselbe Geometrie gerendert wird, in der Regel durch Verwendung komplexerer Vertex- und Fragmentprogramme möglich. Durchläufe verschiedener Renderingtechniken zu weniger Durchläufen zusammenzufassen, verkompliziert dagegen Wechselbeziehungen der beteiligten Renderingtechniken und somit die Komplexität der einzelnen Techniken. Hier gilt es als Entwickler eines Szenengraphsystems einen Kompromiss zu finden.

Ein häufig auftretender Fall, für den das Zusammenfassen von Durchläufen daher sinnvoll sein kann, ist die globale Technik für Shadow-Mapping zusammen mit lokalen Techniken für photorealistische Oberflächenschattierungen. Die Schattentechnik führt für jede Lichtquelle die folgenden Durchläufe durch: 1) Es wird die Shadow-Map erzeugt; 2) Die Shadow-Map wird auf die Szene projiziert, dabei wird der texturbasierte Schattentest durchgeführt und das Ergebnis, also ob Schatten oder nicht, in den Alphapuffer geschrieben; 3) Die Beleuchtung der Lichtquelle wird durch die lokale Renderingtechnik berechnet und zur Szene je nach Wert im Alphapuffer durch additives Blending hinzuaddiert.

Der zweite und dritte Durchlauf können in der Regel zu einem zusammengefasst werden. Dazu muss allerdings während der Beleuchtungsberechnung durch die lokale Renderingtechnik eine Textur verfügbar sein, die die Shadow-Map aufnehmen kann. Außerdem müssen die verwendeten Vertex- und Fragmentprogramme, bzw. die Einstellungen der festverdrahteten Renderingpipeline, so angepasst werden, dass bei der Beleuchtungsberechnung zusätzlich die notwendige Texturkoordinatengenerierung zur Projektion der Shadow-Map verwendet, der texturbasierte Schattentest durchgeführt und sein Ergebnis zur Berechnung der Beleuchtung berücksichtigt wird. Das Problem bei diesen Erweiterungen ist die vergrößerte Komplexität der Implementierung: Die lokalen Renderingtechniken müssen Code-Pfade mit und ohne Schatten unterscheiden, und die Schattentechnik muss überwachen, ob das Zusammenfassen der Durchläufe bei allen verwendeten lokalen Renderingtechniken möglich ist. In Fällen, bei denen beispielsweise bereits alle Texturen belegt sind, muss die Schattenberechnung auf herkömmlichen Wege durchgeführt werden.

In der Praxis waren unsere Erfahrungen mit dieser Optimierung zwiespältig. Die Geschwindigkeit mit Bumpmappingtechniken oder mit der Standardschattierung wurde um etwa 10 bis 20% verbessert, jedoch auf Kosten einer erhöhten Fehlerquote der Implementierung.

Kapitel 7

BILDBASIERTES CSG

Die *Konstruktive Festkörpergeometrie* (Constructive Solid Geometry, CSG) bezeichnet ein Verfahren zur Modellierung von komplexen Geometrie-Objekten durch hierarchische Verknüpfung einfacherer geometrischer Primitive [70]. Die Grund-Primitive sind topologisch geschlossen, das heißt, ihr Inneres und Äußeres ist eindeutig bestimmt. Durch Vereinigung, Intersektion oder Subtraktion von jeweils zwei Primitiven werden komplexere geometrische Objekte definiert, die ihrerseits wiederum als Argumente dieser volumetrischen Mengenoperationen verwendet werden können. Insgesamt ist also eine CSG-Geometrie definiert durch einen Baum, dessen Blattknoten Primitive und dessen innere Knoten Mengenoperationen sind (Abbildung 34).

Ein CSG-Baum kann von der Graphikhardware nicht direkt gerendert werden. Zum Rendern wird gewöhnlich analytisch die Berandung der Geometrie als Dreiecksdarstellung berechnet (die sogenannte boundary representation, [85]) und zur Graphikhardware geschickt. Für interaktive Modellierung von CSG ist dies allerdings sehr aufwändig und für komplexe Modelle in Echtzeit kaum möglich. Eine andere Möglichkeit zum Rendern von CSG-Bäumen ist die Verwendung von bildbasiertem CSG [26]: Dabei wird unter Verwendung von Operationen des Tie-

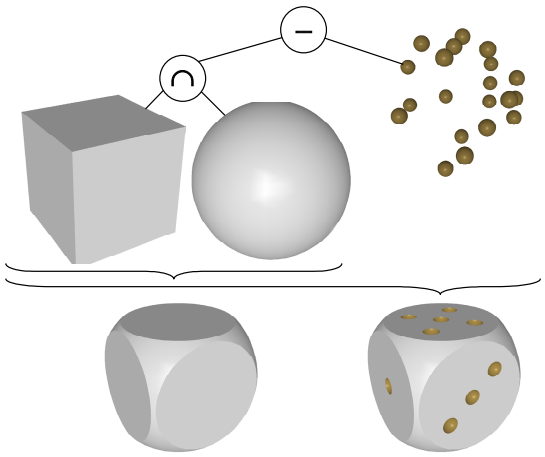

Abbildung 34: Modellierung eines Würfels durch CSG. Der Würfel wird mit der Kugel geschnitten, vom Ergebnis werden die Würfelpunkte entfernt.

fen- und Stencilpuffers auf der Graphikhardware die Tiefeninformation für einen CSG-Baum abgelegt. Die endgültige Geometrie des CSG-Baums wird also überhaupt nicht berechnet. Dazu müssen die einzelnen CSG-Primitive mehrfach unsichtbar gerendert werden, so dass es sich bei bildbasiertem CSG um eine Form des Multipass-Renderings handelt.

Bildbasiertes CSG ist ein weitgehend von anderen Arbeitsgebieten der Computergraphik unabhängiger und zu diesen orthogonaler Bereich. Daher ist es sinnvoll, ein API für bildbasiertes CSG in einer separaten Bibliothek zu implementieren. In dieser Arbeit wurde eine solche Renderingbibliothek mit dem Namen OpenCSG entworfen und implementiert [45]. Sie benötigt als Voraussetzung lediglich OpenGL und kann von jeder anderen Graphikanwendung, die auf OpenGL basiert, genutzt werden.

7.1 Algorithmen für bildbasiertes CSG

Allen existierenden Algorithmen für bildbasiertes CSG ist gemeinsam, dass sie als Eingabe nicht einen beliebigen CSG-Baum, sondern einen vereinfachten CSG-Baum in Normalform haben. Ein CSG-Baum in Normalform besteht aus der Vereinigung mehrerer partieller Produkte, wobei ein partielles Produkt ein CSG-Baum ist, in dem nur Intersektionen und Subtraktionen vorkommen und zusätzlich der rechte Operand dieser Operationen jeweils ein Primitiv ist. Ein partielles Produkt ist also von der Form $(\ldots(x_1 \otimes x_2) \otimes x_3) \ldots \otimes x_n)$, wobei das Symbol $\otimes$ entweder eine Intersektion oder eine Subtraktion bezeichnet. Unter Berücksichtigung der Beziehung $x - y = x \cap y^c$ kann so jedes partielle Produkt als Intersektion von komplementierten oder nichtkomplementierten Primitiven geschrieben werden.

Goldfeather et al. [27] haben gezeigt, dass ein beliebiger CSG-Baum durch wiederholte Anwendung der folgenden mengentheoretischen Umformungen auf seine inneren Knoten in eine äquivalente Normalform umgewandelt werden kann:

a. $x - (y \cup z) \rightarrow (x - y) - z$

b. $x \cap (y \cup z) \rightarrow (x \cap y) \cup (x \cap z)$

c. $x - (y \cap z) \rightarrow (x - y) \cup (x - z)$

d. $x \cap (y \cap z) \rightarrow (x \cap y) \cap z$

e. $x - (y - z) \rightarrow (x - y) \cup (x \cap z)$

f. $x \cap (y - z) \rightarrow (x \cap y) - z$

g. $(x - y) \cap z \rightarrow (x \cap z) - y$

h. $(x \cup y) - z \rightarrow (x - z) \cup (y - z)$

i. $(x \cup y) \cap z \rightarrow (x \cap z) \cup (y \cap z)$

7.1.1 Der Algorithmus von Goldfeather

Der *Algorithmus von Goldfeather* [27] berechnet die Sichtbarkeit jedes einzelnen Primitivs für sich. Dabei spielen für die eigentliche Sichtbarkeitsberechnung für ein Primitiv nur die anderen Primitive des zugehörigen partiellen Produkts eine Rolle, die Sichtbarkeit in Bezug auf andere partielle Produkte bzw. sonstige Objekte in der Szene erfolgt durch den normalen Tiefentest.

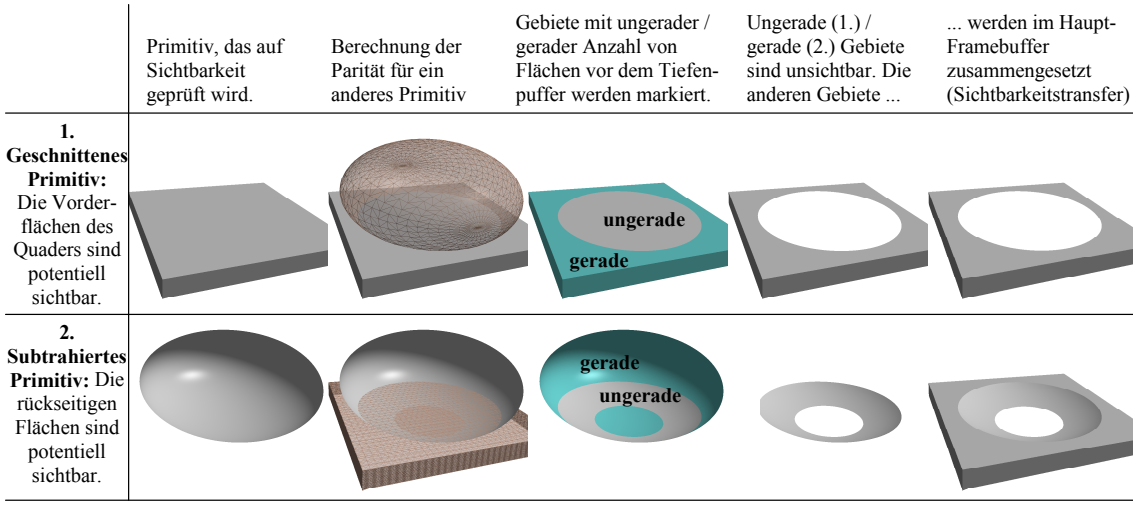

Abbildung 35: Arbeitsweise des Algorithmus von Goldfeather.

Grundlegender Algorithmus

Zum Verständnis des Algorithmus von Goldfeather ist es hilfreich, sich zunächst auf konvexe Primitive zu beschränken. In diesem Fall gelten mehrere Beobachtungen: 1. Nur die Vorderflächen eines geschnittenen bzw. die rückseitigen Flächen eines subtrahierten Primitivs können sichtbar sein. Diese Flächen, die mit Front- oder Back-face Culling ermittelt werden können, heißen *potentiell sichtbar*. 2. Jedes weitere Primitiv Q im partiellen Produkt beeinflusst die Sichtbarkeit eines potentiell sichtbaren Pixels durch die Parität, das heißt, durch die Anzahl der Polygone von Q, die sich vor dem Pixel befinden: Falls Q subtrahiert wird, kann das Pixel bei einer Parität von 1 nicht sichtbar sein; falls Q geschnitten wird, kann das Pixel bei einer Parität von 0 oder von 2 nicht sichtbar sein. In allen anderen Fällen beeinflusst Q die Sichtbarkeit des Pixels nicht. Für den Fall konvexer Primitive liegt die Parität dabei immer zwischen 0 und 2.

Die Funktionsweise des Algorithmus von Goldfeather basiert auf diesen Beobachtungen (Abbildung 35): Die Sichtbarkeit jedes Primitivs P im partiellen Produkt wird separat getestet. Dazu werden die potentiell sichtbaren Flächen von P in einen temporären, zunächst leeren Tiefenpuffer gerendert. Danach wird für alle anderen Primitive Q im partiellen Produkt die Parität berechnet, das heißt, die Anzahl der Polygone eines Primitivs Q vor dem Tiefenpuffer wird bestimmt, indem Q mit Tiefentest, aber ohne Aktualisierung des Tiefenpuffers gerendert und bei Gelingen des Tiefentests ein Bit im Stencilpuffer invertiert wird. Wenn nach der Berechnung der Paritäten für alle Q keine Parität anzeigt, dass ein Pixel unsichtbar ist, wird es als sichtbar markiert, während alle übrigen Tiefenwerte zurückgesetzt werden. Der Inhalt des temporären Tiefenpuffers wird dann unter Berücksichtigung eines gewöhnlichen Tiefentests in den Haupt-Tiefenpuffer kopiert; dies wird im Folgenden als *Sichtbarkeitstransfer* bezeichnet. Danach wird mit der Berechnung der Sichtbarkeit des nächsten Primitivs fortgefahren.

Behandlung allgemeiner Fälle

Zur Verwendung des Algorithmus von Goldfeather in allgemeinen Fällen sind die folgenden Erweiterungen des obigen Algorithmus nötig:

Unterstützung für konkave Primitive. Hierzu muss die Sichtbarkeit jeder potentiell sichtbaren Tiefenschicht eines Primitivs separat getestet werden. Die Bestimmung der Tiefenschichten ei-

nes Primitivs kann mit dem Stencilpuffer erfolgen[1]. Außerdem muss die Berechnung der Parität dahingehend erweitert werden, dass eine ungerade von einer geraden Anzahl von Polygonen vor einem Pixel unterschieden werden kann. Dies bedeutet in der Praxis allerdings keine Änderung und kann wie in der bisherigen Beschreibung des Algorithmus durch das Invertieren eines Stencilbits geschehen.

Unterstützung beliebig vieler Primitive in einem partiellen Produkt. Der Stencilpuffer hat auf gewöhnlicher Graphikhardware eine Tiefe von lediglich acht Bit, so dass gleichzeitig die Ergebnisse von nur acht Paritätsberechnungen dort abgelegt werden können. Falls ein partielles Produkt mehr Primitive enthält, ist es deshalb erforderlich, bereits als unsichtbar erkannte Pixel dauerhaft als solche zu kennzeichnen, bevor weitere Paritätsberechnungen durchgeführt werden. Dies kann zum Beispiel dadurch geschehen, dass nach acht Paritätsberechnungen der Tiefenwert für alle als unsichtbar erkannten Pixel zurückgesetzt und der Stencilpuffer gelöscht wird.

Ausnutzung der Tiefenkomplexität

Die Laufzeitkomplexität des Algorithmus von Goldfeather für ein partielles Produkt ist $O(n^2)$ mit n als Anzahl der Primitive im partiellen Produkt. Nach Stewart et al. [78] ist die Tiefenkomplexität k der Primitive in einem partiellen Produkt oft wesentlich kleiner als n. In diesen Fällen ist die Verwendung des *geschichteten Goldfeather Algorithmus* (layered Goldfeather algorithm) sinnvoll, bei dem die Sichtbarkeit einer Tiefenschicht des partiellen Produkts anstatt der eines einzelnen Primitivs ermittelt wird. Bei k Tiefenschichten ist die Laufzeitkomplexität dieses Algorithmus $O(n \cdot k)$.

Guha et al. beschreiben einen bildbasierten CSG-Algorithmus [33], bei dem es sich letztlich um eine Variante des geschichteten Goldfeather Algorithmus handelt. Bei ihrem Algorithmus erfolgt die Überprüfung, welche Teile eines Primitivs sichtbar sind, mit einer Umformulierung der Paritätsberechnung von Goldfeather. Als wesentliche Erweiterung nutzen Guha et al. eine Tiefenschichtenermittlung (Abschnitt 3.3.2), um potentiell sichtbare Flächen von CSG Primitiven von vorne nach hinten zu ermitteln. Dort, wo ein Primitiv sich als wirklich sichtbar erwiesen hat, verhindert eine Markierung im Stencilpuffer ein weitere Suche potentiell sichtbarer Flächen dahinter. Der wesentliche Vorteil dieser Vorgehensweise liegt darin, dass alle Berechnungen im Haupt-Framebuffer durchgeführt werden können und somit der Transfer von Informationen von einem temporären Tiefenpuffer nicht gebraucht wird. Der Nachteil ist die inhärente, relativ kostspielige Tiefenschichtenermittlung, bei der pro Bild mehrfach das Erzeugen einer dynamischen Tiefentextur erforderlich ist.

7.1.2 Der SCS-Algorithmus

Der *SCS-Algorithmus* [79] ist speziell für konvexe CSG-Primitive entwickelt worden und daher für partielle Produkte, die nur aus konvexen Primitiven bestehen, nahezu immer schneller als der Algorithmus von Goldfeather. Andererseits funktioniert der SCS Algorithmus für konkave Primitive gar nicht. Im Folgenden wird die vereinfachte Darstellung von Stewart et al. beschrieben [80].

Der SCS-Algorithmus erfordert, wie alle bildbasierten Algorithmen für CSG-Rendering, einen CSG-Baum in Normalform. Dabei berechnet er in einem temporären Tiefenpuffer die Sichtbarkeit eines kompletten partiellen Produkts zugleich. Dies erfolgt in drei Phasen; geschnittene und

[1] Eine aufwändige Tiefenschichtenermittlung wie für ordnungsunabhängige Transparenz (Abschnitt 3.3.2) ist hier nicht erforderlich, weil die Tiefenschichten nicht von vorne nach hinten sortiert sein müssen.

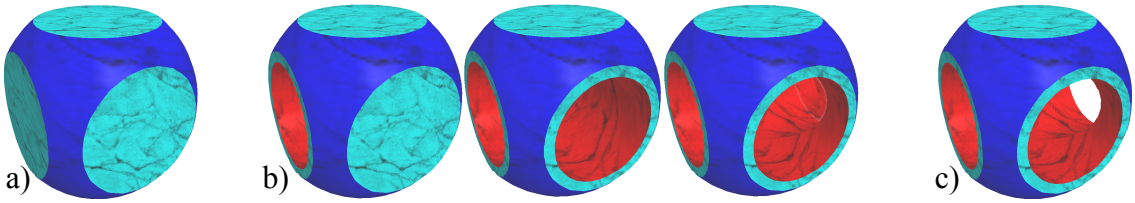

Abbildung 36: Die drei Phasen des SCS Algorithmus. a) Ermittlung der Vorderseite der geschnittenen Primitive. b) Sequenz von Subtraktionen. c) Der Schnitt mit den Rückseiten der geschnittenen Primitive liefert das Endergebnis.

subtrahierte Primitive in einem partiellen Produkt werden dabei getrennt voneinander in verschiedenen Phasen behandelt (Abbildung 36):

a. Zunächst wird die Vorderseite des Schnitts aller n geschnittenen Primitive im partiellen Produkt ermittelt. Hierzu werden zwei Prinzipien genutzt: Diese Vorderseite ist die hinterste aller Vorderseiten der geschnittenen Primitive, und außerdem ist sie nur dort sichtbar, wo sich hinter ihr n Rückseiten von allen Primitiven befinden. Diese Prinzipien beruhen darauf, dass nur konvexe Primitive zugelassen sind.

b. Nun werden alle subtrahierten Primitive mehrfach in einer bestimmten Abfolge vom bisher vorhandenen Tiefenpuffer subtrahiert. Subtraktion ist hierbei eine Operation, die dort, wo die Vorderflächen eines Primitivs P vor und die rückseitigen Flächen von P hinter dem momentanen Tiefenpuffer liegen, die Tiefenwerte der rückseitigen Flächen von P in den Tiefenpuffer schreibt. Insgesamt verschiebt eine Subtraktion also den Tiefenpuffer nach hinten. Die Abfolge der subtrahierten Primitive muss so gewählt werden, dass alle Abhängigkeiten subtrahierter Primitive, die die Sichtbarkeit beeinflussen, korrekt behandelt werden. Ohne Wissen über die räumliche Anordnung der Primitive erfordert dies eine Anzahl von Subtraktion, die quadratisch mit der Anzahl der subtrahierten Primitive wächst. Wenn die Tiefenkomplexität der subtrahierten Primitive k bekannt ist, kann eine kürzere Abfolge von $O(n \cdot k)$ Primitiven gewählt werden.

c. Zuletzt werden die Tiefenwerte mit den Rückseiten der geschnittenen Primitive verschnitten. Wo sich also eine Rückseite eines geschnittenen Primitivs vor dem Tiefenpuffer befindet, wird der Tiefenpuffer zurückgesetzt, um anzuzeigen, dass an dieser Stelle das partielle Produkt nicht sichtbar ist.

Nachdem die Tiefenwerte für ein partielles Produkt auf diese Weise bestimmt wurden, werden mit dem Sichtbarkeitstransfer die Tiefenwerte, ähnlich wie bei dem Algorithmus von Goldfeather, in den Haupt-Tiefenpuffer übertragen.

7.1.3 Einschränkungen

Eine bisher nicht gelöste Schwierigkeit aller genannten Algorithmen ist, dass CSG-Geometrie nur dann korrekt dargestellt wird, wenn sich alle beteiligten Primitive innerhalb des View-Frustums befinden. Das Problem ist, dass CSG geschlossene Primitive voraussetzt, wenn aber ein Primitiv die Near- oder Far-clipping Ebene schneidet und somit Teile des Primitivs abgeschnitten werden, es nicht mehr geschlossen gerendert wird. Damit schlagen Berechnungen wie die Bestimmung der Parität fehl und es ergeben sich Renderingfehler.

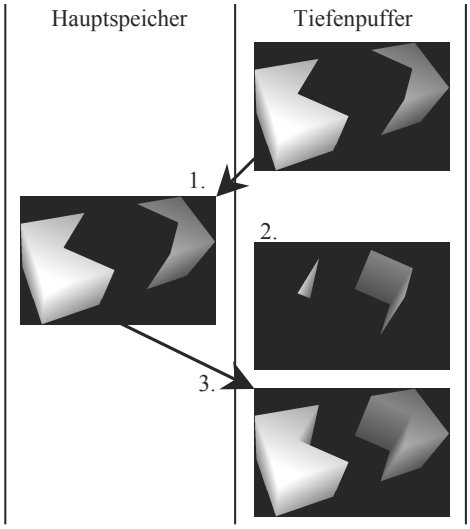

Abbildung 37: Sichtbarkeitstransfer durch Pixeltransfer-Operationen. Der bisherige Tiefenpuffer wird im Hauptspeicher zwischengespeichert, dann wird im Tiefenpuffer die nächste CSG-Berechnung vorgenommen. Die Ergebnisse werden mit den Tiefenwerten im Hauptspeicher verknüpft.

7.2 Optimierungen für bildbasiertes CSG

Bei der Entwicklung von Algorithmen für bildbasiertes CSG sind Geschwindigkeitsoptimierungen besonders wichtig. Dies liegt zum einen an der, je nach Tiefenkomplexität, potentiell quadratischen Laufzeit der Algorithmen, zum anderen an den hohen konstanten Kosten für typischerweise erforderliche Basistechniken. Dieser Abschnitt stellt Optimierungstechniken vor, die bildbasiertes CSG-Rendering in Echtzeit auch für komplexe CSG-Geometrie ermöglichen.

7.2.1 ID-Texturen zur Beschleunigung des Sichtbarkeitstransfers

Der Algorithmus von Goldfeather und der SCS-Algorithmus haben die folgende Gemeinsamkeit: Sie benötigen zwei Tiefenpuffer und verwenden einen Sichtbarkeitstransfer, um temporär berechnete Informationen, welche Primitive an welchen Stellen sichtbar sind, mit den zuvor bereits berechneten Ergebnissen zu kombinieren. OpenGL unterstützt aber zwei Tiefenpuffer nicht. Eine verbreitete Methode zur Emulation zweier Tiefenpuffer ist die Verwendung von Pixeltransfer-Operationen zum Zwischenspeichern des Tiefenpuffers im Hauptspeicher und zum späteren Wiedereinlesen (Abbildung 37). Diese Methode, bereits in der ersten bekannten Implementierung von bildbasiertem CSG für OpenGL von Wiegand [89] verwendet und später in vielen anderen Beschreibungen genutzt [78][79][80], hat zwei gravierende Nachteile:

- OpenGL garantiert nicht, dass die ursprünglichen Tiefenwerte mit den wiederhergestellten Tiefenwerten exakt übereinstimmen. Dies führt häufig zu Renderingartefakten [78].
- Die Tiefenwerte werden vom Speicher der Graphikhardware in den Hauptspeicher kopiert. Dieser Datenpfad erlaubt – verglichen mit Kopien innerhalb des Graphikspeichers – nur geringe Transferraten. In der Praxis wird hierdurch die Geschwindigkeit von bildbasiertem CSG nahezu ausschließlich limitiert.

Texturbasierter Sichtbarkeitstransfer durch Alphatexturen

Ein bildbasierter Sichtbarkeitstransfer unter Verwendung dynamischer Texturen vermeidet die Probleme von Pixeltransfer-Operationen zu diesem Zweck [44]. Es wird ein P-Buffer verwen-

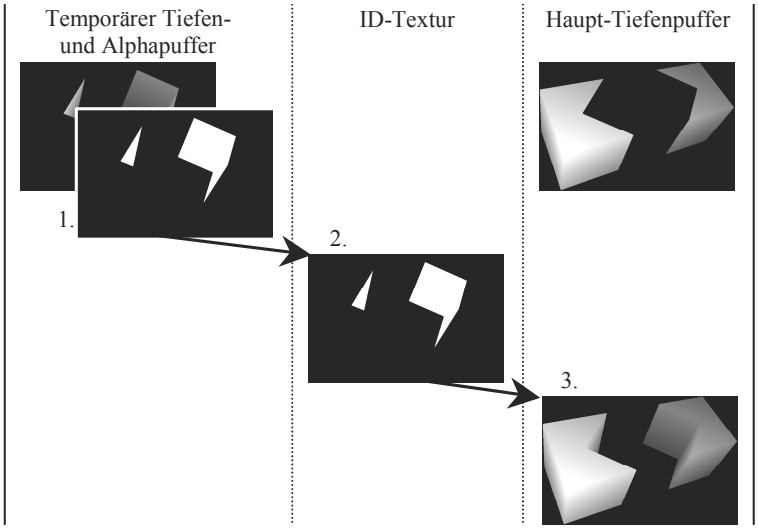

Abbildung 38: Sichtbarkeitstransfer durch ID-Texturen. Ein P-Buffer stellt einen temporären Tiefen- und Alphapuffer bereit, wo die CSG-Berechnung durchgeführt wird. Die berechnete Alphatextur wird zur Rekonstruktion der Tiefenwerte im Haupt-Tiefenpuffer verwendet.

det, um den temporären Tiefenpuffer bereitzustellen und darin die eigentliche CSG-Berechnung vorzunehmen. Die mit dem P-Buffer assoziierte dynamische Textur wird anschließend durch automatische Texturkoordinatengenerierung so auf den Haupt-Framebuffer projiziert, dass jedes Texel in der Textur genau dem Pixel im Framebuffer an der gleichen Stelle zugeordnet ist. Mit der Textur werden die Tiefenwerte des P-Buffers im Haupt-Framebuffer rekonstruiert (Abbildung 38).

Im Falle des SCS-Algorithmus wird dazu jedem Primitiv in einem partiellen Produkt eine eindeutige ID zugewiesen. Falls ein Primitiv an einer bestimmten Stelle sichtbar ist, wird dies im Alphapuffer des P-Buffers mit der ID angezeigt. Das heißt, immer wenn bei der CSG-Berechnung ein Primitiv in den Tiefenpuffer gerendert wird, wird gleichzeitig seine ID im Alphapuffer abgelegt. Die ID null markiert Stellen, an denen kein Primitiv sichtbar ist.

Nachdem die Berechnung des partiellen Produkts abgeschlossen ist, enthält der Alphapuffer des P-Buffers alle notwendigen Informationen, um den Sichtbarkeitstransfer durchzuführen. Dazu wird der P-Buffer als Alphatextur benutzt, die auf den Haupt-Framebuffer projiziert wird. Dabei werden mit einem „kleiner-als“ Tiefentest alle potentiell sichtbaren Flächen der Primitive im partiellen Produkt gerendert. Durch den Alphatest werden jedoch Fragmente nur geschrieben, falls die ID eines Primitivs mit der in der Textur gespeicherten ID übereinstimmt. Damit werden Tiefenwerte nur für die sichtbaren Gebiete eines Primitivs geschrieben, so dass schließlich der Tiefenpuffer die Tiefenwerte für alle bisher berechneten partiellen Produkte enthält.

Beim Algorithmus von Goldfeather wird die Sichtbarkeit jedes Primitivs einzeln berechnet. Daher kann bei diesem Algorithmus die Sichtbarkeit eines Primitivs durch zwei unterschiedliche Werte im Alphapuffer kodiert werden, nämlich null für ein nicht sichtbares und ein anderer Wert für ein sichtbares Primitiv. Der Sichtbarkeitstransfer verläuft genau wie beim SCS-Algorithmus.

Texturbasierter Sichtbarkeitstransfer durch RGBA-Texturen

Sichtbarkeitsinformationen können nicht nur im Alphapuffer, sondern auch in anderen Kanälen des Farbpuffers abgespeichert werden. Während des Sichtbarkeitstransfers ist es dann erforderlich, den Alphatest auf einen Farbwert eines Fragments anzuwenden. Dazu genügt es, in einem

Fragmentprogramm den Farbwert in den Alphakanal zu kopieren; auf älterer Hardware kann durch Berechnung des Skalarprodukts der Texturfarbe mit einer reinen Grundfarbe sowie Speichern des Ergebnisses in den Alphakanal diese Operation nachgebildet werden. Auf diese Weise kann mit einer RGBA-Textur beim Algorithmus von Goldfeather die Sichtbarkeit von vier Primitiven und beim SCS-Algorithmus von vier partiellen Produkten übertragen werden. Dadurch wird die Anzahl an benötigten Texturkopien, die Zahl der Umschaltvorgänge von P-Buffer zu Haupt-Framebuffer und damit die Zahl der Wechsel des Renderingkontextes minimiert und das Verfahren damit weiter beschleunigt.

Bewertung

Gegenüber der Verwendung von Pixeltransfer-Operationen für den Sichtbarkeitstransfer sind ID-Texturen um Größenordnungen schneller. Für den Algorithmus von Goldfeather kann beispielsweise eine Steigerung der Bilderzeugungsrate bis um das Achtfache festgestellt werden. Außerdem sind ID-Texturen nicht anfällig für Renderingartefakte, weil sie nur diskrete Informationen übertragen.

7.2.2 Verdeckungstests zum Ausnutzen der Tiefenkomplexität

Der geschichtete Goldfeather Algorithmus benötigt die Tiefenkomplexität k eines partiellen Produkts, denn diese repräsentiert die Anzahl der Tiefenschichten, die der Algorithmus behandeln muss. Ähnlich profitiert der SCS-Algorithmus von der Kenntnis der Tiefenkomplexität, da dadurch die Abfolge der subtrahierten Primitive von n^2 zu $n \cdot k$ Primitive verkürzt werden kann.

Zur Bestimmung der Tiefenkomplexität verwenden Stewart et al. [78] den Stencilpuffer. Die potentiell sichtbaren Teile der Primitive in einem partiellen Produkt werden ohne Tiefentest gerendert, wobei bei jedem erzeugten Fragment der Wert im Stencilpuffer inkrementiert wird. Schließlich repräsentiert der Maximalwert im Stencilpuffer die Tiefenkomplexität. Um den Wert auszulesen, wird der Stencilpuffer in den Hauptspeicher kopiert, wo das Maximum von der CPU bestimmt wird. Die Pixeltransfer-Operation zum Kopieren des Stencilpuffers ist dabei wegen der geringen Transfergeschwindigkeit auf diesem Datenpfad sehr langsam.

Allgemeine Berechnung der Tiefenkomplexität durch Verdeckungstests

Die Berechnung der Tiefenkomplexität k kann mit Verdeckungstests ohne Kopieren des Stencilpuffers in den Hauptspeicher erfolgen [44]. Ein allgemeines Verfahren hierfür ist, ein partielles Produkt Tiefenschicht für Tiefenschicht zu rendern und bei jeder Schicht die Anzahl der gerenderten Fragmente zu zählen. Die erste Schicht, bei der keine Fragmente gerendert werden, ist die $k+1$ Schicht. Dieses Verfahren lässt sich ohne großen Aufwand in den geschichteten Goldfeather Algorithmus integrieren, weil dieser ohnehin jede Tiefenschicht eines partiellen Produkts rendert. Es muss also nur die Abbruchbedingung des Algorithmus angepasst werden: Sobald eine Tiefenschicht keine Fragmente mehr enthält, wird der Algorithmus beendet.

Indirektes Ausnutzen der Tiefenkomplexität beim SCS-Algorithmus

Für den SCS-Algorithmus lassen sich Verdeckungstests in der Subtraktionsphase auf andere Weise integrieren [44]: Grundlage des Verfahrens ist, bei jeder Subtraktion eines Primitivs durch einen Verdeckungstest die Anzahl der sichtbaren Fragmente zu zählen und mit der Anzahl bei der letztmaligen Subtraktion des Primitivs zu vergleichen. Nur wenn die Anzahl unterschiedlich ist, wird der Tiefenpuffer nach hinten verschoben. Ansonsten ist eine Aktualisierung des Tiefenpuffers nicht nötig, da sich die Sichtbarkeit des Primitivs im Vergleich zu seiner letztmaligen Subtraktion nicht geändert hat.

Abweichend von der Beschreibung von Stewart et al. wird die untenstehende Abfolge subtrahierter Primitive P_1, ..., P_n gewählt:

$$S_n = \underbrace{P_1 P_2 ... P_n \quad . \; . \; . \; . \quad P_1 P_2 ... P_n}_{n-1 \text{ mal}} \; P_1$$

Diese Abfolge behandelt alle möglichen Abhängigkeiten subtrahierter Primitive korrekt und enthält die gleiche Anzahl Primitive wie die Abfolge von Stewart. Eine zusätzliche Eigenschaft der Abfolge ist, dass nach dem Rendern von n aufeinanderfolgenden Primitiven aus der Abfolge sichergestellt ist, dass n verschiedene Primitive gerendert wurden. Dies kann für eine verbesserte Abbruchbedingung der Subtraktionsphase genutzt werden: Wenn für n Subtraktionen hintereinander der Tiefenpuffer nicht aktualisiert werden muss, ist er für alle Primitive schon korrekt, und die Subtraktionsphase wird vorzeitig beendet.

Verdeckungstests in Verbindung mit dieser Abbruchbedingung ermöglichen für den SCS-Algorithmus eine Laufzeitkomplexität von $O(n \cdot k)$ gegenüber $O(n^2)$ beim Standard-SCS-Algorithmus, wobei k die Tiefenkomplexität der subtrahierten Primitive im partiellen Produkt ist. Der einzige Mehraufwand bei der Berechnung sind die zusätzlichen Kosten für die Verdeckungstests, die sich nach unseren Messungen aber nur geringfügig auf die Renderinggeschwindigkeit auswirken. Insbesondere beinhaltet das Verfahren keine zusätzlichen konstanten Kosten, im Gegensatz zur Ermittlung der Tiefenkomplexität durch Auslesen des Stencilpuffers.

7.2.3 Bounding-Box-basierte Optimierungen

Ein weiterer Ansatz zur Steigerung der Renderinggeschwindigkeit ist, die räumliche Anordnung der Primitive in einem partiellen Produkt auszunutzen, zum Beispiel um Gruppen von Primitiven zu identifizieren, die sich gegenseitig nicht beeinflussen [45]. Obwohl solche Optimierungen oft von recht einfacher Art sind, bewirken sie in vielen Fällen dennoch eine wesentliche Steigerung der Renderinggeschwindigkeit.

Primitivgruppen (primitive batches). Beim Algorithmus von Goldfeather wird gewöhnlich die Sichtbarkeit jedes Primitivs einzeln getestet. Aber Primitive in einem partiellen Produkt, die, bzw. deren Bounding-Boxes sich im Bildraum in Z-Richtung nicht überdecken, beeinflussen offensichtlich gegenseitig ihre Sichtbarkeit nicht. Daher können solche sich nicht überdeckende Primitive zu Primitivgruppen zusammengefasst werden. Im weiteren Verlauf des Algorithmus werden Primitivgruppen wie einzelne Primitive behandelt (Abbildung 39). Die Vorteile sind einerseits, dass so die Anzahl der erforderlichen Paritätsberechnungen vermindert wird, und zum anderen, dass weniger Sichtbarkeitstransfers nötig sind. Für den SCS-Algorithmus in allen Varianten ist diese Optimierung ebenfalls möglich, nicht aber für den geschichteten Goldfeather Algorithmus.

Disjunkte Bounding-Boxes (disjoint bounding volumes). Beim Algorithmus von Goldfeather ist die Berechnung der Parität eines subtrahierten Primitivs nur erforderlich, wenn seine Bounding-Box mindestens eine Bounding-Box eines Primitivs in der auf Sichtbarkeit getesteten Primitivgruppe schneidet. Andernfalls beeinflusst das subtrahierte Primitiv die Sichtbarkeit der Primitivgruppe ohnehin nicht.

CSG-Berechnung auf den Intersektionsbereich einschränken. Grundsätzlich können sichtbare Gebiete eines partiellen Produkts sich nur dort befinden, wo sich die geschnittenen Primitive eines partiellen Produkts im Bildraum überlappen. Daher können die Bounding-Boxes der geschnittenen Primitive in Pixelkoordinaten berechnet, die gemeinsam überdeckten Bereiche ermittelt und alle CSG-Berechnungen durch den Scissortest auf diesen Intersektionsbereich eingeschränkt werden. Diese Optimierung ist für sämtliche bildbasierten CSG-Algorithmen möglich.

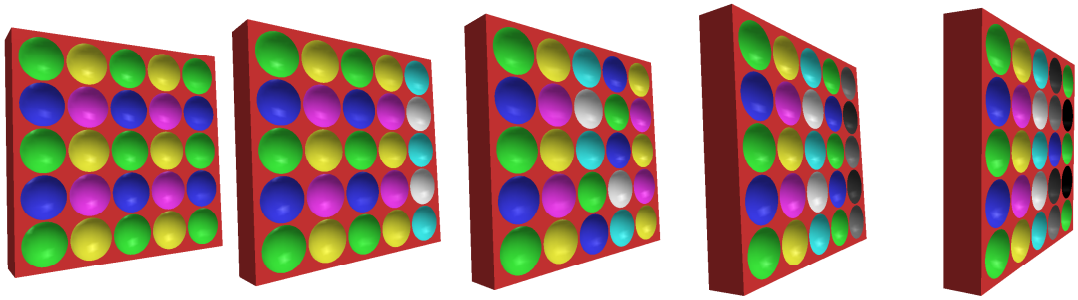

Abbildung 39: Optimierung durch Primitivgruppen. Primitive, deren Bounding-Boxes sich nicht überdecken, werden zusammengefasst und anschließend wie einzelne Primitive behandelt. Die Anzahl der Primitivgruppen, hier durch verschiedene Farbtöne dargestellt, variiert je nach Blickwinkel.

Paritätsberechnung auf Primitivgruppe einschränken. Beim Algorithmus von Goldfeather kann der Scissorbereich noch weiter eingeschränkt werden: Außerhalb der Bounding-Box der auf Sichtbarkeit getesteten Primitivgruppe ist die Berechnung der Parität nicht erforderlich, weil sich dort ohnehin keine potentiell sichtbaren Primitive befinden. Daher wird bei der Berechnung der Parität der Scissorbereich als Schnitt der Bounding-Box der Primitivgruppe mit dem vorher erläuterten Intersektionsbereich festgelegt.

7.3 Eine Renderingbibliothek für bildbasiertes CSG

CSG hat für die Modellierung von 3D-Geometrie eine Reihe von Vorzügen: Es ist intuitiv verständlich und die Resultate sind automatisch wohldefinierte Geometrie-Objekte. Dass CSG dennoch nur selten zur Modellierung verwendet wird, liegt an der fehlenden direkten Unterstützung für das Rendering in Echtzeit. Algorithmen für bildbasiertes Rendering von CSG können hier helfen, jedoch nur, wenn ein kompaktes API zur Kapselung der komplexen Algorithmen zur Verfügung steht. Diese Überlegung führte zur Idee der Entwicklung einer entsprechenden Renderingbibliothek, nämlich *OpenCSG*.

Bisher gab es nur eine einzige Bibliothek für bildbasiertes CSG, nämlich TGS SolidViz [86]. Dieses Produkt der Firma Mercury ist Teil des von ihr entwickelten Szenengraphsystems TGS OpenInventor 5.0. Anwendungen, die nicht auf Open Inventor basieren, können damit SolidViz nicht einsetzen. Der Quellcode von SolidViz ist nicht frei verfügbar, und die Möglichkeiten und Einschränkungen sind unzureichend dokumentiert.

7.3.1 Designziele von OpenCSG

Bildbasiertes CSG ist ein zu den meisten anderen Renderingeffekten orthogonales Verfahren, weil es die Tiefenwerte eines Bilds gesondert berechnet, auf die Farbwerte eines Bilds aber keinen direkten Einfluss nimmt. Bei den meisten anderen Renderingeffekten verhält es sich umgekehrt: Sie modifizieren den Farbpuffer, beeinflussen aber nicht den Tiefenpuffer. Damit ist bildbasiertes CSG parallel zu solchen anderen Renderingeffekten verwendbar. OpenCSG als Bibliothek für bildbasiertes CSG soll dies ebenfalls erlauben. Daher ist eine wichtige Anforderung für das Design ein

- **klar definierter Einsatzzweck.** Die Bibliothek soll ausschließlich für CSG-Rendering verwendet werden, das heißt die Beleuchtung und Schattierung von CSG-Geometrie nicht vornehmen.

Weiterhin wurde auf die folgenden Eigenschaften Wert gelegt:

- **Ein kompaktes Interface.** Das API soll möglichst übersichtlich und einfach sein.
- **Möglichst wenige externe Abhängigkeiten.** Eine prototypische Entwicklung verschiedener CSG-Algorithmen geschah zunächst auf Basis von VRS. Für die praktische Nutzung einer CSG-Renderingbibliothek aus Anwendungen ist eine solche Abhängigkeit jedoch nachteilig, denn so würde eine partikulare Funktionalität wie CSG-Rendering die Architektur der gesamten Graphikanwendung festlegen. Insbesondere in bereits bestehende, nicht auf VRS basierende Anwendungen könnte so CSG-Rendering kaum integriert werden.
- **Nutzung von anwendungsdefinierten Geometrie-Objekten.** Die meisten graphischen Anwendungen definieren eigene Geometrie-Objekte, die oft gültige CSG-Primitive sind. Diese Primitive sollten direkt von der Bibliothek genutzt werden können.
- **Sonstige allgemeine Eigenschaften.** Bei der Entwicklung der Bibliothek wurde weiterhin auf größtmögliche Portabilität, Schnelligkeit und Stabilität geachtet.

OpenCSG hängt lediglich von OpenGL ab und ist damit von allen auf OpenGL basierenden Anwendungen nutzbar. Es ist in C++ – und nicht in C – implementiert, da mit C++ durch Verwendung einer abstrakten Basisklasse für CSG-Primitive die Nutzung anwendungsdefinierter Primitive besonders einfach möglich ist.

7.3.2 Die Programmierschnittstelle von OpenCSG

Beim Design der öffentlichen Programmierschnittstelle von OpenCSG ist großen Wert auf Kompaktheit gelegt worden (Abbildung 40).

Abstrakte Basisklasse für CSG-Primitive

Das API stellt eine abstrakte Basisklasse `OpenCSG::Primitive` bereit, die die Eigenschaften eines CSG-Primitivs kapselt. Zu diesen Eigenschaften gehört, ob das Primitiv mit einem partiellen Produkt geschnitten oder von ihm subtrahiert wird; außerdem die Konvexität des Primitivs und die Bounding-Box des Primitivs im kanonischen Sichtvolumen. Die Basisklasse enthält außerdem eine abstrakte Methode zum Rendern eines Primitivs mit OpenGL, die von abgeleiteten Klassen implementiert wird. Auf diese Weise hängt OpenCSG nicht von Hilfsbibliotheken ab, die die Funktionalität zum Rendern einzelner CSG-Primitive enthalten. Stattdessen setzt OpenCSG darauf, dass in Anwendungen, die OpenCSG verwenden, die Funktionalität zum Rendern von CSG-Primitiven bereits vorhanden ist und von den Algorithmen für CSG-Rendering verwendet werden kann. Als Einschränkung darf in einer entsprechenden Renderingmethode einer abgeleiteten Klassen die primäre Vertex-Farbe nicht verändert werden, weil sie intern zur Darstellung der IDs für den Sichtbarkeitstransfer benötigt wird. Die Implementierung einer solchen Renderingmethode aus einer bestehenden Anwendung heraus ist in der Regel trivial.

Funktion für CSG-Rendering

Das OpenCSG-API enthält weiterhin eine zentrale Funktion, die das zu behandelnde partielle Produkt in Form einer Liste von CSG-Primitiven als Parameter hat und die eigentliche Funktionalität des CSG-Renderings implementiert. OpenCSG stellt sechs verschiedene CSG-Algorithmen zur Verfügung; die Auswahl der Algorithmen erfolgt dabei über zwei weitere Parameter dieser Funktion. Zur Auswahl stehen der Algorithmus von Goldfeather und der SCS-Algorithmus, jeweils in einer Variante ohne Ermittlung der Tiefenkomplexität oder mit Ermittlung der Tiefenkomplexität mit dem Stencilpuffer bzw. durch hardwareunterstützte Verdeckungstests.

Funktionen zur Überführung eines allgemeinen CSG-Baums in Normalform enthält OpenCSG nicht. Hierauf wurde aus zwei Gründen verzichtet: 1) Es wären weitere Klassen und Verwal-

```
namespace OpenCSG {
    enum Operation { Intersection, Subtraction };

    class Primitive {
    public:
        Primitive(Operation, unsigned int convexity);
        virtual ~Primitive();
        void setOperation(Operation);
        Operation getOperation() const;
        void setConvexity(unsigned int);
        unsigned int getConvexity() const;
        void setBoundingBox(float  minx, float  miny, float  minz,
                            float  maxx, float  maxy, float  maxz);
        void getBoundingBox(float& minx, float& miny, float& minz,
                            float& maxx, float& maxy, float& maxz) const;
        virtual void render() = 0;
    };

    enum Algorithm {
        Automatic, Goldfeather, SCS
    };
    enum DepthComplexityAlgorithm {
        NoDepthComplexitySampling, OcclusionQuery, DepthComplexitySampling
    };

    void render(const std::vector<Primitive*>& primitives,
                Algorithm = Automatic,
                DepthComplexityAlgorithm = NoDepthComplexitySampling);
}
```

Abbildung 40: Die Programmierschnittstelle von OpenCSG.

tungsmethoden für Vereinigung, Intersektion und Subtraktion zweier CSG-Bäume notwendig. Hierdurch würde die Schnittstelle also deutlich erweitert werden müssen, was ihrer Übersichtlichkeit entgegenstehen würde. 2) In vielen Fällen kann eine Anwendung durch anwendungsspezifisches Wissen einen kleineren CSG-Baum in Normalform erzeugen, als dies durch eine allgemeine Funktion in einer CSG-Bibliothek möglich wäre.

7.3.3 Die Ausgabe von OpenCSG

Die Funktion für CSG-Rendering berechnet im Tiefenpuffer die Tiefenwerte für das partielle Produkt. Dabei wird der Tiefenpuffer vor dem Aufruf der Funktion mit einem „kleiner-als“ Tiefentest berücksichtigt, wodurch eine korrekte Verdeckungsermittlung geschieht. Die Auswahl des CSG-Algorithmus spielt für das Ergebnis keine Rolle. Alle in Abschnitt 7.2 beschriebenen Optimierungen werden, wo sie anwendbar sind, genutzt.

Minimierung von Seiteneffekten

Abgesehen von der Modifikation des Tiefenpuffers ist die Funktion für CSG-Rendering bestrebt, keine sonstigen Seiteneffekte zu erzeugen. Insbesondere wird der Farbpuffer nicht verändert. Die Schattierung der CSG-Primitive liegt in der Verantwortung der aufrufenden Anwendung, die nach dem Aufruf der Funktion alle Primitive im partiellen Produkt mit einem Tiefentest auf Gleichheit in den Farbpuffer rendern muss. Auf diese Art und Weise wird sichergestellt, dass OpenCSG die Beleuchtung und Schattierung der CSG-Geometrie nicht selbst vornimmt.

Eine Veränderung des Stencilpuffers wird von der Funktion für CSG-Rendering ebenfalls soweit wie möglich verhindert. Dies ist allerdings derzeit technisch nur für partielle Produkte, die ausschließlich konvexe Primitive enthalten, möglich. Bei konkaven Primitiven müssen nämlich während des Sichtbarkeitstransfers die verschiedenen Tiefenschichten eines solchen Primitivs getrennt voneinander rasterisiert werden, wozu der Stencilpuffer erforderlich ist. Das bedeutet, dass, wenn ein partielles Produkt konkave Primitive enthält, der Inhalt des Stencilpuffers nach dem Aufruf der CSG-Renderingfunktion nicht definiert ist. Zur Sicherung des Stencilpuffers müsste er zu Beginn des Aufrufs der CSG-Renderingfunktion zwischengespeichert und danach

wiederhergestellt werden. Dazu wäre die Buffer-Region-Erweiterung geeignet, jedoch wird diese Erweiterung nicht von jeder Hardware unterstützt und ist damit nicht portabel.

Berücksichtigung von Einstellungen des Renderingkontextes

Die Funktion für CSG-Rendering berücksichtigt einige Einstellungen des Renderingkontextes für die CSG-Berechnung, andere Einstellungen hingegen, die intern benötigt werden oder für CSG-Geometrie nicht sinnvoll definiert sind, werden ignoriert. Insbesondere sind alle Einstellungen, die die Schattierung und Beleuchtung betreffen, für OpenCSG bedeutungslos und werden ignoriert. Berücksichtigt werden die geometrischen Transformationen der CSG-Primitive, die Einstellung des Scissortests und, falls sich ausschließlich konvexe Primitive im partiellen Produkt befinden, der Stenciltest. Aus technischen Gründen ignoriert wird der Alphatest, der beim Sichtbarkeitstransfer intern benötigt wird, und Front- bzw. Back-face Culling, die intern zur Ermittlung der potentiell sichtbaren Flächen erforderlich sind, aber ohnehin für CSG-Rendering nicht klar definiert sind. Des weiteren wird als Tiefentest immer eine „kleiner-als“ Funktion gewählt und damit die Voreinstellung des Renderingkontextes ebenfalls ignoriert. Für den Tiefentest käme grundsätzlich auch die Funktion „größer-als“ in Frage, doch einerseits wäre sie in der Praxis vermutlich nicht nützlich, und andererseits würde sie große Änderungen an den internen CSG-Algorithmen erfordern.

7.3.4 Einfaches Anwendungsbeispiel

Die Implementierung einer prototypischen Anwendung, die OpenCSG beispielhaft verwendet, kann wie folgt geschehen: CSG-Primitive werden durch eine abgeleitete Klasse implementiert, welche die ID einer Display-Liste kapselt und diese Display-Liste in der Renderingmethode aufruft. Display-Listen für CSG-Primitive wie zum Beispiel Kugeln, Zylinder, Tori oder Würfel werden mit den GLU bzw. GLUT-Bibliotheken erzeugt. Die Darstellung von einfachen CSG-Modellen erfordert so nur wenige Zeilen Code (Abbildung 41).

7.4 Bildbasiertes CSG in Szenengraphsystemen

7.4.1 Deklaration von CSG-Geometrie

Zur Deklaration von CSG-Geometrie in einer Szenenbeschreibung können verschiedene Ansätze verwendet werden: Zum einen ist die Deklaration von partiellen Produkten durch abstrakte Attribute möglich. Zum anderen kann ein CSG-Baum durch hierarchische Szenenknoten mit jeweils genau zwei Kindern deklariert werden.

Deklaration durch abstrakte Attribute

Zur Deklaration von partiellen Produkten durch Attribute sind zwei abstrakte Monoattribute erforderlich (Abbildung 42a): Das Attribut `CSGPartialProduct` deklariert ein neues partielles Produkt, das heißt, jedes Geometrie-Objekt, das sich im Szenengraph unterhalb dieses Attributs befindet, wird als jeweils ein CSG-Primitiv des partiellen Produkts interpretiert. Das Attribut `CSGComplement` gibt an, ob ein Geometrie-Objekt eines partiellen Produkts geschnitten oder subtrahiert wird, und enthält zusätzlich die Konvexität eines Geometrie-Objekts. Jedes Geometrie-Objekt in der Szenenbeschreibung, das auf diese Weise als CSG-Primitiv genutzt wird, muss natürlich ein gültiges Primitiv, also insbesondere geschlossen sein. Hierzu ist es unter Umständen nötig, verschiedene Geometrie-Objekte zu einem zusammenzufassen, wie zum Beispiel ein offener Zylinder und zwei Kreisscheiben zum Abschließen des Zylinders.

```
class DLPrim : public OpenCSG::Primitive {
public:
  DLPrim(unsigned int displayListId,
         OpenCSG::Operation operation,
         unsigned int convexity)
    : OpenCSG::Primitive(operation, convexity),
      id(displayListId) { }

  virtual void render() { glCallList(id); }

private:
  unsigned int id;
};

// Create display list for box and sphere
GLuint id1 = glGenLists(1);
glNewList(id1, GL_COMPILE);
glutSolidCube(1.8);
glEndList();
GLuint id2 = glGenLists(1);
glNewList(id2, GL_COMPILE);
glutSolidSphere(1.2, 20, 20);
glEndList();

// Create partial product with CSG primitives
namespace OpenCSG;
DLPrim* box=new DLPrim(id1, Intersection, 1);
DLPrim* sphere=new DLPrim(id2,Subtraction, 1);
std::vector<Primitive*> primitives;
primitives.push_back(box);
primitives.push_back(sphere);
```

```
// Initializes depth values of partial product
OpenCSG::render(primitives,
                Goldfeather,
                NoDepthComplexitySampling);

// Render cube and sphere again with z-equal
glDepthFunc(GL_EQUAL);
// Also setup lighting and shading ...
box->render();
sphere->render();

glDepthFunc(GL_LESS);
```

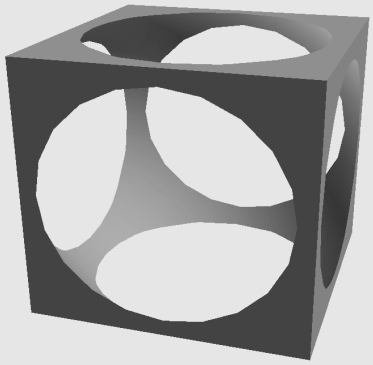

Abbildung 41: Quellcode zum Rendern eines einfachen CSG-Modells mit OpenCSG: Eine Kugel wird von einem Würfel subtrahiert.

Deklaration durch hierarchische Knoten

Eine Deklaration eines allgemeinen CSG-Baums durch abstrakte Attribute ist nicht praktikabel. Weil ein CSG-Baum eine hierarchische Datenstruktur darstellt, sind stattdessen hierarchische Szenenknoten geeignet, einen CSG-Baum zu deklarieren. Hierzu sind drei verschiedene Klassen hierarchischer *CSG-Szenenknoten* für Intersektion (`CSGIntersection`), Subtraktion (`CSGSubtraction`) und Vereinigung (`CSGUnion`) erforderlich (Abbildung 42b). Diese Szenenknoten, von der abstrakten Basisklasse `CSGOperation` abgeleitet, haben jeweils zwei Kindknoten. Jeder von diesen kann entweder ein weiterer CSG-Szenenknoten oder ein *CSG-Blattknoten* (`CSGPrimitive`) sein. Letztgenannter enthält als Kindknoten ein CSG-Primitiv als Menge von Geometrie-Objekten, sowie zugehörige Transformationen und sonstige konkrete Attribute. Des weiteren enthält ein CSG-Blattknoten die Konvexität des Primitivs als Instanzvariable.

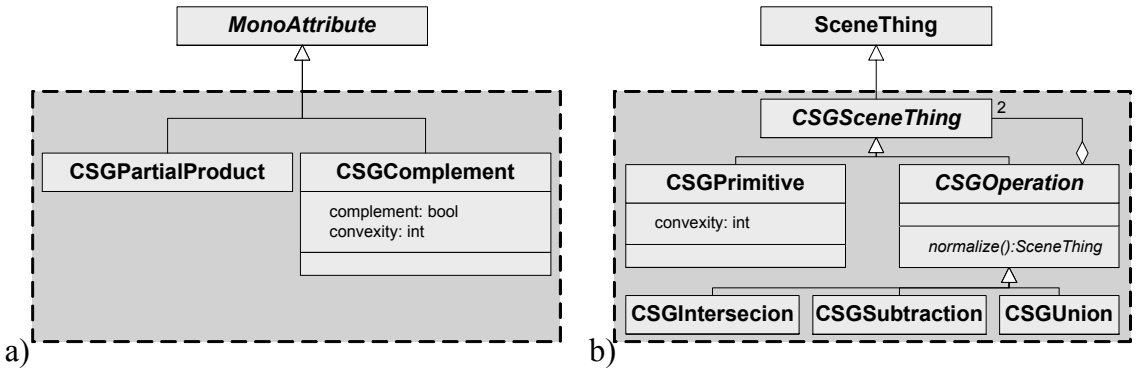

Abbildung 42: Klassendiagramm mit den Klassen zur Deklaration von CSG durch abstrakte Attribute (a) und hierarchische Szenenknoten (b).

Bewertung

In Abbildung 43 ist ein Beispiel zur Deklaration eines CSG-Modells mit beiden Möglichkeiten illustriert. Mit der Deklaration eines allgemeinen CSG-Baums durch hierarchische Szenenknoten wird die Umwandlung des Baums in Normalform durch das Szenengraphsystem durchgeführt. Dies ist ein Vorteil gegenüber der Deklaration durch abstrakte Attribute, bei der die CSG-Geometrie bereits normalisiert angegeben werden muss. Auf der anderen Seite erweist sich bei manchen Anwendungen die Deklaration von partiellen Produkten durch Attribute als flexibler, zum Beispiel für den Fall, dass eine Anwendung zwischen einer CSG- und einer nicht-CSG-Darstellung umschalten soll: Ist ein partielles Produkt durch Attribute deklariert, können diese Attribute einfach entfernt beziehungsweise ausgefiltert werden, um eine Darstellung der CSG-Primitive ohne CSG-Rendering zu erhalten. Bei Deklaration eines CSG-Baums durch CSG-Szenenknoten müssen dagegen diese Szenenknoten durch gewöhnliche hierarchische Szenenknoten ersetzt werden, was ein Umkopieren des Szenengraphen erfordert.

7.4.2 Auswertung von CSG

Die Darstellung von CSG-Geometrie, die durch hierarchische CSG-Szenenknoten deklariert ist, kann auf die Auswertung abstrakter Attribute für partielle Produkte zurückgeführt werden. Die Klasse `CSGOperation` enthält dazu eine Methode zur Normalisierung und Konvertierung: Zunächst wendet sie die Normalisierungsregeln für CSG-Bäume an, um einen CSG-Baum in Normalform zu erhalten. Anschließend wandelt sie diesen CSG-Baum, der zunächst noch durch CSG-Szenenknoten repräsentiert ist, durch eine lineare Traversierung in einen gewöhnlichen Szenengraphen um, bei dem Attribute die einzelnen partiellen Produkte des CSG-Baums in Normalform beschreiben. Dieser Szenengraph wird dann wie im Folgenden beschrieben ausgewertet.

Die Auswertung von Attributen zur Deklaration von partiellen Produkten kann mit lokalen Renderingtechniken (Abschnitt 6.4) erfolgen. Das Erreichen eines `CSGPartialProduct`-Attributs bewirkt den Start der `CSGShader`-Renderingtechnik, die sich intern für das CSG-Rendering auf OpenCSG abstützt.

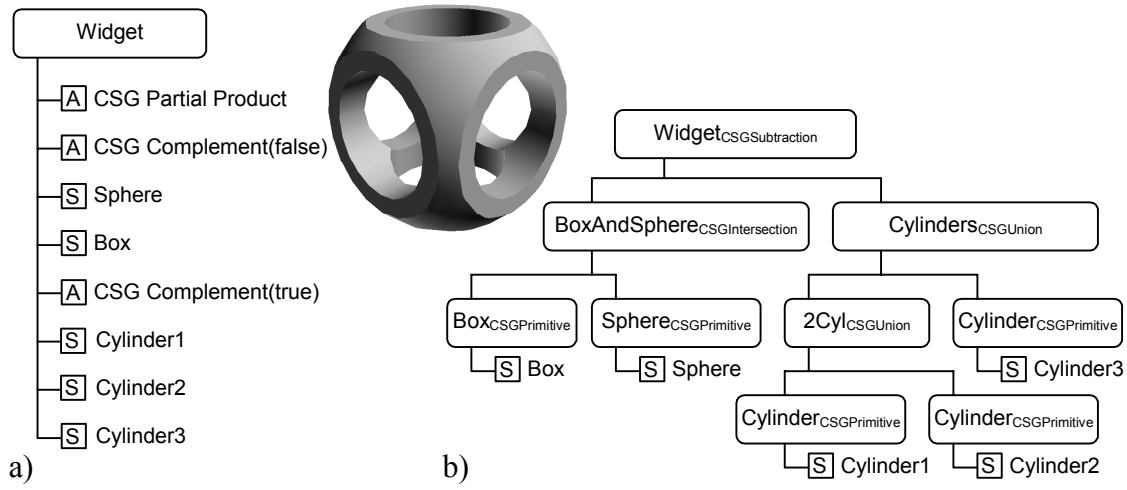

Abbildung 43: Deklaration eines CSG-Modells in einer Szenenbeschreibung durch abstrakte Attribute (a) und durch hierarchische Szenenknoten (b).

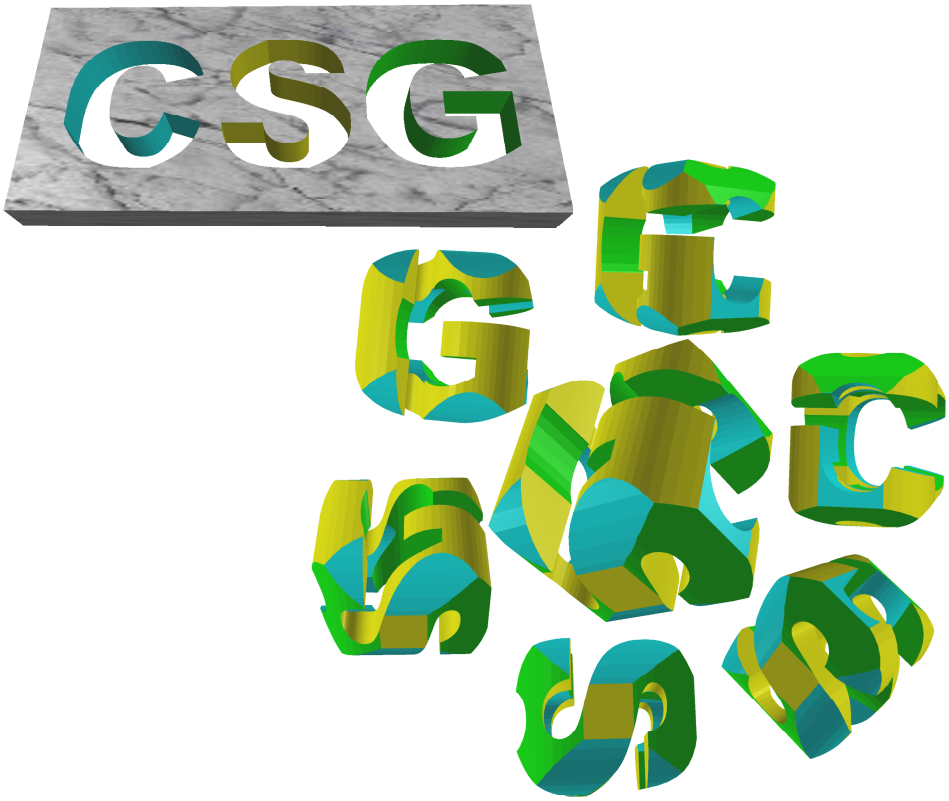

Abbildung 44: Durch CSG kann Geometrie modelliert werden, die exakt durch alle Öffnungen einer Schablone passt. Dazu müssen die extrudierten Geometrie-Objekte rotiert und miteinander geschnitten werden.

Der `CSGShader` bewirkt zwei Durchläufe des Teilgraphen. Der erste Durchlauf ist ein Analysedurchlauf, in dem für jedes auftretende Geometrie-Objekt ein von `OpenCSG::Primitive` abgeleitetes `VRSPrimitive`-Objekt erzeugt wird, mit welchem nach dem Durchlauf das Geometrie-Objekt durch OpenCSG gerendert werden kann. Dabei werden im `VRSPrimitive` die folgenden Informationen abgelegt: Das Geometrie-Objekt selber, seine geometrische Transformation, den verwendeten Evaluationskontext, je nach Geometrie-Objekt weitere für das Rendering benötigte konkrete Attribute, sowie die Konvexität, die Bounding-Box, und ob das Objekt geschnitten oder subtrahiert wird. Nachdem dieser Durchlauf beendet ist, erfolgt der Aufruf der Renderingfunktion von OpenCSG mit allen erzeugten `VRSPrimitive`-Objekten. Danach sind die Tiefenwerte des partiellen Produkts initialisiert, und im zweiten Durchlauf des Teilgraphen beleuchtet und schattiert der `CSGShader` die CSG-Geometrie.

Abbildung 44 zeigt ein Beispiel für VRS-Geometrie, nämlich 3D-Modelle von Buchstaben, die rotiert und durch CSG miteinander geschnitten werden. Die resultierende Geometrie passt, wie bei einem Kinderspielzeug, durch eine Schablone mit entsprechenden Löchern für die Buchstaben. In Abbildung 45 ist eine mit VRS entwickelte prototypische Anwendung abgebildet, mit der die Trassierung einer Straße in einem frühen Planungsstadium visualisiert werden kann. Durch CSG-Operationen werden dabei erforderliche Abgrabungen und Aufschüttungen für die Böschung der Straße deutlich.

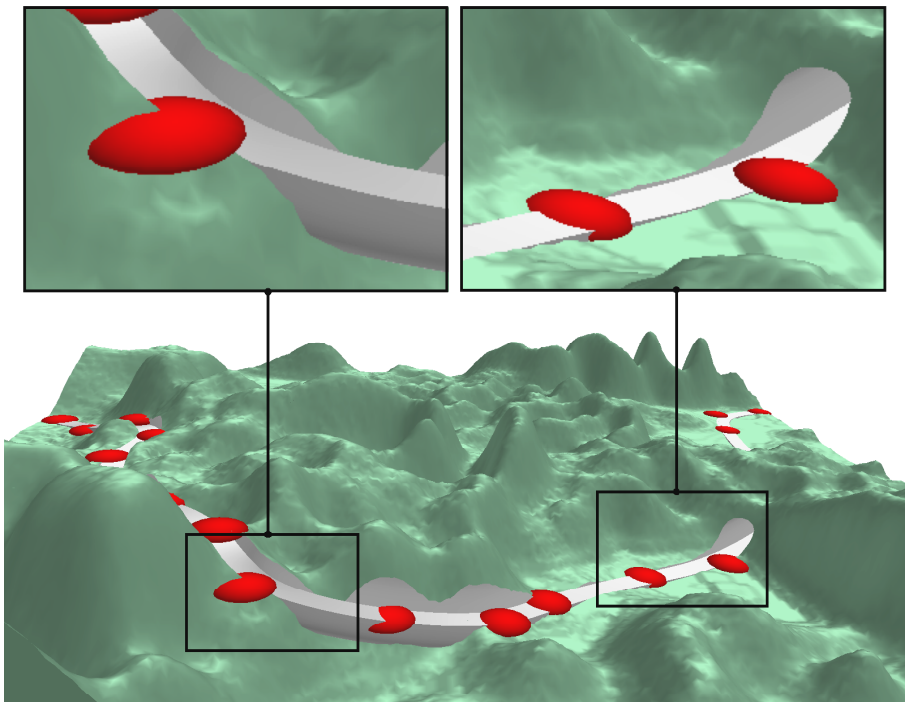

Abbildung 45: Interaktive Modellierung der Trassierung einer Straße durch CSG-Operationen mit Darstellung von Abgrabungen, Aufschüttungen und Tunnel.

7.4.3 Kombination von CSG mit Beleuchtung erster Ordnung

Von Jansen und van der Zalm stammt die bisher einzige Veröffentlichung, die sich explizit mit der Darstellung von Schatten von CSG-Modellen befasst, und zwar mit der Konstruktion von Schattenvolumenpolygonen für CSG-Geometrie [40]. Dieses Verfahren ist also für objektbasiertes Rendering von CSG-Modellen mit Schattenvolumen entwickelt worden und mit bildbasiertem CSG-Rendering nicht kombinierbar. Damit gab es bisher kein praktisches Beispiel für den Einsatz von Schatten oder Reflexionen zusammen mit bildbasiertem CSG. Gerade zur CSG-Modellierung ist aber die Darstellung von Schatten oft wichtig zum Erfassen der Form der CSG-Geometrie (Abbildung 46). Es ist daher davon auszugehen, dass die Verfahren bislang wegen der Komplexität der Einzelalgorithmen und der zusätzlichen Konflikte bei ihrer gleichzeitigen Anwendung nicht kombiniert wurden.

Mit dem bisher beschriebenen Stand der Szenengraphauswertung ist die Kombination von CSG mit Beleuchtung erster Ordnung leicht realisierbar: Der `CSGShader` ist ein Beispiel für eine lokale Renderingtechnik, die nicht nur in Vollschattierungs- und Einzellichtdurchläufen, sondern auch in Geometriedurchläufen durchgeführt wird. Auf diese Weise wird die CSG-Berechnung automatisch auch in globalen Renderingtechniken für Beleuchtung erster Ordnung verwendet, also in Renderingdurchläufen zur Berechnung von Shadow-Maps zur Schattenberechnung und in Renderingdurchläufen zur Darstellung reflektierter Kameraeinstellungen. Nahezu ohne weitere Vorkehrungen kann so CSG-Geometrie mit Schatten oder in einem Spiegel gerendert werden.

Zusammenarbeit mit einzelnen Renderingtechniken

Im einzelnen sind alle Renderingtechniken für Schatten, die eine Variante von Shadow-Mapping implementieren, für CSG-Geometrie nutzbar. Für Lichtquellen in unendlicher Entfer-

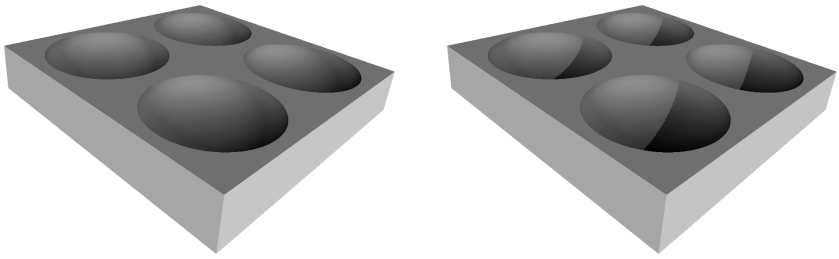

Abbildung 46: Nur mit Schatten ist erkennbar, dass die Kugeln subtrahiert werden.

nung ist die Kombination der Verfahren unproblematisch. Hingegen ist die Behandlung von Schatten einer Punktlichtquelle, die von CSG-Geometrie umgeben ist, mit ähnlichen Problemen wie bei gewöhnlicher Geometrie verbunden (Abschnitt 3.4.2): In vielen Fällen befindet sich die Punktlichtquelle innerhalb eines subtrahierten CSG-Primitivs. Dadurch kann das Koordinatensystem zur Berechnung der Shadow-Map nicht so gewählt werden, dass das View-Frustum der Lichtquelle das CSG-Primitiv enthält. Wie in Abschnitt 7.1.3 beschrieben, treten dann Fehler bei der CSG-Berechnung auf und der Schatten erscheint letztlich fehlerhaft.

Die Renderingtechnik für texturbasierte, planare Reflexionen funktioniert ohne Einschränkungen mit gespiegelter CSG-Geometrie. Die Technik für stencilbasierte Reflexionen ist ebenfalls nutzbar, wenn das gespiegelte partielle Produkt nur konvexe Primitive enthält. Befinden sich andernfalls auch konkave Primitive in einem gespiegelten partiellen Produkt, zerstört OpenCSG bei der CSG-Berechnung den Inhalt des Stencilpuffers. In diesem Fall gibt es also einen Ressourcenkonflikt mit der stencilbasierten Reflexionstechnik, so dass die Reflexion nicht korrekt gerendert wird. Um den Ressourcenkonflikt zu umgehen, muss der Stencilpuffer vor dem Aufruf der Renderingfunktion von OpenCSG zwischengespeichert und danach wiederhergestellt werden, was effizient mit der nichtportablen Buffer-Region-Erweiterung möglich ist.

Die Renderingtechnik für dynamisches Environment-Mapping ist ebenfalls zur gespiegelten und auch zur spiegelnden Darstellung von CSG-Geometrie nutzbar. Beim Durchlauf zum Rendern in eine Seite der Cubemap kann allerdings, wie bei Shadow-Mapping für Punktlichtquellen, das Problem auftreten, dass die CSG-Geometrie nicht vollständig im View-Frustum der Kamera zum Rendern in eine Seite der Cubemap liegt, und daher das CSG-Rendering fehlschlägt. Dieses Problem kann für manche Szenen nicht umgangen werden, zum Beispiel, wenn ein reflektierendes Objekt von einer größeren CSG-Geometrie umgeben ist.

Schattenvolumen lassen sich, wie in Abschnitt 3.8.2 bemerkt, mit bildbasiertem CSG-Rendering nicht kombinieren. Hier handelt es sich um eine grundsätzliche Inkompatibilität.

7.4.4 Ansätze zur Kombination von CSG mit Oberflächenschattierungen

Als lokale Renderingtechnik ist der `CSGShader` nicht gleichzeitig mit anderen lokalen Renderingtechniken zur Oberflächenschattierung nutzbar, denn es ist ja für ein Geometrie-Objekt in der Szenenbeschreibung immer nur eine lokale Renderingtechnik verantwortlich. Somit kann ohne Vorkehrungen CSG-Geometrie nicht mit einer Oberflächenschattierung dargestellt werden. Wir haben mit zwei Ansätzen experimentiert, die diese Kombination von Renderingeffekten ermöglichen.

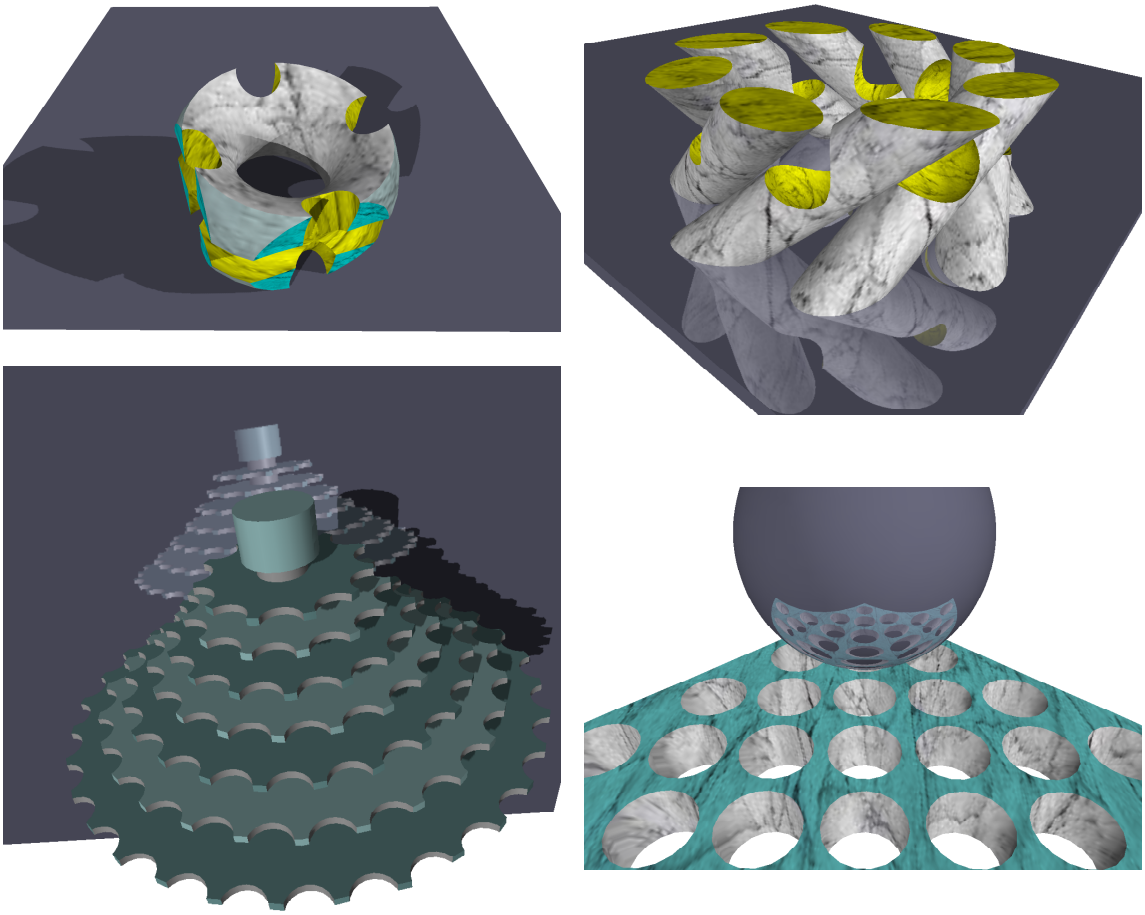

Abbildung 47: CSG-Geometrie mit verschiedenen Beleuchtungseffekten erster Ordnung.

Auswertung von CSG durch globale Renderingtechniken

Der erste Ansatz ist, zur Auswertung der CSG-Attribute eine globale – und nicht eine lokale – Renderingtechnik zu verwenden. Eine entsprechende Technik identifiziert vor dem Haupt-Renderingdurchlauf in einem separaten Durchlauf alle CSG-Attribute in der Szenenbeschreibung und initialisiert am Ende des Durchlaufs den Tiefenpuffer mit den Tiefenwerten der CSG-Geometrie und der sonstigen Geometrie in der Szene. Im Haupt-Renderingdurchlauf wird die Szene mit Tiefentest auf Gleichheit gerendert. Da in diesem Durchlauf lokale Renderingtechniken ausgewertet werden, werden Oberflächenschattierungen wie gewünscht sichtbar.

Jedoch ist diese Vorgehensweise zusammen mit globalen Renderingtechniken zur Beleuchtung erster Ordnung praktisch nicht verwendbar. In den Renderingdurchläufen der Schatten- bzw. Reflexionstechniken müsste explizit ebenfalls die CSG-Berechnung durchgeführt werden. In einer praktischen Implementierung hat sich beispielsweise für Schattenverfahren gezeigt, dass die resultierenden Techniken sehr stark aufeinander abgestimmt werden mussten, die Implementierung schließlich sehr komplex wurde und dadurch nicht mehr wartbar war.

Referenzierung einer weiteren lokalen Renderingtechnik

Der zweite Ansatz ist, dass der `CSGShader` eine andere lokale Renderingtechnik referenzieren kann. Ist diese vorhanden, wird der zweite Renderingdurchlauf vom `CSGShader` durch die Durchläufe dieser Technik ersetzt. Dadurch wird zur Schattierung der CSG-Geometrie die Oberflächenschattierung benutzt, die durch die lokale Renderingtechnik implementiert wird.

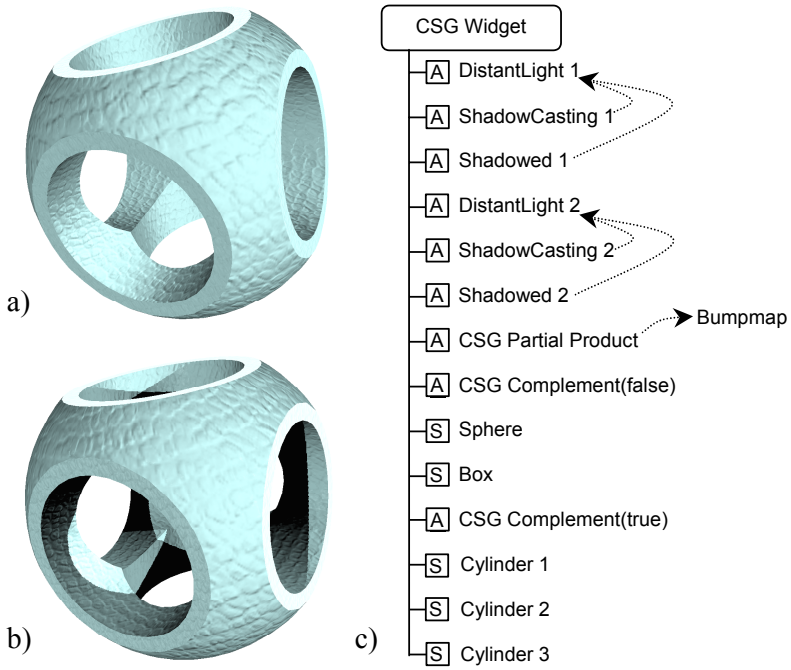

Abbildung 48: CSG-Geometrie mit Bumpmapping (a). Zusätzlich mit zwei Schatten (b). Zugehöriger Szenengraph (c).

Dieser Ansatz erlaubt es, Beleuchtung erster Ordnung auf CSG-Geometrie mit entsprechender Oberflächenschattierung zu nutzen. Abbildung 48b zeigt das Beispiel einer CSG-Geometrie mit Schatten und Bumpmapping, in Abbildung 48c ist der entsprechende Szenengraph abgebildet. Ein Nachteil des Ansatzes ist, dass unterschiedliche Oberflächenschattierungen für die Geometrie in einem partiellen Produkt nicht verwendet werden können.

7.5 Transparentes Rendering von CSG

Bei der Darstellung von CSG-Geometrie können Schatten und Reflexionen in vielen Fällen helfen, den Aufbau eines CSG-Modells nachzuvollziehen. Transparenz ist ein weiterer Renderingeffekt, der hierbei helfen kann, denn durch Transparenz können innere, normalerweise verdeckte Teile des Modells sichtbar werden. Allerdings sind Renderingverfahren zur Darstellung von Transparenz nicht ohne weiteres mit bildbasiertem CSG-Rendering kombinierbar, und daher kann die Transparenztechnik in VRS nicht durch einfache Anpassungen mit dem `CSGShader` für bildbasiertes CSG in einer Szenengraphauswertung kombiniert werden. Stattdessen ist die Entwicklung eines eigenen Renderingverfahrens für bildbasiertes, transparentes CSG-Rendering nötig. Ein solches Verfahren wird in diesem Abschnitt vorgestellt [46].

Überblick

Das Verfahren basiert zum einen auf dem Algorithmus zur ordnungsunabhängigen Transparenz (Abschnitt 3.3.2) und zum anderen auf dem Algorithmus von Goldfeather (Abschnitt 7.1.1). Das Ziel ist also eine Tiefenschichtenermittlung unter Berücksichtigung bildbasierter CSG-Berechnungen. Die Ermittlung der ersten, vordersten Tiefenschicht wird dabei durch den gewöhnlichen Algorithmus von Goldfeather erreicht. Die Berechnung einer inneren, n+1 Tiefenschicht erfordert, dass die vorherige n-te Tiefenschicht bereits ermittelt worden ist und als Tiefentextur vorliegt (Abbildung 49a).

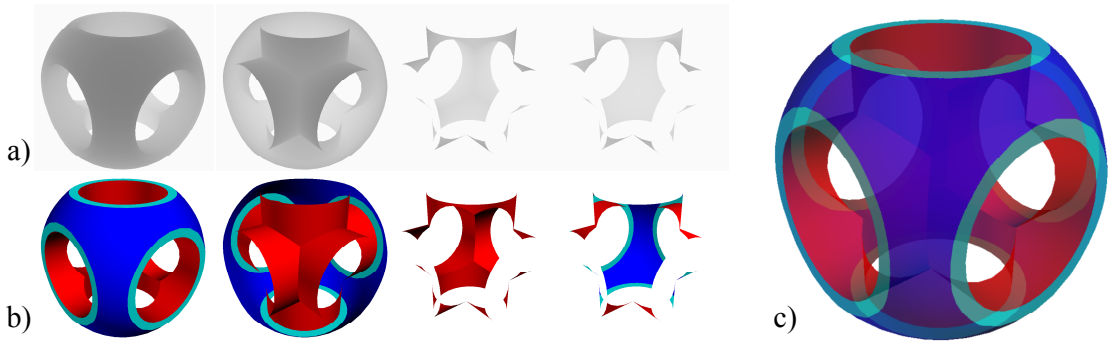

Abbildung 49: Tiefenschichten eines CSG-Modells (a) und zugehörige Farbschichten (b). Resultierendes, transparent dargestelltes CSG-Modell (c).

Berechnung innerer Tiefenschichten

Falls eine Szene mehrere partielle Produkte gleichzeitig enthält, können für innere Tiefenschichten sowohl die Vorderflächen als auch die rückseitigen Flächen aller Primitive Teil der Tiefenschicht, also potentiell sichtbar sein. Daher muss für jedes Primitiv die Sichtbarkeit zunächst für seine Vorderflächen und dann für seine rückseitigen Flächen getestet werden. Alle potentiell sichtbaren Flächen werden dabei unter Berücksichtigung der vorherigen Tiefenschicht ermittelt, um zu gewährleisten, dass nur hinter der Tiefenschicht liegende Geometrie für die CSG-Berechnung herangezogen wird. Hierzu wird, wie bei der ordnungsunabhängigen Transparenz, der texturbasierte Schattentest als zweiter Tiefentest verwendet.

Die anschließende Berechnung der Paritäten erfolgt wie beim normalen Algorithmus von Goldfeather. Zum einen wird also die vorherige Tiefenschicht nicht betrachtet. Zum anderen spielt es keine Rolle, ob gerade die Sichtbarkeit einer Vorderfläche oder einer rückseitigen Fläche berechnet wird. Die Auswirkung der Parität auf die Sichtbarkeit potentiell sichtbarer Flächen gilt also in allen Fällen unverändert.

Nach der Berechnung der Sichtbarkeit der vorder- oder der rückseitigen Fläche eines Primitivs werden mit dem Sichtbarkeitstransfer die Tiefeninformationen in den Haupt-Tiefenpuffer kopiert. Dabei wird wiederum die vorherige Tiefenschicht berücksichtigt, damit Fragmente vor dieser Tiefenschicht im Haupt-Tiefenpuffer nicht erzeugt werden.

Berechnung innerer Tiefenschichten für einzelne partielle Produkte

Falls nur ein einziges partielles Produkt transparent gerendert werden soll, kann der obige Algorithmus deutlich beschleunigt werden. Für diesen Fall kann nämlich ausgenutzt werden, dass sich die Vorderseiten des partiellen Produkts mit seinen Rückseiten abwechseln – eine direkte Folge davon, dass das partielle Produkt topologisch geschlossen ist und als einziges dargestellt wird. Damit sind die potentiell sichtbaren Flächen in der zweiten, und in allen geraden Tiefenschichten genau diejenigen Flächen, die in der ersten, und in allen ungeraden Tiefenschichten nicht potentiell sichtbar sind. Das heißt genauer: In ungeraden Tiefenschichten sind nur die Vorderflächen eines geschnittenen bzw. die rückseitigen Flächen eines subtrahierten Primitivs potentiell sichtbar; in geraden Tiefenschichten sind nur die rückseitigen Flächen eines geschnittenen bzw. die Vorderflächen eines subtrahierten Primitivs potentiell sichtbar. Da es genügt, nur für diese Flächen die CSG-Berechnung durchzuführen, reduziert sich im Vergleich zum Algorithmus für mehrere partielle Produkte die Anzahl der Paritätsberechnungen für innere Tiefenschichten um die Hälfte.

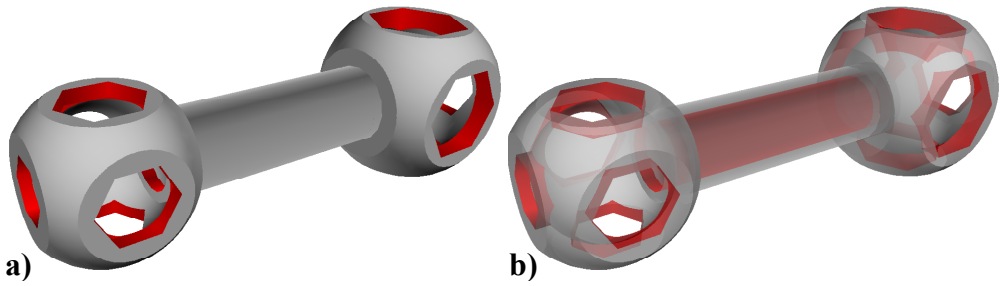

Abbildung 50: Durch CSG modellierter 10-Lochschlüssel ohne (a) und mit Transparenz (b).

Rendering und Ergebnisse

Wie bei der ordnungsunabhängigen Transparenz wird zu jeder Tiefenschicht eine entsprechende Farbschicht angelegt (Abbildung 49b), die die Schattierung und die Transparenz der CSG-Geometrie in der entsprechenden Tiefenschicht enthält. Die zugehörigen RGBA-Texturen werden schließlich zur Darstellung von hinten nach vorne mit Alpha-Blending im Framebuffer zusammengesetzt (Abbildung 49c).

Insgesamt erzeugt dieses Verfahren artefaktfreie Darstellungen transparenter CSG-Modelle. Im Vergleich zur nichttransparenten Darstellung wird, vor allem bei einer kontraststarken Farbauswahl, durch die Transparenz der Gesamtaufbau der CSG-Modelle sehr viel deutlicher (Abbildung 50). Weiter verbessert werden kann die Darstellung durch zusätzliche, bildbasierte Hervorhebung von Kanten für alle Tiefenschichten [46].

-

Kapitel 8

ZUSAMMENFASSUNG UND AUSBLICK

Diese Arbeit stellt Konzepte für die Integration echtzeitfähiger Renderingeffekte in ein Szenengraphsystem vor. Als Kernidee wird die Deklaration von Renderingeffekten in einer Szenenbeschreibung durch den Anwendungsentwickler getrennt von den Algorithmen, die zur Auswertung der Renderingeffekte erforderlich sind. Diese Algorithmen werden durch einen komponentenbasierten Ansatz intern vom Szenengraphsystem bereitgestellt. Durch diese Aufteilung können erstmalig wesentliche Renderingeffekte auf einem hohen Abstraktionsniveau in einem Szenengraphsystem für Echtzeitrendering integrativ eingesetzt werden.

Als geeignetes Mittel für eine Deklaration von Renderingeffekten werden abstrakte Attribute verwendet, durch die die Auswirkung eines Effekts auf die Geometrie in der Szenenbeschreibung festgelegt wird. Abstrakte Attribute erweisen sich vor allem bei der Deklaration von Beleuchtung erster Ordnung anderen Formen der Deklaration als überlegen, weil sie das Deklarieren mehrerer Beleuchtungseffekte für dieselbe Geometrie erlauben.

Zur Auswertung von abstrakten Attributen in einer Szenenbeschreibung muss der Algorithmus zur Szenenauswertung durch verschiedene Konzepte erweitert werden: Nötig sind Auswertungskomponenten zur Kapselung globaler Multipass-Verfahren, globale Renderingtechniken zur Kapselung globaler Multipass-Verfahren mit Bezug auf einzelne Geometrie-Objekte und lokale Renderingtechniken zur Kapselung lokaler Multipass-Verfahren.

Eine kombinierte Nutzung verschiedener Renderingeffekte wird ermöglicht durch verschiedene Ansätze:

- Die Kombination von Komponenten zur Kapselung globaler Multipass-Verfahren mit anderen Renderingtechniken ist unproblematisch, sofern die verwendeten Renderingverfahren nicht die gleichen Ressourcen der Graphikhardware benötigen.
- Globale Renderingtechniken untereinander werden durch eine sorgfältige Abstimmung der einzelnen Renderingdurchläufe der verschiedenen Techniken kombinierbar.

- Von lokalen Renderingtechniken wird definitionsgemäß immer nur jeweils eine zur gleichen Zeit verwendet, was durch den Auswertungsalgorithmus sichergestellt wird.
- Globale Renderingtechniken kommunizieren lokalen Renderingtechniken die Art eines Renderingdurchlaufs, so dass diese daraufhin entscheiden können, ob sie für diesen Durchlauf verwendet werden.

Als komplexes Beispiel eines Renderingeffekts wurde ein bildbasiertes Verfahren der konstruktiven Festkörpergeometrie implementiert und integriert. Zur Auswertung eines CSG-Objekts wird eine lokale Renderingtechnik verwendet, die eine Renderingbibliothek für bildbasiertes CSG nutzt. Dieser Ansatz ermöglicht mit minimalem Aufwand die Kombination mit Beleuchtungseffekten erster Ordnung. Letztlich stellt die Verwendung lokaler Renderingtechniken für CSG allerdings eine Zweckentfremdung dieser dar, denn CSG ist selbst keine Oberflächenschattierung, für die lokale Renderingtechniken ansonsten ausschließlich eingesetzt werden. Die Nutzung von CSG zusammen mit einer Oberflächenschattierung ist dennoch möglich, indem ein CSG-Attribut ein Shaderattribut zur Festlegung einer Oberflächenschattierung referenziert.

Die Entwicklung von OpenCSG als Renderingbibliothek für bildbasiertes CSG ist ein wichtiger Schritt zur Nutzung von CSG nicht nur in allgemeinen Graphiksystemen, sondern auch in beliebigen anderen Anwendungen. Hierzu ist zum einen die Schnittstelle verantwortlich, die es ermöglicht, jede anwendungsdefinierte OpenGL-Geometrie als ein Primitiv für OpenCSG zu verwenden. Zum anderen hat OpenCSG außer OpenGL keine Abhängigkeiten, basiert also nicht auf einem allgemeinen Graphiksystem, und legt darum die Architektur einer Anwendung nicht fest. Damit kann die Bibliothek in alle, auch in schon bestehende Anwendungen leicht integriert werden. Durch die vielfältigen Optimierungen ist die Echtzeitdarstellung auch komplexer CSG-Objekte möglich.

Offene Fragen

Eine bisher nicht geklärte Frage ist, ob für globale Renderingtechniken ein Mechanismus zur automatischen Bestimmung der Reihenfolge ihrer Durchläufe entwickelt werden kann. Dazu wären bestimmte Konventionen über Vor- und Nachbedingungen der Durchläufe einer Renderingtechnik erforderlich. Ein solcher Ansatz würde den derzeitigen Nachteil beheben, dass die Integration einer weiteren globalen Renderingtechnik die manuelle Einordnung ihrer Durchläufe in die schon vorhandenen Durchläufe aller anderen Renderingtechniken erfordert, damit die Techniken zusammenarbeiten können. Bei einer steigenden Anzahl von Renderingtechniken ist diese Arbeit zunehmend fehleranfällig.

Darüber hinaus könnte das Zusammenwirken von Renderingtechniken mit typischen Geschwindigkeitsoptimierungen bei der Szenengraphauswertung verbessert werden. Vor allem State-Sorting ist eine in vielen Anwendungsfällen entscheidende Optimierung, zu dessen Unterstützung eine Weiterentwicklung des Konzepts der Renderingtechniken erforderlich wäre. Des weiteren könnte die Frage, wie die Anzahl von Renderingdurchläufen minimiert werden kann, vor dem Hintergrund eines allgemeinen, zu entwickelnden Kombinationsmechanismus für programmierbare Shader umfassender untersucht werden.

LITERATURVERZEICHNIS

[1] O. Abert: „OpenSG Starter Guide, 1.4.0“, 2004.
http://www.opensg.org/downloads/OpenSG-1.4.0-UserStarter.pdf

[2] M. Abrash: „Quake’s Lighting Model“, *Graphics Programming Black Book, Special Edition*, Coriolis Group Books, 1244--1256, 1997.
http://www.ddj.com/articles/2001/0165/0165f/0165f.htm

[3] T. Akenine-Möller, E. Haines: *Real-Time Rendering, Second Edition*, A.K. Peters Ltd., 2002.

[4] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, C. Cruz-Neira: „VR Juggler: A Virtual Platform for Virtual Reality Application Development“, *Proceedings of IEEE Virtual Reality 2001*, 89--96, 2001.

[5] J. F. Blinn: „Models of Light Reflection for Computer Synthesized Pictures“, *Computer Graphics (Proceedings of SIGGRAPH 77)*, 11(2):192--198, 1977.

[6] J. F. Blinn: „Simulation of Wrinkled Surfaces“, *Computer Graphics (Proceedings of SIGGRAPH 78)*, 12(3):286--292, 1978.

[7] J. F. Blinn: „Ten More Unsolved Problems in Computer Graphics“, *IEEE Computer Graphics and Applications*, 18(5):86--89, 1998.

[8] D. Blythe, B. Grantham, M. J. Kilgard, T. McReynolds, S. R. Nelson: „Antialiasing“, *SIGGRAPH 1999 Course Notes, Course #29 (Advanced Graphics Programming Techniques Using OpenGL)*, 91--94, 1999.

[9] S. Brabec, T. Annen, H.-P. Seidel: „Shadow Mapping for Hemispherical and Omni-directional Light Sources“, *Advances in Modelling, Animation and Rendering (Proceedings Computer Graphics International 2002)*, Springer, 397--408, 2002.

[10] S. Brabec, H.-P. Seidel: „Single Sample Soft Shadows using Depth Maps“, *Proceedings Graphics Interface*, 219--228, 2002.

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan: „Brook for GPUs: Stream Computing on Graphics Hardware“, *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2004)*, 23(3):777--786, 2004.

[12] „CgFX Overview“, *Technical Report, NVidia Corperation*, 2003.
http://www.developer.nvidia.com/attach/5631

[13] G. Coombe, M. J. Harris, A. Lastra: „Radiosity on graphics hardware “, *Proceedings Graphics Interface*, 161--168, 2004.

[14] F. C. Crow: „Shadow Algorithms for Computer Graphics“, *Computer Graphics (SIGGRAPH 77 Proceedings)*, 11(2):242--248, 1977.

[15] Joe Demers: „Depth of Field: A Survey of Techniques“, *GPU Gems – Programming Techniques, Tips, and Tricks for Real-Time Graphics* (Randima Fernando, Editor), Addison-Wesley, 375--390, 2004.

[16] P. J. Diefenbach: *Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering*, PhD thesis, University of Pennsylvania, 1996.

[17] P. J. Diefenbach, B. M. Housel, J. E. John, O. G. Mellet: „The Shadow Node“, *Technical Report, OpenWorlds*, 2001.
`http://www.openworlds.com/newnodes/ shadowpaper/index.html`

[18] J. Döllner, K. Hinrichs: „A Generic Rendering System“, *IEEE Transactions on Visualization and Computer Graphics*, 8(2):99--118, 2002.

[19] J. Döllner, M. Walther: „Real-time Expressive Rendering of City Models“, *Proceedings of the 7th IEEE International Conference on Information Visualization*, 245--250, 2003.

[20] J. Döllner, H. Buchholz, M. Nienhaus, F. Kirsch: „Illustrative Visualization of 3D City Models“, *Proceedings of SPIE, Vol. 5669, Visualization and Data Analysis (VDA 2005)*, 42--51, 2005.

[21] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, D. Weiskopf: „Sampling a Volume Via Texture Mapping“, *SIGGRAPH 2004 Course Notes, Course #28 (Real-Time Volume Graphics)*, 24--36, 2004.

[22] C. Everitt: „Interactive Order-Independent Transparency“, *Technical report, NVidia Corporation*, 2001.
`http://developer.nvidia.com/attach/6545`

[23] C. Everitt, M. J. Kilgard: „Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering“, *Technical Report, NVidia Corperation*, 2002.
`http://developer.nvidia.com/attach/6829`

[24] „Extensible 3D (X3D)“, International Standard ISO/IEC FDIS 19775-1:2004.

[25] J. Foley, A. Van Dam, S. Feiner, J. Hughes, R. Phillips: *Introduction to Computer Graphics*, Addison-Wesley, 1993.

[26] J. Goldfeather, J. P. M. Hultquist, H. Fuchs: „Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System“, *Computer Graphics (SIGGRAPH 86 Proceedings)*, 20(4):107--116, 1986.

[27] J. Goldfeather, S. Molnar, G. Turk, H. Fuchs: „Near Realtime CSG Rendering Using Tree Normalization and Geometric Pruning“, *IEEE Computer Graphics and Applications*, 9(3):20--28, 1989.

[28] R. C. Gonzalez, R. E. Woods: *Digital Image Processing, 2nd Edition*, Prentice Hall, 2002.

[29] A. Gooch, B. Gooch, P. Shirley, E. Cohen: „A Non-Photorealistic Lighting Model for Automatic Technical Illustration“, *Proceedings of ACM SIGGRAPH 1998*, 447--452, 1998.

[30] „GPGPU - General-Purpose Computation Using Graphics Hardware“, 2005.
`http://www.gpgpu.org`

[31] K. Gray: *DirectX® 9 Programmable Graphics Pipeline*, Microsoft Press, 2003.

[32] N. Greene: „Environment Mapping and Other Applications of World Projections“, *IEEE Computer Graphics and Applications*, 6(11):21--29, 1986.

[33] S. Guha, S. Krishnan, K. Munagala, S. Venkatasubramanian: „Application of the Two-Sided Depth Test to CSG Rendering“, *ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics*, 177--180, 2003.

[34] P. Haeberli, K. Akeley: „The Accumulation Buffer: Hardware Support for High-Quality Rendering“, *Computer Graphics (SIGGRAPH 90 Proceedings)*, 24(4):309--318, 1990.

[35] T. Hall: „A How To for Using OpenGL to Render Mirrors“, Posting at `comp.graphics.api.opengl`, 1996.
`http://www.opengl.org/resources/code/rendering/mjktips/TimHall_Reflections.txt`

[36] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, F. Sillion: „A Survey of Real-Time Soft Shadows Algorithms“, *Computer Graphics Forum*, 22(4):753--774, 2003.

[37] P. S. Heckbert, M. Herf: „Simulating Soft Shadows with Graphics Hardware“, *Technical Report TR CMU-CS-97-104*, Carnegie Mellon University, 1997.

[38] T. Heidmann: „Real Shadows Real Time“, *Iris Universe*, 18:28--31, 1991.

[39] W. Heidrich: *High-quality Shading and Lighting for Hardware-accelerated Rendering*, Dissertation, Universität Erlangen-Nürnberg, 51--52, 1999.

[40] F. W. Jansen, A. N. T. van der Zalm: „A Shadow Algorithm for CSG“, *Computers and Graphics*, 15(2):237--247, 1991.

[41] M. J. Kilgard: „Rendering a Magic Halo with OpenGL”, 1997.
`http://www.opengl.org/resources/code/rendering/mjktips/StenciledHaloEffect.html`

[42] M. J. Kilgard: „A Practical and Robust Bumpmapping Technique for Today's GPUs“, *Game Developers Conference*, 2000.
`http://developer.nvidia.com/attach/6588`

[43] F. Kirsch, J. Döllner: „Real-Time Soft Shadows Using a Single Light Sample“, *Journal of WSCG*, 11(2):255--262, 2003.

[44] F. Kirsch, J. Döllner: „Rendering Techniques for Hardware-Accelerated Image-Based CSG“, *Journal of WSCG*, 12(2):221--228, 2004.

[45] F. Kirsch, J. Döllner: „OpenCSG: A Library for Image-Based CSG Rendering“, *Proceedings of the FREENIX / Open Source Track, 2005 USENIX Annual Technical Conference*, 129--140, 2005.

[46] F. Kirsch, M. Nienhaus, J. Döllner: „Visualizing Design and Spatial Assembly of Interactive CSG using Blueprint Rendering“ (eingereicht zum *Eurographics Symposium on Rendering 2005*).

[47] A. Lake, C. Marshall, M. Harris, M. Blackstein: „Stylized Rendering Techniques for Scalable Real-time Animation”, *First International Symposium on Non Photorealistic Animation and Rendering (NPAR)*, 13--20, 2000.

[48] E. Lindholm, M. J. Kilgard, H. Moreton: „A User-Programmable Vertex Engine“, *Proceedings of ACM SIGGRAPH 2001*, 149--158, 2001.

[49] W. R. Mark, K. Proudfoot: „The F-buffer: a Rasterization-Order FIFO Buffer for Multi-Pass Rendering“, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 57--64, 2001.

[50] T. Martin, T.S. Tan: „Anti-aliasing and Continuity with Trapezoidal Shadow Maps“, *Proceedings of the 15th Eurographics Symposium on Rendering*, 153--160, 2004.

[51] M. D. McCool, Z. Qin, T. S. Popa: „Shader Metaprogramming“, *SIGGRAPH/ Eurographics Workshop on Graphics Hardware*, 57--68, 2002.

[52] M. D. McCool, S. Du Toit, T. S. Popa, B. Chan, K. Moule: „Shader Algebra“, *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2004)*, 23(3):787--795, 2004.

[53] Y. V. Miroshnik: „Bumped Reality – Alternate Bump Mapping using OpenGL“, 2003. `http://glazsd.tripod.com/proj/bump`

[54] S. Münz, W. Nefzger: *HTML 4.0 Handbuch, Studienausgabe*, Franzis, 1999.

[55] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, T. Limperis: „Geometric Considerations and Nomenclature for Reflectance“, Monograph 161, National Bureau of Standards (US), 1977.

[56] M. Nienhaus, J. Döllner: „Edge-Enhancement - An Algorithm for Real-Time Non-Photorealistic Rendering“, *Journal of WSCG*, 11(2):346--353, 2003.

[57] „NVIDIA Scene Graph Software Development Kit (NVSG SDK)“, 2004. `http://developer.nvidia.com/object/nvsg_home.html`

[58] M. Olano, K. Akeley, J. C. Hart, W. Heidrich, M. McCool, J. L. Mitchell, R. Rost: „Shader-Model 3.0 – No Limits“, *SIGGRAPH 2004 Course Notes, Course #1 (Real-Time Shading)*, 2004.

[59] OpenGL® Extension Registry, 2005. `http://oss.sgi.com/projects/ogl-sample/registry`

[60] R. Osfield, D. Burns: „OpenSceneGraph“, 2005. `http://www.openscenegraph.com`

[61] C. Peeper, J. L. Mitchell: „Introduction to the DirectX® 9 High Level Shading Language“, *ShaderX²: Introductions and Tutorials with DirectX 9.0*, Wolfgang F. Engel (Editor), Wordware Publishing, 1--61, 2003.

[62] B.T. Phong: „Illumination for Computer Generated Images“, *Communications of the ACM*, 18(6):311--317, 1975.

[63] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein: „Real-time hatching“, *Proceedings of ACM SIGGRAPH 2001*, 579--584, 2001.

[64] „Programmer’s Hierarchical Interactive Graphics System (PHIGS) Functional Description“, International Standard ISO/IEC 9592. Ref. ANSI X3.144-1988, New York, 1988.

[65] K. Proudfoot, W. R. Mark, S. Tzvetkov, P. Hanrahan: „A Real-Time Procedural Shading System for Programmable Graphics Hardware“, *Proceedings of ACM SIGGRAPH 2001*, 159--170, 2001.

[66] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan: „Ray Tracing on Programmable Graphics Hardware“, *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002)*, 21(3):703--712, 2002.

[67] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, P. Hanrahan: „Photon Mapping on Programmable Graphics Hardware“, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 41--50, 2003.

[68] W. T. Reeves, D. H. Salesin, R. L. Cook: „Rendering Antialiased Shadows with Depth Maps“, *Computer Graphics (SIGGRAPH 87 Proceedings)*, 21(4):283--291, 1987.

[69] D. Reiners: *OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications*, Dissertation, Technische Universität Darmstadt, 2002.

[70] A. A. G. Requicha: „Representations for Rigid Solids: Theory, Methods, and Systems“, *ACM Computing Surveys*, 12(4):437--464, 1980.

[71] J. Rohlf, J. Helman: „IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics“, *Proceedings of ACM SIGGRAPH 1994*, 381--394, 1994.

[72] R. J. Rost: *OpenGL Shading Language*, Addison Wesley, 2004.

[73] M. Segal, K. Akeley: „The OpenGL® Graphics System: A Specification (Version 2.0 – October 22, 2004)“, 2004.
`http://www.opengl.org/documentation/spec.html`

[74] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, P. Haeberli, „Fast Shadows and Lighting Effects using Texture Mapping“, *Computer Graphics (SIGGRAPH 92 Proceedings)*, 26(2):249--252, 1992.

[75] H. Sowizral: „Scene Graphs in the New Millennium“, *IEEE Computer Graphics and Applications*, 20(1):56--57, 2000.

[76] H. Sowizral, K. Rushforth, M. Deering: „The Java 3D™ API Specification, Version 1.3“, *Sun Microsystems*, 2002.
`http://java.sun.com/products/java-media/3D`

[77] M. Stamminger, G. Drettakis: „Perspective Shadow Maps“, *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002)*, 557--562, 2002.

[78] N. Stewart, G. Leach, S. John: „An Improved Z-Buffer CSG Rendering Algorithm“, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 25--30, 1998.

[79] N. Stewart, G. Leach, S. John: „A CSG Rendering Algorithm for Convex Objects“, *Journal of WSCG*, 8(2):369--372, 2000.

[80] N. Stewart, G. Leach, S. John: „Linear-time CSG Rendering of Intersected Convex Objects“, *Journal of WSCG*, 10(2):437--444, 2002.

[81] P. Strauss, R. Carey: „An Object-Oriented 3D Graphics Toolkit“, *Computer Graphics (SIGGRAPH 92 Proceedings)*, 26(2):341--349, 1992.

[82] T. Strothotte, S. Schlechtweg: *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*, Morgan Kaufmann, 2002.

[83] S. Sudarsky: „Generating Dynamic Shadows for Virtual Reality Applications“, *Proceedings of the 5th IEEE International Conference on Information Visualization*, 595--600, 2001.

[84] I. Sutherland: *Sketchpad: A Man-Machine Graphical Communication System*, PhD thesis, University of Cambridge, 1963.

[85] M. S. Tawfik: „An efficient algorithm for CSG to b-rep conversion“, *Proceedings of the first ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications (SMA '91)*, 99--108, 1991.

[86] „TGS OpenInventor from Mercury 5 User’s Guide“, Kapitel 25: SolidViz, 587--592, Mercury, 2004.
`http://www.tgs.com/support/oiv_doc`

[87] „The Virtual Reality Modeling Language (VRML)“, International Standard ISO/IEC 14772-1:1997 and ISO/IEC 14772-2:2002.

[88] Wang, Y., S. Molnar: „Second-Depth Shadow Mapping“, *UNC-CS Technical Report TR94-019*, University of North Carolina at Chapel Hill, 1994.

[89] T. F. Wiegand: „Interactive Rendering of CSG Models“, *Computer Graphics Forum*, 15(4):249--261, 1996.

[90] L. Williams: „Casting Curved Shadows on Curved Surfaces“, *Computer Graphics (SIGGRAPH 78 Proceedings)*, 12(3):270--274, 1978.

[91] M. Wimmer, D. Scherzer, W. Purgathofer: „Light Space Perspective Shadow Maps“, *Proceedings of the 15th Eurographics Symposium on Rendering*, 143--152, 2004.

[92] M. Wloka, R. Zeleznik: „Interactive Real-Time Motion Blur“, *The Visual Computer*, 12(6):283--295, 1996.

STICHWORTVERZEICHNIS