Bauhaus-Universität Weimar · Fakultät Medien

Diplomarbeit

Alternative Szenengraphenkonzepte

Andreas-Christoph Bernstein

30. November 2007

Betreuer:
Prof. Dr. Bernd Fröhlich

Externer Betreuer:
Dipl.-Inf. Roland Kuck

Danksagung

Ich möchte mich an dieser Stelle bei Prof. Dr. Bernd Fröhlich für seine Bereitschaft bedanken, diese Diplomarbeit anzunehmen und sie über die Distanz hinweg zu betreuen. Dank geht vor allem auch an alle beteiligten Kollegen der Abteilung Virtual Environments des Fraunhofer Instituts IAIS, im Besonderen an Dr. Manfred Bogen, Roland Kuck, Kai Riege und David D'Angelo. Ebenso gilt mein Dank meinen Eltern für die finanzielle Unterstützung meines Studiums.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Dieses Kapitel gibt einen Überblick über den Inhalt der Arbeit. Nach einer kurzen Einleitung und Motivation wird das Ziel und der Aufbau der Arbeit vorgestellt.

1.1 Motivation

Szenengraphen sind ein erfolgreiches Konzept, das zur Beschreibung von dreidimensionalen und interaktiven Grafikszenen benutzt wird. Sie bilden die Grundlage einer Vielzahl von Modellierungsprogrammen, Computerspielen. Ausserdem werden sie für Simulationen im Bereich der Virtuellen Realität (VR). Grundlage des VR-Framework Avango (Tramberend (1999)) ist beispielsweise der Szenengraph OpenGL|Performer. Szenengraphen bieten Entwicklern eine objektorientierte und strukturierte Herangehensweise, um interaktive Grafikanwendungen zu beschreiben. Als eine Abstraktion zu Low-Level-Bibliotheken wie OpenGL oder DirectX ermöglichen sie es dem Anwender sich eher auf die Beschreibung der Szenedaten, als um die eigentlichen Rendering Vorgänge, zu kümmern.

Zwischen den Objekten einer Szene bestehen verschiedenste Beziehungen. Um dies mit einem Szenengraph auszudrücken, werden Objekte nach verschiedenen Betrachtungen strukturiert. Die Objekte können nach räumlicher Nähe angeordnet werden und relative zueinander positioniert werden. Diese geometrischen Informationen kann genutzt werden um potentiell sichtbare Objekte zu bestimmen. Dadurch wird zur Performanz-Steigerung die Menge der zu zeichnenden Geometrie reduziert. Eine weiter Möglichkeit ist es, die Objekte nach gemeinsame Materialien zu gruppieren. Dies ermöglicht es beim Zeichnen unnötige und teure State-Veränderungen des Graphiksubsystems zu vermeiden. Dies geht aber auf Kosten der räumlichen Kohärenz, so daß unnötige Tests bei der Aussortierung nicht sichtbarer Objekte auftreten können. Ebenso können Objekte aber auch nach logischen Gesichtspunkten gruppiert werden, um zum Beispiel Zugriff auf alle Autos oder Häuser einer Szene zu bekommen. Leider gibt es keine perfekte Organisation die für all diese Betrachtungen zusammen optimal ist.

Ein weiteres Thema sind moderne Rendering-Verfahren, die aus mehreren Zeichenoperationen bestehen und dazu den Graphen mehrmals traversieren. Zwischen diesen Operationen können Abhängigkeiten auftreten, so daß eine Ausführungsreihenfolge gegeben ist. Um damit umgehen zu können, müssen diese möglichst

flexibel sein.

In der vorliegenden Arbeit werden zwei alternative Szenengraphenkonzepte vorgestellt, welche das Verlangen nach Performanz und Flexibilität adressieren. Dazu wurden jeweils zwei Prototypen entwickelt und anhand von Testfällen untersucht und verglichen. Hierbei wurde sich auf Rendering konzentriert.

1.2 Überblick über die Arbeit

Kapitel 2 beschreibt den Begriff des Szenengraphen. Dazu werden Knotentypen und ihre Bedeutung im Graphen vorgestellt. Hier wird erklärt, wie die Aktualisierung des Szenengraphen geschieht. Außerdem werden die wichtigen Operationen Culling und Statesorting zur Performanzverbesserung beschrieben.

Kapitel 3 gibt einen Überblick über wichtige, klassische und bekannte Szenengraphen. Hierzu werden die besonderen Merkmale der jeweiligen Szenengraphen, die sie voneinander unterscheiden, hervorgehoben.

Kapitel 4 beschreibt das Szenengraphenkonzept MultiSG, dessen Hauptaugenmerk auf der Unterstützung mehrerer Strukturen liegt. Dazu werden die benutzten Hierarchien und die Algorithmen, die auf diesen arbeiten vorgestellt.

Kapitel 5 beschreibt das flexible Framework ViewSG, das es erlaubt, komplexe Rechenoperationen zu modellieren und effizient aufzulösen.

In Kapitel 6 wird gezeigt, wie die beiden Konzepte genutzt werden. Dazu werden zwei Testfälle beschrieben und demonstriert, wie die jeweilige Implementierung aussieht.

Kapitel 7 vergleicht und diskutiert die vorgestellten Ansätze. Hierzu wird auf die Punkte Erweiterbarkeit und Benutzbarkeit eingegangen. Zudem wird die Ablaufkoordination von Szenengraphenoperationen untersucht.

1.3 Konvention

Begriffe die aus dem Englischen stammen und für die es keine passende, im Kontext sinnvolle Übersetzung gibt, werden in dieser Arbeit als englische Fachbegriffe gebraucht.

2 Szenengraphen

Dieses Kapitel gibt einen kurzen Überblick und Beschreibung darüber, was man unter Szenengraphen im Allgemeinen versteht. Der Begriff Szenengraph läßt sich nur sehr unscharf definieren. Im Kapitel 3 wird auf einige Frameworks genauer eingegangen.

Ein Szenengraph ist eine Datenstruktur, die genutzt wird, um 2D- oder 3D-Szenen zu beschreiben. Die Szenendaten werden in den Knoten eines gerichteten, azyklischen Graphen gehalten, dessen Knoten Geometrien, Gruppierungen, Transformationen, Lichter oder auch Eigenschaften, wie Materialien repräsentieren können. Die Kanten des Graphen beschreiben logische oder räumliche Zusammenhänge und Beziehungen zwischen den Knoten. Das heißt Kinderknoten erben die States der Vaterknoten.

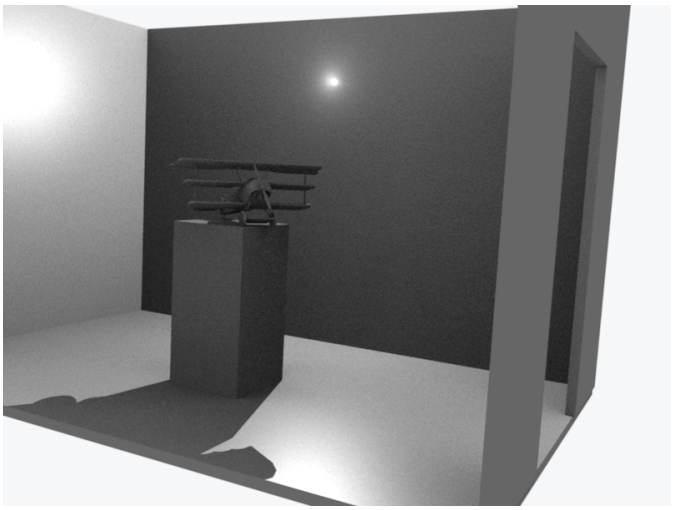

Abbildung 2.1: Beispielszene

Zum Beispiel hat die in Bild 2.1 dargestellte Szene die in Abbildung 2.2 gezeigte Transformationshierarchie. Die Ellipsen in Abbildung 2.2 repräsentieren Knoten, die die Geometrie enthalten, und die Rechtecke stehen für Transformationsknoten.

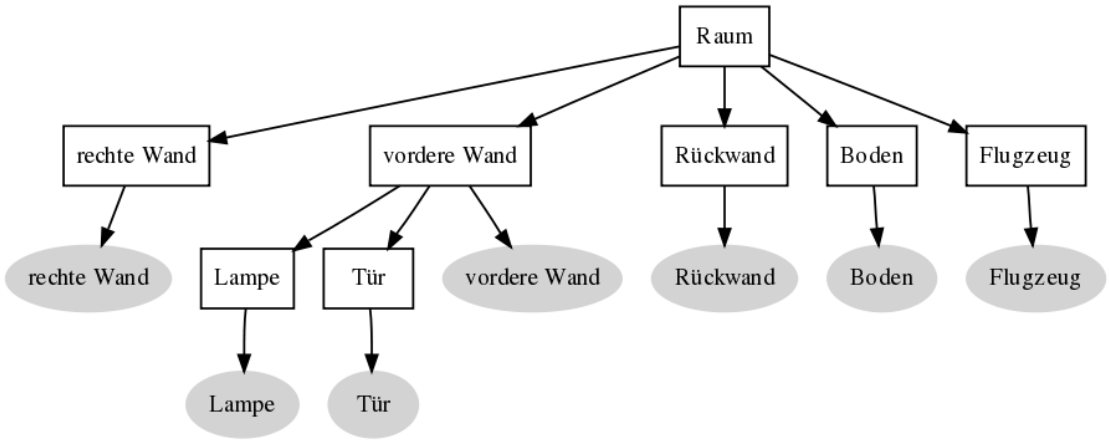

Abbildung 2.2: Hierarchie der Beispielszene

2.1 Knotentypen

Im folgenden werde ich einen kurzen Überblick über Knotentypen geben, die von den meisten Szenengraphen in der ein oder anderen Form benutzt und angeboten werden. Der Wurzelknoten des Szenengraphe repräsentiert die gesamte Szene.

Group

Ein Gruppenknoten ist ein einfacher Container. Mit dessen Hilfe kann man Objekte, die als eine Einheit behandelt werden sollen, zusammenfassen. Das dahinter stehende Design Pattern ist das Composite Pattern (Gamma (2002)).

Transformation

Transformationen werden genutzt um Objekte zu platzieren, orientieren und um ihre Größe zu ändern. Transformationsknoten sind Gruppenknoten, welche eine Transformation halten, die beispielsweise mit einer 4×4 Matrix beschrieben werden kann. Damit wird ein neues lokales Koordinatensystem beschrieben. Die Kante, die von einem Transformationsknoten zu einem Kind führt, steht für eine Positionierung des Kindes relativ zum Transformationsknoten.

LOD – Level-Of-Detail

Ein LOD ist ein Gruppenknoten, welcher von einem Modell verschiedene detaillierte Geometrien als Kinderknoten hat. Die Grundidee ist, daß Modelle, die weiter

vom Betrachter entfernt sind und nur einen kleinen Bildbereich einnehmen, mit einem geringeren Detailgrad dargestellt werden können. Dabei kann beim Zeichnen entweder in Abhängigkeit von der Entfernung des Betrachters eine der Geometrien ausgewählt werden oder es wird zwischen ihnen überblendet.

Switch

Ein Switch ist ein Gruppenknoten der maximal nur ein aktives Kind hat. Sequenzknoten sind Switches, die abhängig von der Zeit das aktive Kind umschalten. Mit Hilfe eines Sequenzknoten können einfache Animationen realisiert werden, indem jeder Frame einer Animation als Kind des Sequenzknoten vorhanden ist.

Geometry

Die Geometriedaten eines zeichenbaren Objektes werden in den Blattknoten des Szenengraphen gehalten.

Light

Die Position einer Lichtquelle im Graphen kann mehrere Bedeutungen haben Eine Lichtquelle beleuchtet ihren Subgraphen. Ihre Position kann über die Position im Graphen, durch Referenzierung eines anderen Knotens der Hierarchie oder in Weltkoordinaten bestimmt sein.

2.2 Berechnung der Welttransformation

Die lokale Transformation spezifiziert die relative Position zum Vaterknoten. Diese Transformation kann durch eine homogene Matrix ohne perspektivische Komponente dargestellt werden. Diese Matrix beschreibt Translation, Rotation, Skalierung und Scherung des Knotens im Koordinatensystem des Vaterknotens.

Die Welttransformation eines Knotens hängt von seiner lokalen Transformation und der der Vorgänger des Knotens bis zur Wurzel ab.

Für einen Transformationsknoten P mit dem Transformationskind C ergibt sich die Welttransformation von C als das Produkt aus P's Welttransformation und C's lokaler Transformation. Die Welttransformation des Wurzelelementes ist einfach dessen lokale Transformation.

Das heißt

$$W_C = W_P L_C$$

wobei W_C die Welttransformation von C, W_P die von P und L_C die Lokale Transformation von C ist. Die Welttransformation eines Elementes E_k im Pfad $E_0 \cdots E_k$,

wobei E_0 das Wurzelelement ist, ist durch rekursive Anwendung obiger Definition gegeben als:

$$W_{E_k} = \prod_{i=0}^{k} L_{E_i}$$

.

2.3 Bounding-Volume-Hierarchie und Culling

Um Szenengraphen effizient rendern zu können, wird eine Hüllkörperhierarchie, auch Bounding-Volume-Hierarchie genannt (BVH), eingesetzt. Jeder Knoten des Graphen hat einen Hüllkörper, der dessen räumliche Ausdehnung inklusive seiner Kinder umfaßt. Vor dem Rendern werden zuerst alle potentiell sichtbaren Objekte bestimmt. Dazu wird der Graph von der Wurzel aus traversiert. Jeder Hüllkörper wird gegen die Sichtpyramide der Kamera getestet. Liegt der Knoten außerhalb, müssen seine Kinder folglich auch außerhalb der Sichtpyramide sein und nicht weiter betrachtet werden. Dadurch ist das aussortieren eines ganzen Subgraphens möglich. Dieses Aussortieren wird im weiteren Text als Culling bezeichnet.

In Abbildung 2.3 ist eine schematische Darstellung zu sehen. Die roten Vierecke sind sichtbar, das blassrote Viereck nur teilweise und das transparente ist nicht sichtbar.

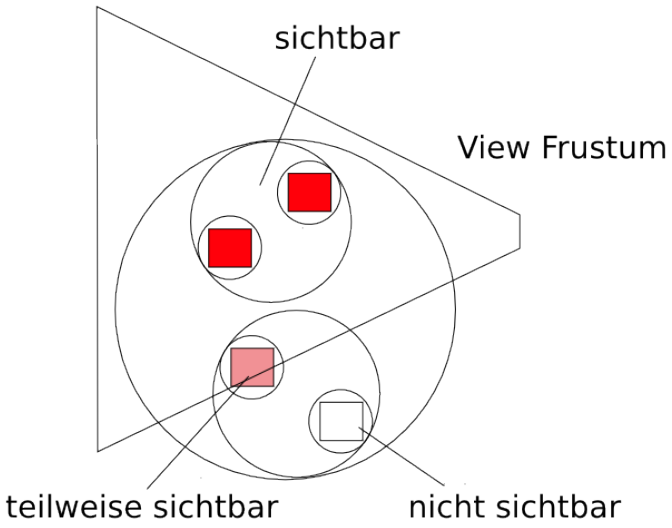

Abbildung 2.3: Culling Beispiel

Dieselben ineinander verschachtelten Hüllkörper unterstützen auch eine Kollisionserkennung. Wenn der Hüllkörper eines Knotens ein zu betrachtendes, anderes

Bounding-Volume nicht schneidet, müssen auch nicht mehr dessen Kinder getestet werden. Um den Hüllkörper eines inneren Knoten berechnen zu können, müssen zuvor die Hüllkörper seiner Kinderknoten bekannt sein. Dies geschieht, wenn ein rekursiver Traversierer wieder den Graphen aufsteigt.

2.4 Renderstate und Sortierung

Die von den meisten Szenengraphen genutzten Grafikbibliotheken sind OpenGL und DirectX, welche State-Maschinen sind.

Der Renderstate ist eine Menge von Attributen und Modi, die genutzt werden, um den Zustand des Grafiksubsystems zu konfigurieren. Dadurch wird das Aussehen von zu zeichnender Geometrie bestimmt.

Der Renderstate wird auf unterschiedliche Weise gehandhabt. Einige Szenengraphen kapseln den Renderstate ab, um das sortieren zu vereinfachen. Die Knoten referenzieren eine separate Struktur welche ein Set von Material-, Textur-, Transparenz-Attributen und dergleichen umfaßt. Auf solch ein Set können auch mehrere Knoten verweisen. Andere Implementierungen haben Stateknoten im Graphen, wie zum Beispiel einen Materialknoten. Knoten die unterhalb von diesem hängen, erben dann diese Materialeigenschaft. Bei der Modellierung eines Raums beispielsweise, dessen vier Wände aus Beton bestehen, könnte ein “Beton” Materialknoten eingeführt werden dessen Kinder die Wände des Raumes sind. Eine Traversierung vom Wurzelknoten bis zu einem Blattknoten des Graphen akkumuliert den Renderstate, der zum Zeichnen der Geometrie im Blattknoten nötig ist.

Vor dem Zeichnen eines Blattknotens entscheidet der Zeichenprozess, ob sein innerer Status, sprich sein State, verändert werden muß. Da beim Rendern der Wechsel von einem State zum anderen eine teure Operation ist, versucht man diese zu vermeiden, indem Objekte nach gleichen Renderstate sortiert werden, oder nach dem Attribut für das State-Veränderungen minimiert werden soll.

2.5 Picking

Die oben beschriebene Bounding Volume Hierarchie kann auch genutzt werden, um mit Hilfe eines Strahls Elemente der Szene zu selektieren.

Dazu wird ein Schnittest rekursiv zwischen den Hüllkörpern und einem Strahl ausgeführt, um das nächste getroffene Objekt zu bestimmen. Das kann zum Beispiel genutzt werden, um Objekte zu bestimmen, die sich unter dem Mauszeiger befinden, oder für einfache Kollisionserkennung.

2.6 Rendering eines Szenengraphens

Als Renderer wird in diesem Text die Komponente des Grafiksystems bezeichnet, die die Geometrie eines Objektes zeichnet. Dazu nutzt sie den aktuellen State, der auch als Renderstate bezeichnet wird. Das Rendering schließt dann alles ein was zum Zeichnen beiträgt.

Der Renderer nutzt eine Kamera, welche eine Sichtpyramide definiert. Diese Sichtpyramide wird in der Computergrafik meist als *View Frustum* bezeichnet. Der Prozess des Renderns umfaßt eine Traversierung des aktualisierten Graphens.

Dazu werden zuerst die nicht sichtbaren Objekte mittels Culling aussortiert. Wenn ein Subgraph nicht aussortiert ist, wird er rekursiv traversiert. Dabei werden die States aufgesammelt bis ein Blattknoten erreicht wird. Dort hat der Renderer nun alle Informationen, um die Geometrie des Blattknotens zu zeichnen. Dazu muß dieser seine Geometriedaten, wie zum Beispiel Vertices, Normalen, Texturekoordinaten und dergleichen, dem Renderer bereitstellen.

Häufig bieten Szenengraphen auch die Möglichkeit kurz vor und nach dem Zeichnen Funktionen aufzurufen, was zum Beispiel zur dynamischen Tesselierung eines Objektes genutzt werden kann.

3 Related Work

In diesem Kapitel werden verschiedene Szenengraphen vorgestellt und beschrieben, wie sie die Datenstruktur hinsichtlicher einer optimalen Darstellung nutzen.

3.1 Nutzung von Hierarchischen Datenstrukturen

Viele der Techniken, die moderne Szenengraphen heute benutzen, wurden erstmals in (Clark (1976)) beschrieben, wie die Verwendung einer hierarchischen Datenstruktur, Level-Of-Detail (LOD), hierarchisches View Frustum und Occlusion Culling und Nutzung eines Arbeitsset zum Geometrie Caching.

Als das Paper geschrieben wurde, nutzte man hierarchische Datenstrukturen hauptsächlich, um Objekte relativ zueinander zu platzieren und um Clipping Zeiten zu verringern.

Transformationshierarchie

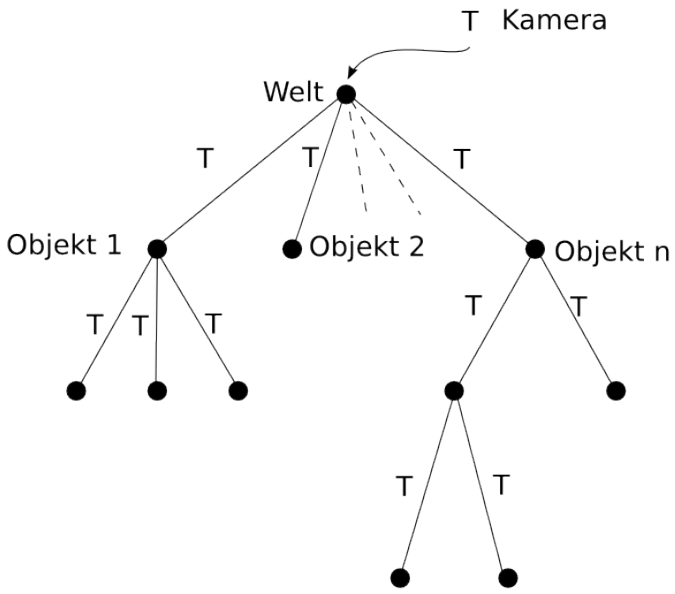

Abbildung 3.1: Transformationshierarchie

Die meisten Rendering-Algorithmen benutzten eine Transformations- oder Bewegungsstruktur, um Position und Orientierung von Objekten relativ zueinander zu beschreiben. Jedoch nutzte kaum jemand diese Strukturen auf der Ebene der Rendering Algorithmen selbst. Das heißt, alle Polygone eines Objektes werden in ein gemeinsames Bildschirmkoordinatensystem transformiert, in welchem der Rendering Algorithmus arbeitet. Ein Beispiel ist in 3.1 gezeigt. Jeder Knoten repräsentiert eine Menge von geometrischen Primitiven, wie zum Beispiel Polygone. Die Kante die zu einem Knoten führt repräsentiert eine Transformation. Das heißt eine Kante beschreibt die Orientierung und Platzierung relativ zum Vaterknoten.

Verringern der Clipping-Zeit

Bei der Simulierung einer Kamera in einer 3D-Umgebung müssen Teile der Umgebung gegen das Sichtfeld (FOV) der Kamera geclippt werden. Dies kann einmal dadurch geschehen, daß alle geometrische Primitive eines Objektes in das Kamera Koordinatensystem transformiert und einzeln geclippt werden, oder indem zu erst ein Hüllkörper des Objektes auf Schnitt mit dem Kamera Frustum getestet wird. Wenn kein Schnitt festgestellt wurde, liegt das Objekt komplett innerhalb oder außerhalb, was das Testen der einzelnen Teile des Objektes erspart.

Die Veröffentlichung von Clark stellte grundlegend neue Nutzungsweisen von Struktur vor, um die Performanz und die Darstellungsqualität von Rendering Algorithmen zu verbessern, die heutzutage von den meisten Szenengraphen genutzt werden.

Level-Of-Detail (LOD)

Der erste Vorschlag von Clark ist heute unter dem Namen Level-Of-Detail bekannt.

Indem man den Detailgrad wählt mit welchem ein Objekt dargestellt wird, fixiert man die Minimaldistanz, von welcher das Objekt realistisch dargestellt werden kann. Beispielsweise sieht ein Dodekahedron in ausreichend großer Entfernung wie eine Kugel aus. Dieser kann somit genutzt werden, um eine Kugel darzustellen solange er von dieser Entfernung oder weiter betrachtet wird. Aus näherer Entfernung sieht das Objekt jedoch wie ein Dodekahedron aus. Eine Lösung ist es, es einfach mit mehr Detailgrad als nötig zu definieren. Jedoch kann es dann mehr Detail haben als nötig ist, um es aus größeren Distanzen zu betrachten. In einer komplexen Welt mit vielen solchen Objekten gäbe es dann zu viele Polygone für die Rendering-Algorithmen, um diese effizient zu handhaben.

Die Lösung bis dato war, die Objekte relativ grob zu definieren und einen cleveren Algorithmus zu nutzen, der die Konturen glättet oder das Shading verbessert, damit das Objekt auch unter näherer Betrachtung realistisch aussieht. Die Schwierigkeit an dieser Herangehensweise ist, daß der Entfernungsbereich zur Betrachtung

nur wenig verbessert wird und daß das Problem von zu vielen Details für größere Entfernung bestehen bleibt. Seine Schlußfolgerung ist, daß verschiedene Detailgrade genutzt werden müssen um adäquat komplexe Umgebungen darzustellen. Das heißt sein Vorschlag ist ein LOD Ansatz.

Dazu definiert Clark “Objekte” in einer Hierarchie, wie in 3.2 zu sehen. Die ganze darzustellende Umgebung ist selbst ein Baum. In dieser Struktur gibt es zwei Arten von Kanten. Die einen repräsentieren Transformationen und die anderen zeigen auf detailliertere Struktur. Alle inneren Knoten stehen für eine ausreichende Beschreibung eines Objektes, wenn dieses nur einen kleinen Anzeigebereich bedeckt. Kanten von diesem Knoten ausgehend verweisen auf detailliertere Versionen des Originalobjektes, die genutzt werden, falls das Objekt eine größere Fläche des Bildschirmes bedeckt. In den Blattknoten des Baumes befinden sich grafische Primitive wie zum Beispiel Polygone. In einer komplexen Umgebung variiert der

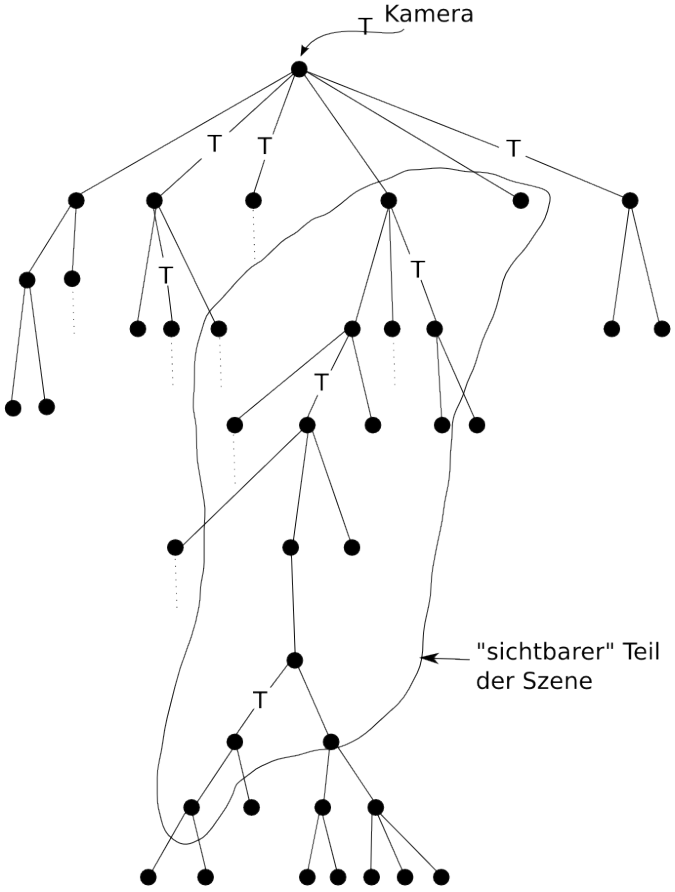

Abbildung 3.2: Hierarchie mit LOD

präsentierte Detailgrad von Objekten in Abhängigkeit vom eingenommenen Anteil

des Sichtfeldes.

View-Frustum-Culling

Eine weitere Verbesserung ist die Kombination von Clipping und dem oben beschriebenen Level-Of-Detail-Ansatz. Das heißt, daß das Culling nicht nur den sichtbaren Teil einer Szene bestimmt, sondern auch nur dessen auflösbare Teile. Die Auflösbarkeit hängt natürlich auch von der Auflösung eines Anzeigegerätes ab.

Dies impliziert ein Bestimmen der sichtbaren Knoten des Baumes, wie in 3.2 zu sehen ist.

Damit das Clipping effizient ausgeführt werden kann, wird ein Hüllkörper genutzt, um zu testen, ob ein Objekt sich komplett innerhalb oder außerhalb des Sichtfeldes befindet.

Der Algorithmus steigt rekursiv den Baum herab, transformiert die Hüllkörper in das perspektivische Sichtkoordinaten System und testet die durch den Hüllkörper belegte Sichtfläche und den Schnitt mit den Grenzen des Sichtfeldes. Das Absteigen im Baum wird durch den Flächentest bestimmt, während die Inklusion oder Ablehnung durch einen Test des Hüllkörpers gegen das Viewfrustum ermittelt wird. Erst nachdem der Flächentest terminiert oder ein Blattknoten erreicht wurde, ist es notwendig, die Zeichenprimitive zu transformieren und zu clippen.

Dieser einfache Mechanismus, um den Detailgrad einer Szene zu variieren, legt noch einige andere Möglichkeiten nahe. Da dem Szenenzentrum meist das meiste Interesse entgegenkommt, könnte man die Szene mit einer Zentrumsgewichtung rendern. Das heißt das der Flächentest zur Peripherie hin gröber wird. Außerdem werden sich bewegende Objekte vom Auge weniger stark aufgelöst und können so in Abhängigkeit ihrer Geschwindigkeit gröber dargestellt werden.

Heutzutage bezeichnet man das Bestimmen der potentiell sichtbaren Elemente durch testen des View-Frustums gegen die Hüllkörper allgemein als View-Frustum-Culling.

Geometrie Caching

Da darzustellende Szenen häufig sehr komplex sind, muß sich um ein Speichermanagment gekümmert werden. Die Objekte, die in der Nähe des Sichtfeldes, innerhalb, oder in der “Nähe” der Auflösung des Bildes sind, bestimmen ein Arbeitsset. Clark schlägt vor, die Elemente dieses Arbeitssets in einem direkt zugreifbaren Speicher zu halten. Ein Least-Recently-Used-Algorithmus (LRU) kann sich um dessen Verwaltung kümmern. Die restlichen Objekte befinden sich in einem zweiten langsameren Speicher. Da meist die Unterschied von Frame zu Frame gering sind, ändert sich die Menge nur langsam. In den Fällen in dem die Unterschiede aufgrund einer schnellen Kamerabewegung groß sind, kann die Szene mit gerin-

gerem Detail dargestellt werden. Alle Hüllkörper müssen im direkt zugreifbaren Speicher vorhanden sein, um die Bestimmung des Arbeitsset zu ermöglichen.

Occlusion Culling

Die geometrische Hierarchie kann für einen weiteren Culling Ansatz verwendet werden. Große Teile der Struktur müssen nicht weiter betrachtet werden, wenn sie durch ein Objekt verdeckt sind. Dazu wird für jedes Objekt ein Occluder-Volumen Δ definiert, so daß falls dieses verborgen ist, auch das Objekt selbst nicht sichtbar ist. Desweiteren wird ein einfaches Occluder-Volumen δ für jedes Objekt definiert, so daß, falls δ etwas verbirgt, es sicher ist, daß dieses etwas durch das Objekt verborgen wird. Offensichtlich existiert Δ für jedes Objekt, wobei δ möglicherweise nicht für alle Objekte existiert, wie zum Beispiel ein Zylinder mit offenen Enden oder ein transparentes Objekt. Das Bounding Volume kann einfach für Δ genutzt werden. Für δ jedoch ist zusätzliche Information nötig die, für jedes Objekt gespeichert werden muß.

Diese Betrachtungen legen einen rekursiv absteigenden Algorithmus nahe, bei welchem auf jeder Ebene alle Objekte nach ihren Hüllkörpern sortiert werden. Falls irgendwelche Hüllkörper seitlich und vertikal überlappen, erlaubt der Verdeckungstest möglicherweise, daß eines oder mehrere Objekte und dessen Unterobjekte aus der Betrachtung entfallen.

Da dieser Algorithmus und der Clipping-Algorithmus rekursiv auf dem Baum absteigen, liegt es nahe, sie zu kombinieren. Dadurch können Flächentest für verborgene Objekte entfallen. Außerdem verkleinert sich so das Arbeitsset. Die vorgestellte Struktur bietet einen sinnvollen Weg, um den Detailgrad einer Szene zu variieren, entsprechend der Anzeigenfläche, die ein Szeneobjekt nutzt und der Geschwindigkeit mit welcher sich ein Objekt oder die Kamera bewegt. Außerdem schlägt Clark bequeme Arten vor, wie auf Objekte schnell zugegriffen werden kann, indem ein Arbeitsset genutzt wird. Dies ermöglicht Framekohärenz.

Im Folgenden werden die wichtigsten und bekanntesten Frameworks vorgestellt, die auch maßgeblich den Begriff des Szenengraphen mitbestimmt haben.

3.2 OpenInventor

OpenInventor (Strauss u. Carey (1992); Strauss (1993); Wernecke (1993)) ist ein Framework mit dessen Hilfe man 3D Szenen erstellen kann. OpenInventor bietet verschiedene Bausteine an, um interaktive Anwendungen zu erzeugen. Großes Augenmerk liegt hierbei auf der Interaktion und auf der Erweiterbarkeit.

OpenInventor ist objektorientiert. Das Hauptziel bei der Entwicklung von Inventor war es, das Erstellen von interaktiven 3D Anwendungen zu vereinfachen.

Dazu bietet es eine Menge von direkt manipulierbaren 3D Objekten an. Bei der Entwicklung hatte einfache Benutzbarkeit Priorität gegenüber Performanz.

Szenengraph

OpenInventor benutzt einen Szenengraphen. Dieser ist ein gerichteter, azyklischer Graph. Die Knoten (bzw. Nodes) des Graphen sind Container. Diese enthalten sogenannte Felder, die letztendlich die Daten eines Knoten speichern. Jeder Knoten definiert eine Anzahl von öffentlich zugänglichen Feldern. Mit jedem Feld ist ein Typ assoziiert. Die Felder bieten einen konsistenten Mechanismus zum Editieren, Anfragen, Lesen, Schreiben und Überwachen von Daten innerhalb eines Knotens.

Nodes

Inventor bietet eine Vielzahl von Knotentypen an. Zum Beispiel verschiedene *Shape Nodes*, welche 3D-Geometrien repräsentieren. Des weiteren *Property Nodes*, die diverse Eigenschaften wie Materialien, Zeichenstile, Transformationen und dergleichen beschreiben.

Eine Anzahl von Gruppenknotenklassen wird genutzt, um Knoten zu einem Graphen zu strukturieren. Diese bestimmen auch, wie die Kinder traversiert werden und wie Eigenschaften an diese vererbt werden. Seperatoren sind Gruppenknoten, welche bei der Traversierung den derzeitigen Traversierungs-State vor dem besuchen der Kinderknoten speichern. Wenn alle Kinder besucht wurden, wird der gespeicherte Zustand wieder hergestellt.

Die Position eines Lichtknotens bestimmt zwei Dinge. Erstens beleuchtet der Lichtknoten alles was auf ihn im Graphen folgt. Zweitens wird für den Lichtknoten durch die Position im Graphen seine Transformation bestimmt. Das ist problematisch, da man häufig diese beiden Dinge getrennt beschreiben möchte. Beispielsweise bewegen sich Autoscheinwerfer mit einem Auto und beleuchten die Straße, nicht aber das Auto.

Um diese Trennung zu ermöglichen, kann die Lichtposition zum Beispiel über eine *Field Connection* oder einen Sensor gesetzt werden. Ein Sensor ist ein Inventor Objekt, das bei bestimmten Ereignissen eine durch den Nutzer gegebene Funktion aufruft.

Ein Kameraknoten generiert ein Bild aller Objekte die auf ihn im Szenengraph folgen, weshalb er meist oben links im Szenengraphen platziert wird. Diese Position bestimmt auch seine Transformation. Diese Doppelbedeutung der Position im Graph führt zu ähnlichen Problemen wie bei Lichtknoten.

Actions

Um Operationen wie z.B. Zeichnen, Picking oder Hüllkörper-Berechnungen auf dem Szenengraph auszuführen, werden sogenannte *Actions* verwendet. Dazu können diese Action-Objekte den Szenengraph traversieren. Bei Gruppenknoten werden dessen Kinder der Reihenfolge nach besucht. Andere Knoten, wie zum Beispiel Materialknoten, modifizieren den State, so daß alle folgenden Geometrien mit diesem Material gezeichnet werden. Um diese Effekte auf Knoten einer Gruppe zu beschränken, kann man sogenannte Seperatoren nutzen, welche auch Gruppenknoten sind. Bevor die Kinderknoten besucht werden speichern Seperatoren den derzeitigen Zustand. Wenn alle Kinder besucht wurden, wird der gespeicherte Zustand wieder hergestellt. Abhängig von Action-Objekt und besuchtem Knoten wird eine Operation ausgeführt, die mit Hilfe einer zwei-dimensionalen Virtual Function Tabelle bestimmt wird. Durch diesen Double-Dispatch-Mechanismus ist es einfach, den Szenengraph um neue Knoten und Actions zu erweitern.

3D Event Model

Inventor implementiert ein 3D Event Model, um fensterspezifische Ereignisse, die durch Nutzer Aktionen generiert werden, zu prozessieren. Dazu werden die 2D Events in 3D Events übersetzt und durch eine Event Handling Action im Szenengraph verteilt. Jeder Knoten der Interesse an diesem Event hat, kann dieses Prozessieren und daraufhin anzeigen, daß es behandelt wurde. Die Graphen Traversierung wird gestoppt, sobald ein Knoten das Event behandelt hat.

Mithilfe des soeben beschriebenen Event Modells, können Manipulatoren wie z.B. ein Trackball und andere interaktive Knoten einfach in eine 3D Anwendung eingebaut werden.

Datenflußgraph

Um neben Geometrie und State auch Verhalten im Szenengraph abzukapseln, bietet Inventor Engines an. Engines besitzen, wie normale Knoten Felder, die ihren kompletten State ausdrücken. Zusätzlich haben Engines auch noch Ausgabefelder, die durch eine Evaluierungsfunktion berechnet werden, sobald sich ein Feld ändert. Die Felder einer Engine können mit anderen Felder des Szenengraphen verbunden werden. Diese Verbindungen werden *Field Connections* genannt. Sie kopieren den Wert eines Eingabefeldes in ein Ausgabefeld sobald sich die Eingabe ändert. Mit ihrer Hilfe läßt sich der Szenengraph zu einem Datenflußgraph verdrahten. Dieser erlaubt es einem Entwickler, komplexes animiertes Verhalten in den Szenengraph einzubringen.

3.3 OpenGL Performer

OpenGL Performer (Rohlf u. Helman (1994)) ist ein kommerzielles Programmier-Toolkit um performante, Multiprozessor fähige Grafik Anwendungen für Visualisierung, Virtual Reality und Echtzeit 3D Grafik Anwendungen zu entwickeln.

Performer bietet ein Software-Schicht namens *libpr* über OpenGL an, welche effizient Geometrie und State Informationen verwaltet. Darüber liegt die High-Level Bibliothek *libpf*, welche Klassen für eine Hierarchie und automatische Multiprozess Fähigkeiten hinzufügt. Diese Schichtung ist in Abbildung 3.3 dargestellt. Des weiteren nutzt Performer Multiprozessing für Pipelined Rendering, um die Performanz zu verbessern.

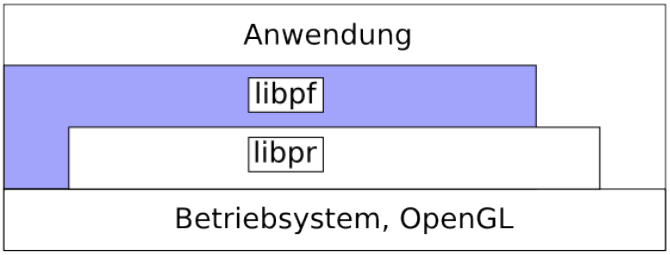

Abbildung 3.3: Bibliotheksschichten

Effizientes Rendering

Für performantes Zeichnen bietet Performer außerdem eine effiziente Geometrie an. Die Klasse `pfGeoSets` hat verschiedene Primitive, um Geometrie zu repräsentieren.

Der Datentransport zwischen CPU und GPU muß sorgfältig organisiert werden, da sonst die GPU auf die Applikation warten muß, was zu einer schlechten Rendering Performanz führt. Die Klasse `pfGeoSets` bietet spezielle Rendering Routinen für alle Kombinationen von Attribute Arrays, Attribute Bindings und Primitiven Typen an. Jede dieser Routine besteht aus einer spezifischen optimierten Rendering Schleife genau für diese Kombination. Dadurch bietet Performer optimale Rendering Performanz für eine große Mannigfaltigkeit von Geometrie Repräsentationen an.

Die Grafikstate-Anweisungen konfigurieren die GPU mit bestimmten Modi, wie beispielsweise das Shading Model oder Attribute, wie zum Beispiel Textur, was das Erscheinungsbild einer dargestellten Geometrie beeinflußt.

Performers Bibliothek *libpr* bietet spezielle Grafikprimitive und Grafikstate-Management an, die eine effiziente Nutzung der vorhandenen Hardware ermöglichen.

Datenbank Hierarchie und Traversierung

Über der *libpr* Schicht liegt die high-level Bibliothek *libpf*, welche eine Datenbank Hierarchie und automatische Multiprozess-Fähigkeiten zu *libpr* hinzufügt.

Der Performer Szenengraph ist ein gerichteter azyklischer Graph von Knoten. Die Geometrie Information befindet sich in den Blattknoten, während innere Knoten Konzepte, wie Gruppierung, Transformation, Sequenzierung, Level-of-Detail und Morphing, anbieten.

Datenbank Prozessierung findet mit Hilfe von Traversierungsoperationen des Szenengraphen statt. Typischerweise aktualisiert eine Anwendung den Szenengraph und die Sichtparameter für den nächsten Frame und initiiert dann einen oder mehrere Traversierungen, um das nächste Bild zu erzeugen. Die Bibliothek *libpf* bietet eine objektorientierte Programmierschnittstelle und ist in C++ implementiert.

Transformationen, die in `pfSCS` und `pfDCS` enthalten sind, werden bei der Traversierung auf einem Traversierungsstack akkumuliert und vor dem Zeichnen einer Geometrie angewendet. Jeder Subgraph definiert sein eigenes lokales Koordinatensystem.

Zusätzlich hat Performer eine Bounding-Volume-Hierarchie (BVH) Jeder Knoten hat eine Hüllkugel, die den Knoten sowie dessen Kinder umgibt. Diese Hüllkörper werden immer dann neu berechnet, wenn sich die Ausdehnung der Geometrie oder die Graphentopologie ändert. Diese Hüllkörperhierarchie erlaubt effiziente Schnitt- und Culling-Traversierungen

Stateattribute werden in Statesets verwaltet, die von Knoten referenziert werden. Diese Statesets sind nicht Teil der Hierarchie.

Nachdem die Anwendung den Szenengraph für den nächsten Frame modifiziert hat, können drei verschiedene Traversierungen angewendet werden.

Culling Traversal(CULL): sortiert Geometrie außerhalb der Sichtpyramide des Viewing Frustum aus, berechnet Level-Of-Detail-Switches (pfLOD) und sortiert Geometrie nach ihrem Grafikstate.

Drawing Traversal(DRAW): sendet Geometrie- und Grafikanweisungen mit Hilfe von OpenGL an die GPU.

Intersection Traversal(ISECT): prozessiert Liniensegment basierte Schnittanfragen für eine einfache Kollisionserkennung und Terrainverfolgung. Die Traversierung nutzt die Hüllkörperhierarchie, um den Schnittest zu beschleunigen.

Abhängig von Blickpunkt und Blickrichtung kann das View-Frustum-Culling einen Großteil der Geometrie einer Szene aussortieren und dadurch die Datenmenge, die zur GPU gesendet wird, reduzieren.

Die CULL Traversierung konvertiert den gecullten und sortierten Graphen in eine effiziente Display Liste, welche letztendlich den ganzen Frame enthält, und gibt diese an DRAW weiter. DRAW traversiert daraufhin diese Displayliste und sendet OpenGL-Kommandos an die Hardware. Aufgrund der Vorbearbeitung kann dies sehr schnell und effizient geschehen.

Multiprozessing

Performer nutzt ein Pipelining als Multiprozesschema, das auf Workstations mit wenigen Prozessoren ausgerichtet ist, im Gegensatz zu massiven parallelen Maschinen mit tausenden von Prozessoren. Die Arbeitsteilung beruht auf Prozessierungsstufen. Eine Stufe ist ein diskreter Abschnitt einer Prozessierungspipeline, die einer bestimmten Arbeit gewidmet ist. Performers Basisarbeitseinheiten sind die verschiedenen Traversierungen, ISECT, CULL, und DRAW. Konsequenterweise sind die verschiedenen Multiprozessingstufen um die verschiedenen Traversierungen gebaut. Zusammen mit einer weiteren Applikationstufe (APP) bilden sie zwei Arten einer Prozess-Pipeline, die *Rendering Pipeline* und die *Intersection Pipeline*

Die *Rendering Pipeline* besteht aus drei Stufen, der APP-, CULL- und DRAW-Stufe. Die APP-Stufe führt anwendungsspezifische Manipulationen des Szenengraph aus und gibt den resultierenden Szenengraph an die CULL-Stufe weiter. CULL generiert eine gecullte und sortierte Displayliste, die an die DRAW-Stufe weiter gereicht wird. Jede Stufe kann auf einem eigenen Prozessor laufen.

3.4 OpenSG

Eines der wichtigsten Features, welches in den zuvor vorgestellten Szenengraphen fehlt, ist ein Multi-Thread-sicherer Datenzugriff und sichere Datenbehandlung.

VR-Applikationen können multiple, asynchrone Threads besitzen, die einen konsistenten Zugriff auf Daten benötigen. Dies können beispielsweise neben dem Rendering noch eine haptische oder eine Physiksimulation sein, die mit verschiedenen Geschwindigkeiten laufen. Alle diese Threads brauchen konsistente Daten. Außerdem ist Multi Threading zur Nutzung neuer MultiCore CPUs sinnvoll. Ein Designziel von OpenSG (Reiners u. a. (2002)) ist Multi-Thread-sicherer Datenzugriff.

OpenSG erlaubt es, aufgrund seines Designs mehreren asynchronen Threads unabhängig von einander den Szenengraph zu manipulieren. Dazu müssen dessen Daten repliziert werden.

Da Szenengraphdaten sehr groß werden können, werden die Daten nur wenn nötig repliziert. Außerdem wird eine Unterscheidung zwischen der Graphenstruktur und den verwalteten Daten getroffen. Dazu trennt OpenSG die Daten und die Graphenstruktur durch Aufspalten der Szenengraphknoten. Die Klasse `Node` wird

zur Strukturierung des Graphen genutzt und Klassen vom Typ `NodeCore` enthalten die Daten.

Eine weitere erwähnenswerte Besonderheit ist die Positionierung von Lichtern und Kameras. Um Lichter und Kameras zu Positionieren und Orientieren wird ein beliebiger Knoten des Szenengraphens referenziert und dessen Transformation genutzt. Dieser Knoten wird Beacon genannt. Mit der Position eines Lichtknotens im Graphen gibt man an, daß der gesamte Subgraph beleuchtet ist.

Des Weiteren bietet OpenSG Reflektion an, um das ganze Konzept generisch und einfach erweiterbar zu halten.

Multi-Thread Safe Data

Damit jeder Thread nicht nur Zugriff auf den Szenengraph hat, sondern diesen auch ändern kann benötigt jeder eine private Datenkopie. Da Daten sehr groß werden können, ist es nicht möglich und auch nicht effizient, einfach alles zu kopieren. OpenSG nutzt wie Open Inventor ein Felderkonzept, um Daten abzukapseln. Dazu definiert OpenSG drei Klassen, die als Basis Klassen für alle Daten im System genutzt werden: `SingleField`, `MultiField` und `FieldContainer`.

Instanzen der Klasse `SingleField` sind das Äquivalent von einfachen Member Variablen. Sie speichern einen Wert eines zugehörigen Typs im Feld. Diese Felder werden direkt repliziert. Es wird angenommen, daß ihre Werte so klein sind, daß sie den Speicher durch eine Replikation nicht zu sehr belasten.

Die Klasse `MultiField` ist ein Array von Werten. Die Daten werden nur von einem Feld referenziert, aber außerhalb des Feldes gespeichert. Es wird angenommen, daß MultiFields den Hauptteil der Daten halten. Deshalb werden auch nur Referenzen auf die Daten repliziert und nicht die Daten selbst. Die Daten werden nur wenn nötig repliziert, beispielsweise wenn ein Thread empfangene Daten in seine eigene Kopie schreibt. Wenn zwei Threads mit eigenen Kopien ihre Daten synchronisieren, wird eine der Kopien verworfen und beide Threads referenzieren wieder die gleichen Daten bis einer von beiden sie wieder beschreibt.

Die Klasse `FieldContainer` kapselt Felder ab. OpenSG repliziert FieldContainer anstelle der einzelnen Felder. Mehrere Kopien eines FieldContainers liegen nacheinander im Speicher. Für den Zugriff werden eigene Pointer Objekte genutzt. Sie kennen die Startaddresse und die Größe eines Containers. Um nun auf die korrekte Kopie zugreifen zu können, wird einfach der Index des derzeitigen Threads genutzt.

Synchronisation

Für eine effektive Synchronisation signalisiert die Applikation explizit, wenn Daten geändert werden. Die Datenkopien müssen zwischen den Threads synchronisiert

werden, so daß Threads sich gegenseitig beeinflussen können. Dazu müssen die Daten von einem Thread zum anderen kopiert werden. Meist ändern sich nur kleine Teile des Szenengraphen. Dazu werden in einer Changelist der geänderte `FieldContainer` zusammen mit der Information, welche Felder sich geändert haben, gespeichert. Wenn zwei Threads ihre Daten synchronisieren wollen, werden nur die `FieldContainer` in der Changelist abgeglichen und da auch nur die angegebenen Felder.

Reflektion

OpenSG bietet Reflektion an, um Entwicklern das Hinzufügen von neuen Klassen und die Erweiterung bestehender zu vereinfachen. Dazu enthält jeder `FieldContainer` eine `Type` Klasse, die Informationen über diesen und all seine Felder hat. Diese Informationen sind dann zur Laufzeit abfragbar.

Erweiterbarkeit

OpenSG nutzt das Prototyp Pattern. Das Prototypobjekt ist eine Instanz der Klasse die erzeugt werden soll. Dieser liegt in einem zentralen und zugreifbaren Speicherbereich. Dies erlaubt es den Prototypen zur Laufzeit mit einer Instanz einer anderen Klasse zu ersetzen, die das gleiche Interface besitzt. Dadurch kann eine neue Klasse genutzt werden, die besser an eine neue Hardware angepaßt ist oder andere neue Entwicklungen implementiert.

Clustering

Da PCs immer mächtiger werden, ist es ein Trend, mehrere für eine größere Rechenleistung zu koppeln. Damit diese zusammen ein gemeinsames Bild erzeugen können, benötigen sie die gleichen Daten. Dieses Problem ist dem der Multi-Thread Synchronisation sehr ähnlich. OpenSG bietet dafür auch die Möglichkeit des Clusterings mit Hilfe eines Netzwerkes an.

3.5 Zusammenfassung

In diesem Kapitel wurden verschiedene Szenengraphen vorgestellt. Dazu wurden die jeweiligen Besonderheiten vorgestellt. Alle diese Szenengraphen beschränken sich auf eine Struktur, einen Graphen. Hauptsächlich ordnet man seine Szene räumlich, indem man nahe oder verbundene Objekte zusammen gruppiert. Dadurch kann man frühe Traversierungsentscheidungen treffen, wie zum Beispiel beim Culling. Wenn jedoch der Szenengraph hauptsächlich nach State sortiert ist, kann

unnötiges Culltesten mehrmals auftreten. Da States sich außerdem nicht so oft ändern sollte die Szene nicht jeden Frame neu sortiert werden.

Weitere Fragen treten zum Beispiel bei der Positionierung von Lichtquellen auf. Wird über die Position einer Lichtquelle im Graph angegeben, wo sie sich befindet oder was beleuchtet wird ? Dies wird von OpenSG beispielsweise mit einem Beacon-Konzept gelöst. Vielen modernen Spielen werden klassische Szenengraphen aufgrund der Anforderungen an das Rendering nicht gerecht, weshalb dort meist hybride Systeme verwendet werden. Es bietet sich an, für solche Formulierung diese Struktur aufzubrechen. Ein weiteres Ziel ist es Systeme zu entwickeln, welche die Beziehungen besser formulieren und reflektieren.

4 MultiSG

Im folgenden Kapitel werde ich einen Szenengraphenansatz beschreiben, der mehrere Strukturen zur Organisation von Szenendaten verwendet.

4.1 Die Idee

Es gibt keine perfekte Organisation für einen Szenengraphen, die gleichzeitig für Spatial, State, Semantische, und CPU-Betrachtungen optimal ist. Die Idee von Bar-Zeev (Bar-Zeev (2003)) ist es nun, die Beschränkung auf eine einzige Szenengraphenstruktur zu entfernen.

Hierbei gibt es eine Art Objektsuppe: die Szenendatenbank. Diese enthält Geometrien, Texturen, Lichter oder Transformationen. Dazu gibt es verschiedene Views, die Strukturen beziehungsweise Relationen auf diesen Daten bilden, wie zum Beispiel für Renderstate oder verschiedene Beziehungen, wie räumlich, semantisch, logische und applikationsspezifisch. Auf diesen unabhängigen und komplementären Strukturen arbeiten Algorithmen wie Culling, StateSorting oder Suchanfragen.

Eine Möglichkeit dies zu implementieren, besteht in der Trennung von Szenengraphen Knoten und den Objekten, die sie repräsentieren. Szenenobjekte referenzieren Knoten in verschiedenen Strukturen. Dieser „Knoten“-Teil eines Objektes wird hier Aspekt genannt. Zum Beispiel hat ein zeichenbares Objekt einen geometrischen und einen Renderstate-Aspekt. Im folgenden wird eine Implementierung des eben vorgestellten Ansatzes mit Ruby und OpenGL beschrieben. Die Skriptsprache Ruby (Matsumoto (1993)) wurde aufgrund ihrer Flexibilität gewählt und zum Rapid-Prototyping benutzt.

4.2 SceneObject

Die Elemente der Szene, wie zeichenbare Objekte, Kameras oder auch Lichter, sind vom Typ `SceneObject`. Jedes dieser Objekte hat verschiedene Aspekte, wie zum Beispiel einen Renderstate oder eine Geometrie, welche in den verschiedenen Hierarchien verknüpft sind. Die Abbildung 4.1 zeigt ein Klassendiagramm dieser Modellierung. Die Klasse `Drawable`, die von `SceneObject` abgeleitet ist, repräsentiert zeichenbare Objekte. Sie besitzt Aspekte im `SpatialView` und im `StateView`.

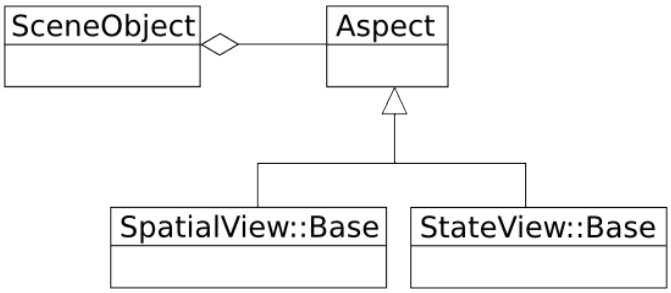

Abbildung 4.1: SceneObject

Die Klasse `Camera` hingegen besitzt nur einen Aspekt im `SpatialView`, da sie nicht gezeichnet wird, sondern zum Zeichnen genutzt wird.

4.3 SpatialView

Die Klassen für die räumliche Hierarchie sind in Abbildung 4.2 dargestellt. Mit Hil-

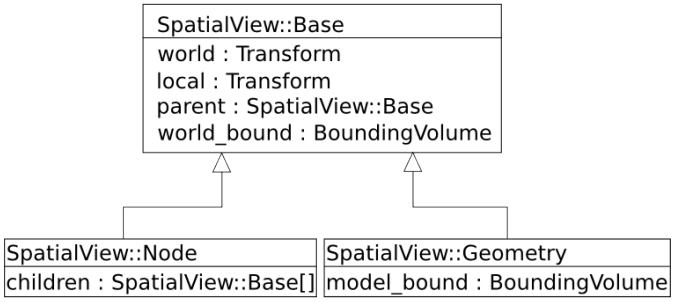

Abbildung 4.2: Klassendiagramm der SpatialView Klassen

fe der Transformationen können Objekte platziert, orientiert und skaliert werden. Die Hüllkörper werden für Culling und Picking verwendet. Die Klasse `SpatialView::Base` ist die abstrakte Basis Klasse für `SpatialView::Geometry` und `SpatialView::Node`. Die Klasse `SpatialView::Base` verwaltet eine Referenz auf den Vaterknoten, eine lokale Transformation, eine Welttransformation und ein Bounding Volume. Die `SpatialView::Node` Klasse repräsentieren innere Knoten der Hierarchie, daß heißt sie wird zur Gruppierung von Knoten genutzt. Sie verwaltet Referenzen auf eine Menge von Kinderknoten. Die Klasse `SpatialView::Geometry` repräsentiert die Blattknoten der Hierarchie, welche die Geometriedaten enthalten. Für die Geometriedaten verwaltet sie außerdem ein Bounding-Volume in Modellkoordinaten. Mit Hilfe dieser Knoten wird eine Hierarchie aufgebaut, die eine Transformations- und Bounding-Volume-Hierarchie in sich vereint.

4.3.1 Aktualisierung des SpatialView

Bei Änderung der lokalen Transformation eines Knotens der Hierarchie müssen die Welttransformationen, sowie die Hüllkörper aktualisiert werden. Dazu wird vom Einstiegspunkt ausgehend der Subgraph rekursiv besucht. Beim Absteigen werden die Welttransformationen aktualisiert. Beim darauf folgenden Aufsteigen wird die Größe der Hüllkörper in Weltkoordinaten entsprechend angepasst, so daß ein inneres Volumen die Volumina seiner Kinderknoten umfaßt. Anschließend wird vom Einstiegspunkt aus die Hüllkörperhierarchie bis zur Hierarchiewurzel aktualisiert. In Listing 4.1 ist der zugehörige Ruby-Quellcode zur Verdeutlichung aufgeführt.

```
def SpatialView::update_geometry_state(start)
  vis = update_visitor()
  depth_first_visit(vis, start)
  propagate_bound_to_root(vis, start)
end
```

Listing 4.1: Aktualisierung des SpatialView

Die Klassen des `SpatialView` und die Algorithmen zu dessen Aktualisierung sind an das Design von (Eberly (2001)) angelehnt.

4.3.2 Culling

Um nur die Potentiell sichtbaren Objekte zu zeichnen, wird die Klasse `CullingOperation` genutzt, welche ein View Frustum Culling ausführt. Die Sichtbarkeit wird ermittelt, indem die Hüllkörper gegen die Ebenen des Kamera Frustums getestet werden. Wenn ein Objekt als möglicherweise sichtbar bestimmt wurde, wird dieses abhängig davon, ob es transparent ist oder nicht, in einen Container für transparente Objekte oder in einen Container für opaque Objekte eingefügt.

4.4 StateView

Der StateView wird verwendet, um den Renderstate von Objekten zu beschreiben.

Die Klasse `StateView::Base` ist die abstrakte Basis Klasse für `StateView::State` und `StateView::StateModifier`.

Die Klasse `StateView::State` bildet die Blattknoten der Hierarchie und hält ein vollständiges Set von `StateAttribute`. Dieser Renderstate kann dann zum Zeichnen eines `SceneObject` genutzt werden.

Die inneren Knoten werden von der Klasse `StateView::StateModifier` repräsentiert. Sie können `StateAttribute` halten, mit denen sie den vorgegebenen Renderstate für ihren Subgraph modifizieren.

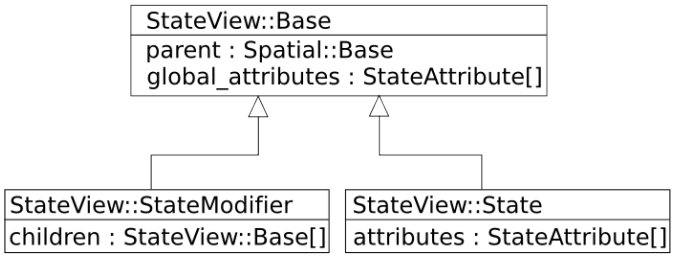

Abbildung 4.3: Klassendiagramm der StateView Klassen

4.4.1 StateAttribute

Folgende State-Attribute, die genutzt werden können, um das Erscheinungsbild von Objekten zu beeinflussen, sind implementiert.

Alpha: zur Konfiguration der Transparenz.

Depth: konfiguriert, ob und wie ein Tiefentest stattfindet.

Cull: bestimmt das Cullen von Polygonen, was zum Backface Culling genutzt wird.

Fog: bestimmt Parameter und Funktionen, um Nebel zu simulieren.

Material: spezifiziert Materialeigenschaften für OpenGL.

Polygonmode: bestimmt wie Polygone beim Rastern gezeichnet werden.

Polygonoffset: spezifiziert Skalierung und Einheiten für Tiefenversatz.

texture2D: spezifiert und konfiguriert eine zwei-dimensionale Textur.

textureCube: spezifiert und konfiguriert eine CubeMap.

4.4.2 Aktualisierung des StateView

Wenn Attribute gesetzt, entfernt oder modifiziert werden, muß der `StateView` aktualisiert werden.

Dazu wird die Klasse `UpdateRenderState` genutzt. Sie sammelt alle StateAttribute im Pfad von der Wurzel der Hierarchie bis zum Einstiegspunkt auf. Dann werden rekursiv vom Einstiegsknoten ausgehend im Subgraphen die Attribute aufgesammelt und die Renderstates in den Blattknoten aktualisiert. Dazu benutzt `UpdateRenderState` einen Stack der von den Attributen entsprechend modifiziert wird. In Listing 4.2 ist der zugehörige Ruby Quellcode zur Verdeutlichung aufgeführt.

```
def updateState(start)
  vis=update_visitor()
  apply_parent_states(vis,start)
  depth_first_visit(vis,start)
end
```

Listing 4.2: Aktualisierung des StateViews

4.4.3 Sortierung nach Renderstate

Um kostspielige und unnötige Änderungen des States beim Zeichnen zu vermeiden und um Transparenzen korrekt darzustellen, werden opaque Objekte nach Renderstate und transparente Objekte der Tiefe nach sortiert. Ein Sortierer führt eine Vorsortierung aller Objekte unter Nutzung des Stateview aus. Dazu führt der Sortierer einen Tiefensuche-Algorithmus aus geht zu den Kinderknoten, die ja den kompletten State eines Objektes beschreiben. Wenn es transparent ist wird es in einen Container für transparente Objekte eingefügt. Ansonsten wird das folglich opaque Objekt in einen Hash eingefügt. Im Hash werden zu jedem State der Wert eines Zählers gespeichert, der mit jedem neuen opaquen State um eins erhöht wird. So sind dann zum Schluß alle opaquen Objekte von 1 bis n durchnumeriert, wodurch eine Ordnung gegeben ist. Diese muß nur dann neu bestimmt werden, wenn Änderungen an der State Hierarchie vorgenommen werden.

Nachdem der Culler die sichtbaren Objekte bestimmt hat, übergibt er diese an den Sorter. Die Opaquen Objekte werden dann unter Ausnutzung des Hash sortiert. Die transparenten Objekte werden ins Kamerakoordinatensystem transformiert und entsprechend ihrer Entfernung zur Kamera von hinten nach vorne geordnet. Dann werden zunächst alle opaquen Objekte gezeichnet und danach alle transparenten.

Sonstige Funktionalität

Neben den erwähnten Objekten wurden verschiedene zweckmäßige Klassen implementiert. Es gibt eine Fensterabstraktion, eine `Viewport` Klasse die einen Bildausschnitt verwaltet. Außerdem werden einige primitive Objekte wie Kugel, Box und Ebene und die Möglichkeit, *OBJ* Dateien zu laden, angeboten. Um das entwickeln einfacher Anwendungen zu erleichtern, steht die Klasse `SimpleSceneManager` zur Verfügung deren Verwendung in Kapitel 6 illustriert wird.

4.5 Zusammenfassung

In diesem Kapitel wurde der MultiSG vorgestellt. Dieser addressiert Probleme die bei der Beschränkung auf eine Hierarchie entstehen. Dazu benutzt er im Gegensatz zu herkömmlichen Szenengraphen mehrere Hierarchien. MultiSG bietet eine Hierarchie zur Organisation von Renderstate und eine weitere Hierarchie zur räumlichen Organisation an. Im Kapitel 6 wird vorgestellt, wie der MultiSG zum rendern von Szenen genutzt wird.

5 ViewSG

Im Folgenden Kapitel wird eine View basierter Ansatz beschrieben. Dieser besteht aus drei Hauptkomponenten. Applikationen bestehen hierbei aus einer ungeordneten Menge von Elementen, welche die Szenendaten halten. Desweiteren hat eine Anwendung eine Menge von Views auf die Elemente der Szene. Views listen alle Elemente auf, die bestimmte vom View verlangte Attribute haben. Um Berechnungen auf diesen Daten auszuführen, wie beispielsweise ein Bild zu rendern, werden sogenannte Prozesse benutzt. Diese Objekte, arbeiten auf Views. Berechnungsabhängigkeiten zwischen Prozessen werden automatisch aufgelöst. Dazu wurde ein Push-Pull-Modell implementiert. Dieses Framework ist sehr flexibel und erlaubt es verschiedene Berechnungsvorgänge zu modellieren, bei denen Abhängigkeiten zwischen Rechenoperationen existieren. Hier wird es als Basis für ein Grafiksystem genutzt.

5.1 Element

Ein Element ist ein Container für eine Menge von Attributen. Alle Daten werden von diesen Attributen gehalten. Attribute sind Objekte die einen Namen und einen Wert haben. Daneben speichern sie noch, ob der Wert aktuell oder eine Neuberechnung nötig ist. Die Attributwerte enthalten Daten, wie zum Beispiel Transformationen oder Geometrie, die von Prozessen für Berechnungen genutzt werden. Außerdem können Attribute zusätzlich andere Elemente referenzieren. Dadurch können verschiedene Hierarchien dargestellt werden.

5.2 View

Für die Berechnungen, die nötig sind, um letztendlich ein Bild zu erzeugen, sind verschiedene Attribute der Elemente wichtig. Um beispielsweise die Welttransformation eines Szeneelementes zu berechnen, benötigt man dessen lokale Transformation und die Welttransformation des Vaterknotens in der Transformationshierarchie. Mit Hilfe eines Views trifft man eine Auswahl von Elementen, die bestimmte Attribute enthalten, zum Beispiel um nur Zugriff auf die Elemente der eben erwähnte Transformationshierarchie zu haben. Dazu werden die Attributnamen der Elemente genutzt. Bei Änderung der Elemente oder der Anfrage, werden

die Views neu erzeugt. Somit wird dynamisch eine Teilmenge aus der Menge aller Szenenelemente bestimmt. Dadurch erscheinen neu hinzugefügte Elemente automatisch in den entsprechenden Views. Dazu werden verschiedene an SQL angelehnte Operatoren genutzt. Mit Hilfe einer Select-Anweisung wird eine Auswahl aus der globalen Elementeliste getroffen. Diese Anweisung würde in SQL wie folgt aussehen. Die eigentliche Implementierung wurde in Ruby geschrieben und sieht daher etwas anders aus.

```
SELECT Auswahlliste [WHERE Where-Klausel]
```

Die Auswahlliste nennt die auszuwählenden Attributnamen, die ein Element enthalten muß, um im View zu erscheinen.

Die Attributnamen können bei der Auswahl zusätzlich noch umbenannt werden.

```
SELECT Attribut AS Alternativname
```

Beispiel:

```
SELECT
EnvMapFrontCamWorldTransformation AS WorldTransformation,
EnvMapFrontCamProjection AS ProjectionMatrix,
EnvMapFrontTexture AS Rendertarget
WHERE Name = MaterialKugel
```

Mit dem Mengenoperator

```
UNION
```

kann man die Ergebnisse vereinigen. Beispiel:

```
SELECT RenderTargetA AS RenderTarget
UNION
SELECT RenderTargetB AS RenderTarget
```

5.3 Process

Für Berechnungen mit Hilfe der Daten werden so genannte Prozesse benutzt. Diese arbeiten auf Views, das heißt auf Mengen von Elementen deren Attribute sie lesen und in die sie Berechnungsergebnisse schreiben können. Zu jedem Attribut kann man sich den Prozess vom System geben lassen, der es beschreibt oder berechnet. Jedes Attribut referenziert den Prozess, von dem es berechnet wird. Des weiteren referenziert jedes Attribut die Liste aller Prozesse, die dieses lesen. Intern benutzt das System bei Berechnungen ein Push-Pull-Modell.

Push-Pull-Modell

Um unnötige Berechnungen zu vermeiden und um Berechnungabhängigkeiten automatisch aufzulösen, wird ein Push-Pull-Modell verwendet.

Bei diesem wird das aktive Datensenden und Datenempfangen in einem Netzwerk bestehend aus Knoten und Kanten unterschieden. Beim Push wird das Senden von Daten an einem Knoten initiiert, welcher die Daten an alle adjazenten Knoten weiterleitet. Im Gegensatz dazu wird beim Pull an einem Knoten eine Datenanfrage gestartet, die an alle inzidenten Knoten propagiert wird.

Das Push-Pull-Model in Verbindung mit den Prozessen

Das Push-Pull-Modell wird benutzt, um Abhängigkeiten zwischen Berechnungsergebnissen aufzulösen. Mit Attributen, Views und Prozessen wird dabei ein Abhängigkeitsgraph aufgebaut.

Alle Attribute besitzen ein *dirty flag*, welches kennzeichnet, ob das Attribut valide ist oder nicht. Wenn ein Attribut geändert wird, kümmern sich die Prozesse, die es für Berechnungen benötigen, um die Invalidierung der abhängigen Attribute. Beispielsweise wird beim Setzen der lokalen Transformation eines Elementes deren Welttransformation und die Welttransformationen der darunter hängenden Transformationshierarchie invalidiert. Die Invalidierung geschieht durch das propagieren beziehungsweise pushen des *dirty flag*.

Beim Lesen des Wertes eines Attributs wird überprüft ob es valide ist. Ist es invalide berechnet der zugehörige Process aus seinen Eingaben den Wert des Attributs neu.

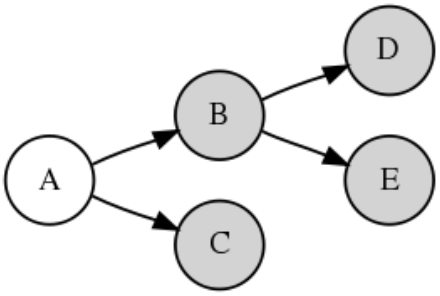

Abbildung 5.1: Attribut A ist aktualisiert

Eine schematische Darstellung ist in Abbildung 5.1 zu sehen. Valide Attribute sind als weiße und invalide als graue Kreise dargestellt. Im Beispiel ist Attribute **A** ist als valide gekennzeichnet.

Wenn nun Attribut **E** angefragt wird, erkennt es, daß es aktualisiert werden muß und läßt seinem Wert vom assozierten Prozess neu berechnen. Dieser fragt

daraufhin **B** an. Da **B** ebenso invalide ist, wird der Wert von **A** angefragt. Da **A** valide ist, kann nun der Prozess dessen Wert nutzen, um **B** neu zu berechnen. Dessen Wert kann nun der Prozess verwenden, um **E** zu berechnen und dadurch zu aktualisieren. Nach dem **E** berechnet wurde, sieht der Graph wie in Abbildung

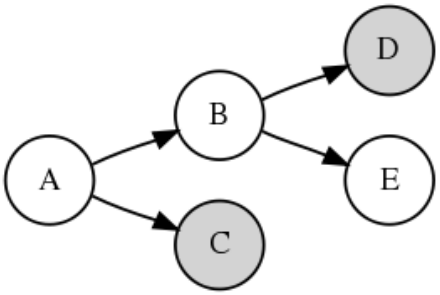

Abbildung 5.2: Attribut E wurde aktualisiert

5.2 aus.

Dazu implementiert ein Prozess die in 5.1 angegebenen Methoden. Die Neuberechnung eines Attributes wird mit `pull` initiiert. Die Propagierung des *dirty flag* wird mit `set_dirty` angestoßen.

```
class Process
  # propagate the dirty flag starting at element.attribute
  # by setting the attributes which are affected to dirty
  def set_dirty(element)
  end
  # actual compute
  def pull(element)
  end
end
```

Listing 5.1: Process Interface in Ruby

Die Klasse Attribute hat die Methode `get_value(v)`, `set_value` und `set_dirty()`. Die Funktion `get_value()` gibt den aktuellen Wert des Attributes zurück. Ist es nicht mehr aktuell, berechnet der zugehörige Prozess seinen Wert neu. Die Methode `set_value(v)` setzt den Wert des Attributes und benachrichtigt alle abhängigen Prozesse ihre abhängigen Attribute zu invalidieren.

5.4 Zusammenfassung

In diesem Kapitel wurde der ViewSG vorgestellt. Dieses flexible Framework erlaubt es verschiedene Berechnungsvorgänge zu modelieren, bei denen Abhängigkeiten zwischen Rechenoperationen existieren. Mit Hilfe von Views können Mengen

von Elementen ausgewählt werden. Prozesse die Berechnungen ausführen, arbeiten auf diesen Views. Falls durch Hinzufügen von Attributen zu Elementen, diese daraufhin einer Viewspezifikation entsprechen, tauchen sie automatisch in entsprechenden Views auf, wo sie von Prozessen, die auf diesen arbeiten, genutzt werden können. Um unnötige Berechnungen zu vermeiden und um Abhängigkeiten zwischen Prozessen automatisch aufzulösen, wird ein Push-Pull-Modell genutzt. Im Kapitel 6 wird vorgestellt, wie das System zum rendern von Szenen genutzt wird.

6 Use Cases

6.1 Zeichnen einer einfachen Szene

Als erstes Beispiel soll eine einfache Szene dienen. Sie besteht aus einer Ebene mit einem Material, einer Kugel mit Material, einer Punktlichtquelle und einer Kamera. Die Kugel wird in Bezug auf die Ebene platziert, wozu eine Transformationshierarchie verwendet wird. In Abbildung 6.1 ist diese Szene dargestellt.

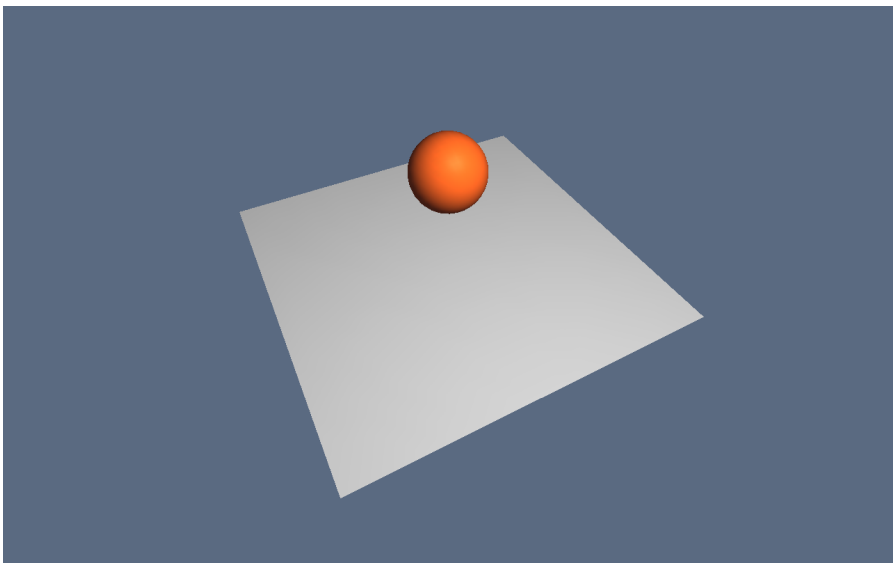

Abbildung 6.1: Einfache Szene

6.1.1 Realisierung mit MultiSG

Der MultiSG Szenengraph bietet eine Klasse `SimpleSceneManager`, die das Schreiben einfacher Anwendungen erleichtert. Sie bietet ein Interface an, um auf einen Wurzelknoten der SpatialView-Hierarchie und einen Wurzelknoten der StateView-Hierarchie zuzugreifen.

Außerdem bietet MultiSG einige einfache Objekte an, die von `Drawable` abgeleitet wurden. Die Klasse Drawable besitzt einen Aspekt `SpatialView::Geometry` und

einen Aspekt `StateView::State`, auf die jeweils mit den Akzessoren `:geometry` und `:state` zugegriffen werden kann. Für einige einfache Objekte wie Ebenen, Kugeln und Würfel gibt es schon fertige Klassen die von `Drawable` abgeleitet sind.

Szenedaten

Um nun die Szene in MultiSG zu beschreiben, wird zuerst eine Instanz des `SimpleSceneManager` angelegt.

```
ssm = SimpleSceneManager.new
```

Dieser bietet jeweils Zugriff auf die Wurzelknoten der SpatialView- und der StateView-Hierarchie. Außerdem bietet er eine vorkonfigurierte Kamera an.

Desweiteren wird eine Punktlichtquelle angelegt und konfiguriert, welche die Szene beleuchtet. Sie wird mit Parametern konfiguriert, die von OpenGL zusammen mit dem Material von Objekten, zu deren Shading benutzt werden.

```
light = Light.new do |l|
  l.ambient  = [0.5, 0.5, 0.5, 1.0]
  l.diffuse  = [1.0, 1.0, 1.0, 1.0]
  l.specular = [1.0, 1.0, 1.0, 1.0]
  l.position = [0.0, 10.0, 7.0, 1.0]
end
```

Listing 6.1: Erzeugen einer Punktlichtquelle

Der Code-Block zwischen `do` und `end` wird dem Konstruktor des Objektes übergeben und ausgeführt. Das zu konfigurierende Objekt wird dem Block dabei als Parameter, im Beispiel 6.1 als `l` bezeichnet, überreicht.

Nun werden die Ebene und die Kugel angelegt und jeweils mit einem Material konfiguriert. Die Objekte die ein Szeneobjekt in den verschiedenen Hierarchien referenziert, heißen Aspekte. Dieser Zusammenhang ist in **??** genauer erklärt. Der State-Aspekt wird in die StateView-Hierarchie eingefügt und der Geometry-Aspekt wird in die SpatialView-Hierarchie eingehängt. Außerdem wird angegeben von welchen Lichtern die beiden Objekte beleuchtet werden.

```
# Create a Sphere
Drawable::Sphere.new do |sphere|
  sphere.state.set_attribute(StateAttributes::Material.new(
                  :ambient => [0.35, 0.15, 0.1, 1.0],
                  :diffuse => [1.0, 0.4, 0.1, 1.0],
                  :specular => [0.2, 0.1, 0.1, 1.0],
                  :shininess => [40]))
  sphere.illuminated_by light
  ssm.state_root.attach_child sphere.state
```

```
end
```

Listing 6.2: Erzeugen der Kugel

Um die Ebene zu platzieren wird dann ein `SpatialView::Node` erzeugt, welcher an den vom `SimpleSceneManager` angebotenen Wurzelknoten der SpatialView-Hierarchie angehängt wird. Um die Kugel relativ zur Ebene zu Platzieren, wurde ein weiterer Knoten vom Typ `SpatialView::Node` erzeugt und an den Platzierungsknoten der Ebene angehängt.

```
plane_spatial = SpatialView::Node.new
sphere_spatial = SpatialView::Node.new do |node|
  node.local = Matrix.translate(0.0,3.0,0.0)
end

ssm.spatial_root.attach_child plane_spatial
plane_spatial.attach_child plane.geometry
plane_spatial.attach_child sphere_spatial
sphere_spatial.attach_child sphere.geometry
```

Listing 6.3: Erzeugen der räumlichen Hierarchie

Da die SpatialView-Hierarchie zur Positionierung verwendet wird, kann der `attach_child` Aufruf durch die aussagekräftigere Methode `transformed_by` ersetzt werden. Das gleiche gilt auch für Knoten des StateView, bei welchen die Methode `modified_by` heißt. Die vollständige Quellcodeauflistung der Applikation ist in A.1 zu sehen.

Zeichnen der Szene

Vor jedem Zeichnen wird der `SpatialView` durch einen Traversierer, wie in 4.3.1 beschrieben, aktualisiert. Das heißt, es werden die Welttransformationen und die Hüllkörper vom Wurzelknoten ausgehend rekursiv berechnet. Daraufhin berechnet der Culler die sichtbaren Objekte, indem die Sichtpyramide gegen die Hüllkörperhierarchie getestet wird.

Anschließend werden die sichtbaren Szeneobjekte einem Sortierer übergeben, der die opaquen Objekte nach State und die transparenten Objekte der Entfernung zur Kamera nach sortiert.

Zum Schluß werden zuerst die sichtbaren opaquen und dann die transparenten Objekte vom `Renderer` in der vom Sortierer gegebenen Reihenfolge gezeichnet.

6.1.2 Realisierung mit ViewSG

Zur Positionierung von Kugel und Ebene wird eine Transformationshierarchie genutzt, die mit Hilfe der Attribute dargestellt werden kann. In der Tabelle 6.2 sind

alle Elemente der Szene mit ihren Attributen und deren Typ aufgelistet. Des weiteren werden ebenfalls die Interfaces genannt, die jedem Element zugeordnet werden können. Ein Interface ist einfach ein Name für eine Menge von Attributen. Die Elemente und ihre Attribute sind hier aufgelistet.

Szenedaten

Um verschiedene Views zu erzeugen, auf denen dann die bildberechnenden Prozesse arbeiten, werden folgende Interfaces definiert. Jedes Interface listet die Attribute auf, die ein Element besitzen muß, um diesem zu entsprechen.

Drawables: Elemente, die dargestellt werden sollen, benötigen folgende Attribute.

Drawables
WorldTransformation
Geometry
Material
Lights

Die `WorldTransformation` dient zur Platzierung, die `Geometry` für die Form, die `Lights` zur Beschreibung von Lichtquellen und das `Material` zur Spezifizierung des Reflektionsverhaltens. Diesem Interface entsprechen im Beispiel Kugel und Ebene.

Cameras: Elemente die als Kamera agieren, benötigen eine Welttransformation zur Positionierung, Angaben zur Beschreibung der Kameraparameter, ein Rendertarget und ein Renderergebnis. Das Rendertarget ist hierbei ein Fenster oder ein Hardwarebuffer auf der Grafikkarte.

Cameras
WorldTransformation
FOV
Near
Far
ProjectionMatrix
Rendertarget
Renderresult

Diesem Interface entspricht folglich das Element Kamera in der Szene.

Transformables: Transformationen dienen zur Platzierung, Orientierung und Skalierung von Elementen. Um Elemente relativ zueinander platzieren zu können,

wird eine Transformationshierarchie benutzt. Die `LocalTransformation` eines Elementes platziert ein Element bezüglich des Vaters in der Transformationshierarchie, das heißt bezüglich des `TransformParent`. Die globale Positionierung beziehungsweise `WorldTransformation`, hängt von der eigenen lokalen Transformation und der seiner Vorgänger ab. Um Teil der Transformationshierarchie zu sein, benötigen Elemente die in der folgenden Tabelle angegebenen Attribute.

Transformables
WorldTransformation
LocalTransformation
TransformParent
TransformChildren

Diesem Interface entsprechen die Elemente Ebene, Kugel und Kamera.

Material: Mit Hilfe von Materialien wird beschrieben, wie ein Objekt mit Licht interagiert.

Materials
Ambient
Diffuse
Specular
Shininess

Lights: Die hier beschriebenen Lichter haben verschiedene Parameter, die OpenGL zusammen mit den Materialattributen eines Objektes zu dessen Shading verwendet. Die Position der Punktlichtquelle wird hier im Weltkoordinatensystem angegeben.

Lights
Ambient
Diffuse
Specular
Shininess
Position

In dem Listing 6.4 ist aufgezeigt, wie die Kugel konfiguriert wird.

```
# Die Elemente der Szene
materialebene = Element.new(
                  :name=>"materialebene",
```

```
                    :parameter => MaterialParameterDefault.new())

kugel =  Element.new(
                :name=>"Kugel",
                :world_transform => Transform.identity(),
                :local_transform => Transform.translate(1,2,0),
                :transform_parent => ebene,
                :transform_children => nil,
                :geometry => SphereGeometry.new(),
                :material => materialkugel,
                :graphicsState => DefaultState.new())
```

Listing 6.4: Erzeugen der Kugel

Zum Zeichnen der Objekte wird die aktuelle Welttransformation vom Renderprozess benötigt, so daß diese zuvor von einem Prozess aktualisiert werden muß.

Bei diesem Beispiel hängt der Prozess zum Rendern `Render` von `CalculateWorldTransformation` ab, da dieser die zum Zeichnen nötige Welttransformation aktualisiert.

Zeichnen der Szene

Zum Zeichnen der einfachen Szene wurden verschiedene Prozesse definiert. `CalculateWorldTransformation` aktualisiert das Attribut `WorldTransformation` und der Prozess `Render` aktualisiert das Attribut `Renderbuffer` eines `Rendertargets`.

Prozess CalculateWorldTransformation

Für ein Element P mit dem Transformationskind C ergibt sich die Welttransformation von C als das Produkt aus P's Welttransformation und C's lokaler Transformation.

Der Prozess invalidiert beim Setzen der lokalen Transformation eines Elementes die Welttransformation und die Welttransformation aller Transformationskinder durch Pushen eines *dirty flags*. Beim Pullen findet die eigentliche Berechnung statt. Diese wird rekursiv ausgeführt und ist nur dann nötig, wenn das *dirty flag* beteiligter Attribute gesetzt ist. Des weiteren müssen beim Hinzufügen von Transformationskindern alldiese invalidiert werden.

Prozess Render

Der Prozess `Render` verwaltet eine Menge von `Drawables` und eine Menge von `Camera` die jeweils über einen View gegeben sind. Bei einem Pull wird des `Renderresult` einer `Camera` eingerichtet. Dann werden alle Elemente in der Menge der `Drawables`

nach Bestimmung der Welttransformation, Geometrie und des Materials gezeichnet. Bei einem Push wird das `Renderresult` der `Camera` invalidiert.

6.2 Zeichnen einer Szene mit Spiegelung

Bei diesem Use Case wird eine Kugel gezeichnet, welche ihre Umgebung spiegeln soll. Diese Materialeigenschaft wird mit Hilfe einer Environmentmap simuliert, welche in diesem Fall eine Cubemap ist. Im Gegensatz zum Use Case in Abschnitt 6.1 sind zum Zeichnen mehrere Passes notwendig. Zur besseren Verdeutlichung wurden sechs rotierende Würfel zur Szene hinzugefügt. In Abbildung 6.3 ist diese Szene dargestellt.

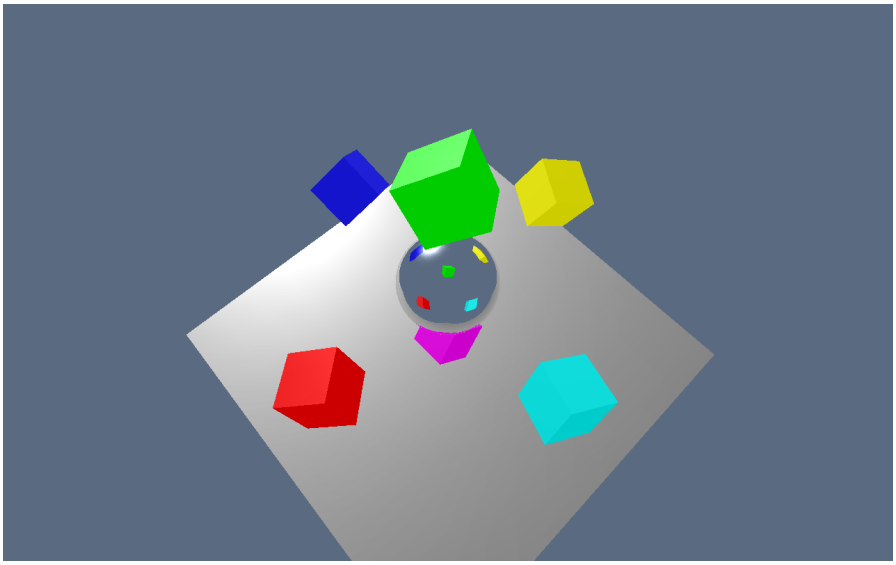

Abbildung 6.3: Szene mit Reflektion

6.2.1 Environment Mapping

Environment-Mapping, auch als Reflection-Mapping bezeichnet, ist eine einfache Technik, um Spiegelungen zu simulieren. Diese Technik wurde von Blinn und Newell (Blinn u. Newell (1976)) zum ersten Mal vorgestellt. Es wird angenommen, daß die reflektierten Objekte und Lichter sich weit entfernt vom reflektierenden Objekt befinden und das sich dieses selbst nicht reflektiert. Hier in diesem Beispiel wird das Cubic-Environment-Mapping (Akenine-Möller u. Haines (2002)) genutzt. Dazu werden sechs quadratische Texturen verwendet, welche die Seiten eines Würfels bilden. Jede Würfelseite steht für eine Menge von Richtungen entlang der Hauptachsen.

Um die Cubemap zu erzeugen, wird aus der Position der reflektierenden Kugel die Szene in positiver und negativer x-, y-, und z-Achse in eine Textur gerendert. In Abbildung 6.4 ist ein aufgefalteter Würfel mit der Cube-Map dieser Szene zu

sehen.

6.2.2 Realisierung mit MultiSG

Szenedaten

Da die Kugel eine Environment-Map nutzt, wird ihr Renderstate mit einem Texturattribut konfiguriert. Die genutzte Textur ist eine Cubemap.

Dann wird für den reflektierenden Teil der Szene ein `SpatialView::Node` und ein `StateView::Modifier` in die jeweiligen Hierarchien eingefügt. An diese beiden Knoten werden dann jeweils die Geometrien und States angehängt der sechs Würfel und Ebene angehängt. Die Berechnung der Cubemap aus Position der Kugel übernimmt die Klasse `CubeMapEffect`. Zur Initialisierung erhält die Klasse das reflektierende Objekt, einen `SpatialView::Node`, der die zu spiegelnde Szene hält, und einen `Renderer`. Diese Klasse verwaltet einen Hardwarebuffer, der mit der Cubemaptextur der Kugel konfiguriert wird. In diesen Buffer werden die sechs Würfelseiten jeden Frame hineingezeichnet.

Zeichnen der Szene

Die Applikation erlaubt es vor und nach dem Zeichnen Callbacks aufzurufen. Damit wird vor jeder Bildberechnung die Bestimmung der Cubemap angestoßen, so daß diese beim Zeichnen der Kugel immer aktuell ist.

```
ssm.on_frame_begin = lambda do |m|
  # animate cubes
  cubes_spatial.children.each do |c|
    c.local = c.local.rot_y(0.5).rot_x(0.6)
  end
  m.spatial_root.update
  # compute reflection map
  cubemapeffect.compute_reflection_map()
end
```

Listing 6.5: Callback vor dem Zeichnen

Die Berechnung der Cubemap wird von `CubeMapEffect` wie folgt ausgeführt. Zuerst wird das Sichtfeld und die Auflösung der in das Kugelzentrum verschobenen Kamera, angepaßt. Außerdem wird ein Hardwarebuffer aktiviert, so daß der Renderer nun statt in ein Fenster in diesen zeichnet.

Für alle positiven und negativen x-, y-, und z-Achsen findet nun folgendes statt. Zuerst wird der entsprechende Texturpart der Cubemap aktiviert und die Kamera wird in die entsprechende Achse orientiert. Daraufhin werden vom Culler die sicht-

baren Objekte bestimmt, die vom State-Sortierer dem Renderstate nach sortiert und vom Renderer anschließend gezeichnet werden.

Dann wird beim Zeichnen der Szene die Cubemap zur Darstellung der Kugel verwendet.

6.2.3 Realisierung mit ViewSG

Szenedaten

Für die rotierenden Würfel werden sechs Elemente, die eine Würfelgeometrie besitzen und dem Drawable Interface entsprechen, der in 6.1.2 beschriebenen Szene hinzugefügt.

Um spiegelnde Elemente kümmert sich der Prozess `SetupEnvironmentMap`. Dieser Prozess richtet Elemente ein, wenn sie neu in dem von ihm verwalteten View auftauchen, und stößt die Aktualisierung der Environmentmap an, wenn auf diese zugegriffen wird. Damit die Kugel von diesem Prozess behandelt wird, erhält das Element ein weiteres Attribut namens `:environmentmapped`.

```
envmapview = create_view(:name => "environmentmapped_view",
                         :select => [ :environmentmapped,
                                      :material,
                                      :world_transform])

setup_envmap_proc =
    ViewSG::Proces::SetupEnvironmentMap.new(envmapview)
```

Listing 6.6: SetupEnvironmentMap Prozess label

Prozesse

Wenn der Prozess `Render` ein Element zeichnet, schaut er beim Anwenden des Materials, ob das Element spiegeln soll, sprich das Attribut `:environmentmapped` besitzt. Falls ja wird ein Pull auf dem Attribut aufgerufen. Dies triggert den Prozess `SetupEnvironmentmap`.

SetupEnvMapProc

Beim erstmaligen Aufrufen fügt dieser Prozess als Texturattribut eine `CubeMapRenderTexture` zum Material des Elementes hinzu. Dieses Attribut erhält den Namen `:cubemap`.

Attribute die Kameras in der Position des Elementes beschreiben, werden dann für alle sechs Seiten des Würfels hinzugefügt.

Als Rendertarget der Kameras wird ein Objekt vom Typ `CubeRenderTexturePart`, welches eine Seite einer Cubemap ist, verwendet. Zum Beispiel für die Kamera in positiver x-Achse sehen die hinzugefügten Attribute wie in Abbildung 6.7 aus.

```
{
 :em_px_fov => 90.0, :em_px_near => 1.0, :em_px_far => 100.0,
 :em_px_width => 64, :em_px_height => 64,
 :em_px_projection_matrix => nil,
 :em_px_local_transform =>
              element.local_transform * px_rotation
 :em_px_world_transform => nil,
 :em_px_transform_parent => transform_root,
 :em_px_transform_children => [],
 :em_px_rendertarget =
              CubeRenderTexturePart.new(cubemap_rt, :pos_x)
 :em_px_renderresult => nil,
 :em_px_camera => nil # Tag zum auffinden
}.each{|k,v| material.add_attribute(k,v)}
]
```

Listing 6.7: Kamera für positive X-Achse

Ansonsten passt der Prozess die Transformation an, damit die Kamera auch im Objektzentrum und in die entsprechende Hauptachse orientiert ist.

Render

Dann wird für jede der sechs Seiten ein Kameraview erzeugt, welcher die Attribute umbenennt, so daß sie dem Kamerainterface entsprechen. Dies ist im Listing 6.8 zu sehen.

```
camview_pos_x =
          create_view(
            :name => "cameraview",
            :select => [
              :em_pos_x_world_transform,
              :em_pos_x_fov,
              :em_pos_x_near,
              :em_pos_x_far,
              :em_pos_x_projection_matrix,
              :em_pos_x_rendertarget,
              :em_pos_x_renderresult]
            :as => {
              :em_pos_x_world_transform    => :world_transform,
              :em_pos_x_fov                => :fov,
```

```
:em_pos_x_near              => :near,
:em_pos_x_far               => :far,
:em_pos_x_projection_matrix => :projection_matrix,
:em_pos_x_rendertarget      => :rendertarget,
:em_pos_x_renderresult      => :renderresult })
```

Listing 6.8: View Kamera in positiver X-Achse

Diese Views werden mit einem `union_view` vereint und dem Renderer zur Initialisierung mit übergeben.

```
envmap_cams_view = union_view(camview_pos_x, camview_neg_x,
                              camview_pos_y, camview_neg_y,
                              camview_pos_z, camview_neg_z)
```

Listing 6.9: Union der Kamera-Views

Um die Cubemap zu erzeugen, wird aus der Position der reflektierenden Kugel die Szene in positiver und negativer x-, y-, und z-Achse jeweils in eine Textur gerendert. Zum zeichnen wird der in 6.1.2 beschriebene Render-Prozess genutzt. Dieser erhält einen Cameraview, der aus den spiegelnden Elementen erstellt wird, und einen Drawablesview, welcher die zu spiegelnden Elemente aufführt.

Der Renderprozess wird durch `SetupEnvMapProc` initiiert, indem dieser auf das `:renderresult` Attribut der Cubemap-Kameras zugreift.

Attributnamen	Type	Interface
Ebene		
Name WorldTransformation LocalTransformation TransformParent Geometry Renderstate Material	String Transformation Transformation ElementReferenz Geometry Renderstate ElementReferenz	Drawables and Transformable
MaterialEbene		
Name Parameter	String Parameter	Material
Kugel		
Name WorldTransformation LocalTransformation TransformParent Geometry Renderstate	String Transformation Transformation ElementReferenz Geometry Renderstate	Drawable and Transformable
MaterialKugel		
Name Parameter	String Parameter	Material
Kamera		
Name WorldTransformation LocalTransformation TransformParent TransformChildren ProjectionMatrix Rendertarget Renderresult	String Transformation Transformation ElementReferenz Array Matrix RenderTarget Bool	Camera and Transformable

Abbildung 6.2: Elemente der einfachen Szene

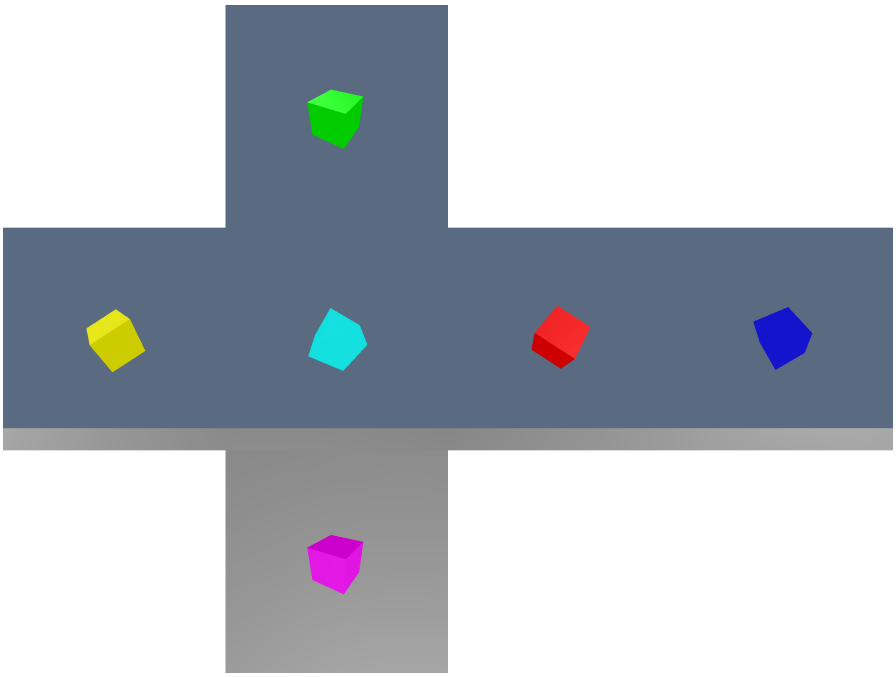

Abbildung 6.4: Cubic-Environment-Map

7 Vergleich und Diskussion

In diesem Kapitel werden die vorgestellten Ansätze bezüglich Erweiterbarkeit und Benutzbarkeit verglichen.

7.1 Benutzbarkeit

MultiSG ist in weiten Teilen wie ein klassischer Szenengraph zu nutzen. Der Hauptunterschied ist das Szeneobjekte auf mehrere Knoten verweisen können. Das bedeutet das der Nutzer diese Knoten in die jeweilige Hierarchie einzuhängen hat damit das System erwartungsgemäß arbeitet. Das erleichtert aber das Aufbauen des Graphen, da nicht überlegt werden muß, ob man nach logischen Zusammenhang, räumlicher Nähe oder Materialeigenschaften gruppiert, sondern alle Vorteile hat. Außerdem haben die entstandenen einzelnen Hierarchien meist geringere Tiefen als wenn alle Knoten in einem Graphen verwaltet werden.

Außerdem haben die entstandenen einem meist geringere Tiefen im Vergleich zu klassischen die alle Knoten in einer Hierarchie verwalten. Die Abfolge von Szenengraphenoperationen ist wie üblich explizit anzugeben.

Die Benutzung von ViewSG unterscheidet sich von der üblicher Szenengraphen. Um ein Element für einen bestimmten Zweck nutzen zu können, werden einfach benötigte Attribute hinzugefügt. Views werden genutzt um Elemente für Prozesse auszuwählen. Wenn ein Element einem gewissen View entspricht, taucht es automatisch in diesem auf und muß nicht per Hand hinzugefügt werden. Dadurch wird das Element automatisch von dem Prozess, der mit diesem View assoziiert ist, bearbeitet. Die Abfolge von Szenengraphenoperationen muß auch nicht explizit angegeben, da durch die Assoziationen von Attributen mit Prozessen ein Abhängigkeitsgraph gegeben ist, der automatisch auf Attributanfrage aufgelöst wird. Dadurch das die Attribute nur bei Anfrage neu berechnet werden, ist dieser Vorgang auch performant.

ViewSG

7.2 Erweiterbarkeit

Szenengraphen werden durch neue Knoten und neue Traversierer erweitert. Dazu wird üblicherweise ein Double-Dispatch-Mechanismus genutzt. Hierbei wird

die auszuführende Methode in Abhängigkeit vom Traversierertyp und besuchten Knoten bestimmt. Dieser Mechanismus wird ebenfalls beim MultiSG zur Erweiterung genutzt. Außerdem bieten die Szeneobjekte die Möglichkeit zur Laufzeit neue Aspekte hinzuzufügen, das heißt einen neuen Knoten einer Hierarchie zu referenzieren.

Beim ViewSG sieht das etwas anders aus.Hier arbeiten Prozessklassen auf homogenen Views. Da Views mit einer Auflistung bestimmter Attribute beschrieben werden, erfüllen alle Elemente eines Views, die Anforderungen des zugehörigen Processes. Dadurch ist im allgemeinen kein Double-Dispatch nötig. Zur Erweiterung werden neue Attribute-Typen und Prozesse entwickelt. Bei der Entwicklung eines Process muß dazu spezifiziert welche Attribute er nutzt. Bei der Initialisierung eines Prozesses wird der übergebene View auf Kompatibilität getest. Wenn der Test erfolgreich war, registriert der Prozess sich bei allen Attributen die er liest und schreibt. Um neue Daten hinzuzufügen, werden einfach neue Attributeklassen implementiert. Außerdem sind die Elemente des ViewSG mit beliebigen Attributen zur Laufzeit erweiterbar.

Der MultiSG kann einfach um weitere Hierarchien erweitert werden, um neue Beziehungen zu formulieren. Dazu wird zu wird die neue Knotenhierarch von `Aspect` abgeleitet, so das Szeneobjekte die Knoten dieser neuen Hierarchie referenzieren können.

Neue Hierarchien können beim ViewSG mit Hilfe von Attributen, die andere Elemente referenzieren, aufgebaut werden.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurden Szenegraphen und alternative Konzepte zur Strukturierung der Szenendaten mit Augenmerk auf Rendering untersucht. Zuerst wurden Szenengraphen allgemein vorgestellt und erklärt, wie mit ihnen Grafikszenen beschrieben und dargestellt werden können. Mit Hilfe von Szenegraphen können Objekte nach verschiedenen Gesichtspunkten wie räumlichen Nähe, gemeinsame Materialien und logische Zusammenhänge, geordnet werden. Da diese Betrachtungen nicht in einer Struktur vereinbar sind, wurden alternative Konzepte untersucht.

Mit Hilfe des MultiSG wurde aufgezeigt, daß die Hierarchie des Szenengraphen aufgebrochen werden kann. Dazu wurde er in eine Räumliche- und eine Renderstate-Hierarchie zerlegt. Er bietet die Funktionalität, die von klassischen Szenegraphen erwartet werden. Dazu wurde Aktualisierung der räumlichen Hierarchie, die eine Transformations- und Hüllkörperhierarchie umfaßt, implementiert. Außerdem wurde ein Culling-Traversierer der potentiell sichtbare Objekte programmiert. Als Zweites wurde eine Renderstate-Hierarchie implementiert, wodurch man eine explizite Statesortierung durchführen kann, indem Objekte mit gemeinsamen Renderstates zusammen gruppiert werden. Mit Hilfe der durch die Statehierarchie gegebenen Ordnung können Objekte zur Minimierung von Statewechseln gezeichnet werden, so daß unnötiges Sortieren entfällt, solange sich die Topologie der Hierarchie nicht ändert. Das Culling bestimmt sichtbare Objekte, die unter Ausnutzung der durch die Statehierarchie gegebenen Ordnung mit wenigen State-Änderungen gezeichnet werden können.

Als weiteres Konzept wurde der ViewSG entwickelt. Dieser ist ein Allgemeines Framework mit welchem sich Prozessabhängigkeiten modellieren lassen. Dazu wird ein Abhängigkeitsgraph automatisch aufgebaut. Durch Benutzen von durch Suchanfragen spezifizierten Containern können Objekte automatisch sortiert und ihre Werte von Prozessen, die auf Views arbeiten, berechnen lassen. Durch Nutzen des Push-Pull-Models werden unnötige Berechnungen vermieden und nur auf Anfrage gestartet. Durch das sehr allgemeine Konzept sind sie erweiterbar.

Anschließen wurden Testfälle implementiert. Mit diesen wurde gezeigt, wie die beiden Prototypen zum Rendering genutzt werden können. Das Ergebnis zeigt, daß es möglich und sinnvoll ist die Hierarchie aufzuspalten. Mit dem ViewSG wurde außerdem gezeigt wie die Ablaufkoordination von Berechnungen automatisch bestimmt wird.

Da eine Auftrennung möglich ist, besteht die Möglichkeit weitere sinnvolle Struk-

turen zu nutzen. Interessant wäre es beispielsweise eine semantische Hierarchie und eine Anwendungshierarchie einzuführen. Die semantische Hierarchie bildet eine Art Wörterbuch um auf Daten zuzugreifen. Dadurch können logische zusammenhängende Objekte ausgewählt werden. Mit Hilfe der Applikationsschicht könnte Verhalten und Controlling für Animation und Interaktion beschrieben werden. Dazu könnten Controller genutzt werden, die Werte der Szeneobjekte und deren Aspekte verändern und animieren. Außerdem wäre zu Untersuchen, wie die beiden Konzepte kombiniert werden können.

Eine Erweiterung um Multi-Threading wäre auch sinnvoll. In modernen VR Anwendungen wird neben Rendering auch noch Haptik und Physik genutzt. Haptische Simulationen beispielsweise laufen im kHz-Bereich um akzeptable Ergebnisse liefern zu können. Die Physiksimulation im Gegensatz, läuft aufgrund der Komplexität der Berechnungen mit geringerer Framerate. Unterstützung von Multi-Threading ist außerdem sinnvoll um Multi-Core CPUs effektiv nutzen zu können. Hier kann das multithread-sichere Design von OpenSG als Vorbild dienen.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bonn, 30. November 2007 _____

Unterschrift

Glossar

Clipping	(v. engl. to clip = abschneiden, abtrennen) Clipping bezeichnet die Begrenzung von Grafikprimitiven auf einen bestimmten Bereich um nur deren tatsächlich sichtbare Teile zu zeichnen.
CubeMap	(v. engl. Cube = Würfel, v. engl. Map = Abbild) besteht aus sechs zwei-dimensionalen Texturen die wie ein Würfel angeordnet sind. Wird zumeist genutzt um Umgebungsspiegelungen auf Objekten zu simulieren. Kann ebenfalls zur Berechnung von Per-Pixel Normalen verwendet werden.
Culling	(v. engl. to cull = aussortieren) Culling bezeichnet den Prozess der feststellt ob ein Objekt nicht sichtbar ist. Das meistgenutzte Verfahren testet dabei das View Frustum gegen den Hullkorper eines Objektes. Dieses wird als View Frustum Culling bezeichnet.
Depth Buffer	(v. engl. Depth Buffer = Tiefenpuffer) bezeichnet einen Hardwarepuffer, der die Tiefenwerte fur gezeichnete Pixel halt.
Environment Mapping	(v. engl. Environment = Umgebung, Mapping = Abbilden), bezeichnet eine einfache Technik um Spiegelung zu simulieren.
Field-Of-View (FOV)	(v. engl. Field-Of-View (FOV) = Sichtfeld) bezeichnet das Sichtfeld und in der Optik den Bildwinkel einer Kamera.
Least Recently Used (LRU)	(v. engl. Least Recently Used = "Am längsten nicht verwendet ") bezeichnet einen Optimierungsalgorithmus der aus einer Menge das Element entfernt, dessen Verwendung zeitlich am längsten zurückliegt.
Licht	reprasentiert eine Lichtquelle. In der Computergrafik werden zumeist Ambiente-, Direktionale, Spot- und Punktlichtquellen aus Performanzgrunden verwendet.
Materialien	Objekte in der Szene haben Materialien, deren Attribute zusammen mit der Beleuchtung ihr Aussehen bestimmen.
Renderer	bezeichnet in dieser Arbeit das Objekt welches ein Objekt mit einem vorhandenen Renderstate zeichnet. Der Renderstate bezeichnet den Zustand des Grafiksubsystems.
Shading	(v. engl. to Shade = schattieren) bezeichnet die Simulation und Darstellung von Objektoberflächen.
View Frustum	(v. engl. View Frustum = Sichtpyramide, Sicht-

Literaturverzeichnis

Akenine-Möller u. Haines 2002
AKENINE-MÖLLER, Tomas ; HAINES, Eric: *Real-Time Rendering*. Natick, MA, USA : A. K. Peters, Ltd., 2002. – ISBN 1568811829

Bar-Zeev 2003
BAR-ZEEV, Avi: *Scenegraphs: Past, Present and Future*. `http://www.realityprime.com/articles/scenegraphs-past-present-and-future`. Version: 2003

Blinn u. Newell 1976
BLINN, James F. ; NEWELL, Martin E.: Texture and reflection in computer generated images. In: *Commun. ACM* 19 (1976), Nr. 10, S. 542–547. `http://dx.doi.org/http://doi.acm.org/10.1145/360349.360353`. – DOI http://doi.acm.org/10.1145/360349.360353. – ISSN 0001–0782

Clark 1976
CLARK, James H.: Hierarchical geometric models for visible surface algorithms. In: *Commun. ACM* 19 (1976), Nr. 10, S. 547–554. `http://dx.doi.org/http://doi.acm.org/10.1145/360349.360354`. – DOI http://doi.acm.org/10.1145/360349.360354. – ISSN 0001–0782

Eberly 2001
EBERLY, David H.: *3D Game Engine Design*. San Francisco, CA, USA : Morgan Kaufmann Publishers, 2001. – ISBN 1558605932

Gamma 2002
GAMMA, Erich: *Design Patterns: elements of reusable object-orientated software*. Addison-Wesley, 2002

Matsumoto 1993
MATSUMOTO, Yukihiro: *Ruby*. `http://www.ruby-lang.org/`. Version: 1993

Reiners u. a. 2002
REINERS, D. ; VOSS, G. ; BEHR, J.: *OpenSG: Basic concepts*. `citeseer.ist.psu.edu/reiners02opensg.html`. Version: 2002

Rohlf u. Helman 1994
ROHLF, John ; HELMAN, James: IRIS performer: a high performance multiprocessing

toolkit for real-time 3D graphics. In: *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1994. – ISBN 0–89791–667–0, S. 381–394

Strauss 1993

Strauss, Paul S.: IRIS Inventor, a 3D graphics toolkit. In: *SIGPLAN Not.* 28 (1993), Nr. 10, S. 192–200. `http://dx.doi.org/http://doi.acm.org/10.1145/167962.165889`. – DOI http://doi.acm.org/10.1145/167962.165889. – ISSN 0362–1340

Strauss u. Carey 1992

Strauss, Paul S. ; Carey, Rikk: An object-oriented 3D graphics toolkit. In: *SIGGRAPH Comput. Graph.* 26 (1992), Nr. 2, S. 341–349. `http://dx.doi.org/http://doi.acm.org/10.1145/142920.134089`. – DOI http://doi.acm.org/10.1145/142920.134089. – ISSN 0097–8930

Tramberend 1999

Tramberend, Henrik: Avocado: A Distributed Virtual Reality Framework. In: *VR '99: Proceedings of the IEEE Virtual Reality*. Washington, DC, USA : IEEE Computer Society, 1999. – ISBN 0–7695–0093–5, S. 14

Wernecke 1993

Wernecke, Josie: *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1993. – ISBN 0201624958

A Quellcode Beispiele

Hier ist aufgelistet wie die in 6.1 in MultiSG und mit ViewSG implementiert ist. Beide Beispiele sind vollständige Applikationen.

A.1 Einfache Szene mit dem MultiSG

```
ASG::SimpleSceneManager.new do |ssm|

  ssm.mouse_controller.reset
  ssm.mouse_controller.rotate(100, 100)
  ssm.mouse_controller.zoom = -100
  ssm.camera.set_frame_by_viewmatrix(ssm.mouse_controller.matrix)

  light = Light.new do |l|
    l.ambient  = [0.5, 0.5, 0.5, 1.0]
    l.diffuse  = [1.0, 1.0, 1.0, 1.0]
    l.specular = [1.0, 1.0, 1.0, 1.0]
    l.position = [0.0, 10.0, 7.0, 1.0]
  end

  plane_spatial = SpatialView::Node.new do |n|
    n.transformed_by ssm.spatial_root
  end
  # Einfuegen in die Transformationshierarchie ist auch
  # wie folgt spezifizierbar
  #ssm.spatial_root.attach_child plane_spatial


  # Create a Plane
  Drawable::Plane.new do |g|
    g.state.set_attribute(StateAttributes::Material.new(
                    :ambient =>  [ 0.2, 0.2, 0.2, 1.0 ],
                    :diffuse => [ 0.5, 0.5, 0.5, 1.0 ],
                    :specular => [ 0.2, 0.2, 0.2, 1.0 ],
                    :shininess => [ 0.0 ]))
```

```
    g.illuminated_by light
    g.geometry.transformed_by plane_spatial
    g.state.modified_by ssm.state_root
  end

  sphere_spatial = SpatialView::Node.new do |n|
    n.local = Matrix.translate(0.0,3.0,0.0)
    n.transformed_by plane_spatial
  end

  # Create a Sphere
  Drawable::Sphere.new do |c|
    c.state.set_attribute(StateAttributes::Material.new(
                  :ambient => [0.35, 0.15, 0.1, 1.0],
                  :diffuse => [1.0, 0.4, 0.1, 1.0],
                  :specular => [0.2, 0.1, 0.1, 1.0],
                  :shininess => [40]))
    c.illuminated_by light
    c.geometry.transformed_by sphere_spatial
    c.state.modified_by ssm.state_root
  end

end.run
```

A.2 Einfache Szene mit dem ViewSG

```
# Light
light = create_element( :name => "light",
                    :ambient  => [0.5, 0.5, 0.5, 1.0],
                    :diffuse  => [1.0, 1.0, 1.0, 1.0],
                    :specular => [1.0, 1.0, 1.0, 1.0],
                    :position => [0.0, 10.0, 7.0, 1.0])

# Materials
gray_material = create_element( :name => "gray",
                    :ambient =>  [ 0.2, 0.2, 0.2, 1.0 ],
                    :diffuse => [ 0.5, 0.5, 0.5, 1.0 ],
                    :specular => [ 0.2, 0.2, 0.2, 1.0 ],
                    :shininess => [ 0.0 ])

orange_material = create_element(:name => "orange",
```

```
                    :ambient => [0.35, 0.15, 0.1, 1.0],
                    :diffuse => [1.0, 0.4, 0.1, 1.0],
                    :specular => [0.2, 0.1, 0.1, 1.0],
                    :shininess => [40])

# Transform Root
transform_root = create_element(:name => "transform_root",
                    :local_transform =>
                      Matrix.translate(0.0,0.0,0.0),
                    :world_transform => nil,
                    :transform_parent => nil,
                    :transform_children => []
                    )

# Plane
plane = create_element( :name => "plane",
                    :world_transform => nil,
                    :local_transform =>
                      Matrix.translate(0.0,0.0,0.0),
                    :transform_parent => transform_root,
                    :transform_children => [],
                    :geometry =>
                      ViewSG::PlaneGeometry.new(10.0, 20.0),
                    :material => gray_material,
                    :lights => [light]
                    )

# Sphere
sphere = create_element(:name => "sphere",
                    :world_transform => nil,
                    :local_transform =>
                      Matrix.translate(0.0,3.0,0.0),
                    :transform_parent => plane,
                    :transform_children => [],
                    :geometry =>
                      ViewSG::SphereGeometry.new(1.0,30.0,30.0),
                    :material => orange_material,
                    :lights => [light]
                    )

# create a View for all Transformable Objects
transformview = create_view(:name => "transformview",
                    :select => [:world_transform,
```

```
                                  :local_transform,
                                  :transform_parent,
                                  :transform_children])

calc_world_proc =
  ViewSG::Process::CalculateWorldTransformation.new(transformview)

kamera_transform = Matrix::look_at(Vector[0.0, 10.0, 10.0, 1.0],
                                   Vector[0.0, 0.0, 1.0, 1.0],
                                   Vector[0.0, 1.0, 0.0, 0.0]
                                  ).inv.transpose

kamera = create_element(:name => "kamera",
                        :fov => 60.0,
                        :near => 1.0,
                        :far => 1000.0,
                        :width => 800,
                        :height => 600,
                        :projection_matrix => nil,
                        :local_transform => kamera_transform,
                        :world_transform => nil,
                        :transform_parent => transform_root,
                        :transform_children => [],
                        :rendertarget =>
                          ViewSG::RenderWindow.new( 800, 600,
                                                    :double,
                                                    :rgb, :depth),
                        :renderresult => nil,
                        :main_camera => nil
                       )


renderview = create_view(:name => "renderview",
                         :select => [:world_transform,
                                     :geometry,
                                     :material])

cameraview = create_view(:name => "cameraview",
                         :select => [:world_transform,
                                     :fov, :near, :far,
                                     :projection_matrix,
                                     :rendertarget,
                                     :renderresult])
```

```
render_proc = ViewSG::Process::Render.new(cameraview, renderview)

display = lambda do
  cam ||= ViewSG::View::views.find do |view|
    view.name == "cameraview"
  end.first
  cam.local_transform = cam.local_transform.rot_x(0.01)
  cam.get_value_of(:renderresult)
end

keyboard = lambda do |key, x, y|
  case (key)
   when ?q
     exit
  end
end

GLUT.DisplayFunc(display)
GLUT.IdleFunc(display)
GLUT.KeyboardFunc(keyboard)

GLUT.MainLoop();
```