Vergleichende Untersuchung von szenengraph-basierten Renderingsystemen.

Studienarbeit

vorgelegt von
Mona Gerus

Institut für Computervisualistik
Arbeitsgruppe Computergrafik

Betreuer und Prüfer: Prof. Dr. Stefan Müller

Oktober 2003

Inhaltsverzeichnis

1. Einführung

1.1. Motivation

3D Graphiken sind für die wissenschaftliche Visualisierung und VR unumgänglich. Die Programmierung von 3D Graphikapplikation war seit jeher eine zeitaufwendige Aufgabe, die dem Programmierer gewaltiges Know How abverlangte. Um diese schwierige Aufgabe zu vereinfachen, haben Entwickler seit Anbeginn der Computergraphik immer wieder abstrahierende Funktionen geschaffen, die über den elementaren Graphikbefehlen liegen.
Bei diesen einfachen Lösungen kam aber immer die Interaktivität zu kurz. Die Anwendungen waren viel zu spezialisiert, um ein einfaches Manipulieren der Graphikdaten zu ermöglichen.
Ein echtes interaktives Toolkit sollte also eine gewisse Menge an 3D-Objekten unterstützen, die dann aber auch einfach manipuliert werden können sollten. Mit „einfach“ ist gemeint, dass der Anwender im gleichen Fenster, in dem er seine Graphikszene sieht, auch seine Veränderungen durchführen kann.
Zur effizienten, computergraphischen Darstellung komplexer Szenen werden vor allem szenengraph-basierte Renderingverfahren eingesetzt, wobei es eine Reihe von Systemen gibt, die unterschiedliche Vor- und Nachteile aufweisen, mit denen es sich näher zu befassen gilt.

1.2. Zielsetzung

Ziel dieser Arbeit ist es, aus der Auswahl der szenengraph-basierten Renderingsysteme zwei zu bestimmen, diese im Vergleich zu diskutieren und mit Hilfe von selbstgewählten Beispielen zu demonstrieren.
Dabei steht keine systematische Einführung in die APIs (**A**pplication **P**rogramming **I**nterface), sondern die allgemeine Charakterisierung der zwei gewählten 3D-Grafiksprachen im Vordergrund.

1.3. Schwerpunkte

Es werden als Schwerpunkte gesetzt:
- die Diskussion der Vor- und Nachteile, bzw. Gemeinsamkeiten der einzelnen Systeme im Vergleich anhand von auszuwählenden Kriterien,
- die Aneignung von spezifischen Programmierkompetenzen durch geeignete Literatur, Tutorials usw.
- die Implementierung von Beispielprogrammen zur Demonstration der Unterschiede der beiden Systeme.

1.4. Auswahl der Renderingsysteme

Die Auswahl von zwei zu untersuchenden Renderingsystemen fiel bei dieser Arbeit auf Anwendungen, die auch den Studenten des Studienganges Computervisualistik aufgrund Ihrer Vorkenntnisse leicht näher gebracht werden können. Zum Einen also auf Java 3D, da die Programmiersprache Java bereits im Grundstudium gelehrt und angewandt worden ist und zum Anderen Open Inventor als „Erweiterung“ von OpenGL, mit dem im Hauptstudium in der Comutergrafik gearbeitet worden ist. Großer Wert ist während der Studienarbeit auf die Darstellung von Aufbau, Struktur und Funktionalität gelegt worden, weniger tatsächlich als auf einen *bewertenden* Vergleich.

2. Grundlagen zu szenengraph-basierten Renderingsystemen

2.1. Existierende 3D-Graphiksysteme

Es existierende verschiedene Renderingsysteme auf dem Markt, deren APIs für unterschiedliche Bereiche ausgelegt sind. Ganz klar muss hierbei unterschieden werden, für welchen Bereich diese APIs entwickelt wurden. Eine kleine, nicht den Anspruch an Vollständigkeit erhebende Auswahl existierender 3D-Systeme, mit deren Ausrichtungen, wird in der folgenden Übersicht gegeben:

DirectX (Microsoft)

Dieses API wurde ausschließlich für moderne 3D-Spiele entwickelt und besitzt daher auch nur eingeschränkte 2D-Fähigkeiten. In diesem Bereich ist es jedoch sehr beliebt, da für sehr viele 3D-Beschleunigerkarten Treiber existieren. Seit Version 6 kann man mit DirectX sehr einfach über ein rudimentäres Klassengerüst programmieren, was die Entwicklungszeiten enorm verkürzt.

OpenGL (Verschiedene - SGI)

OpenGL ist eine universelle Grafik-Library, welche in Version 1.1 auch für Entertainment-Software sehr beliebt ist. Die Ursprünge liegen allerdings im professionellen Bereich, in welchem Version 1.2 verwendet wird. Da die Spezifikationen von OpenGL durch SGI offengelegt wurden, existiert es für nahezu jedes Betriebssystem.

IRIS Performer (SGI)

ist eine für Echtzeitsimulationen ausgelegtes Highlevel-Szenengraphen-API. Die Architektur des Performer ist daher von vornherein für den Betrieb mit mehreren Prozessoren, Grafikkarten und Bildschirmen konzipiert.

Java 2D/3D (SUN - JavaSoft)

Diese Schnittstellen für die Sprache Java bauen auf den APIs DirectX oder OpenGL auf und erweitern diese um eine übersichtliche Klassenstruktur und - bei Java 3D - um weitere Fähigkeiten wie Raypicking, Sound u.a. Java 2D/3D wird im Moment hauptsächlich im semi-professionellen Bereich verwendet.

VTK (Kitware)

Das "Visual Toolkit" ist vorwiegend im professionellen Bereich anzutreffen. Dabei baut es in den meisten Fällen auf der Funktionalität von OpenGL auf, ist jedoch nicht darauf beschränkt. VTK existiert für die populären Betriebssysteme und ist trotz des aufgesetzten Klassengerüsts sehr schnell.

Open Inventor (SGI)

Diese API ist ein Klassengerüst für OpenGL. Die Funktionalität von OpenGL und der GLUT (OpenGL Utility Toolkit) wird in Klassen gekapselt. Auch diese API ist im professionellen Bereich (z.B. Medizin) sehr beliebt, die Weiterentwicklung wurde allerdings von SGI vor einiger Zeit eingestellt.

In der Fachliteratur findet sich die Aussage, daß es eigentlich nur zwei "echte" APIs gibt (OpenGL und DirectX), was sich allerdings einzig und alleine mit der Tatsache begründet, dass hierfür die beste Hardwarebeschleunigung existiert.

Abbildung 1: Logos von Graphik-API's

2.2. Genereller Aufbau eines 3D-Graphik-Toolkit

Es gibt drei elementare Ebenen, auf denen ein 3D-Toolkit die Entwicklung von Graphik-applikationen ermöglichen kann:

1. **Die Repräsentation der Objekte**

Dinge sollten als veränderbare Graphikobjekte verwaltet werden, und nicht nur als eine Menge von Zeichenprimitiven. Das bedeutet, dass Applikationen, die dieses Toolkit verwenden, nur wissen müssen, "was" ihre Objekte sind, und nicht auch "wie" sie dargestellt werden müssen.

2. **Interaktivität**

Zusätzlich zur graphischen Darstellung der Objekte muß es eine Möglichkeit zur Verwaltung von Ereignissen (events) geben, damit auf die Darstellung der Szene und der Eigenschaften der Objekte (Position, Lage, usw.) in geeigneter Weise Einfluss genommen werden kann.

3. **Die Architektur**

Das Toolkit sollte nicht verändert werden müssen, wenn man spezielle, komplexe Probleme damit lösen will. Das Toolkit sollte in seinem Funktionsumfang, den Möglichkeiten der Einbindung und seiner Flexibilität ausreichen, um jedes graphische Darstellungsproblem zu lösen.

Die ausgewählten Renderingsysteme entsprechen diesen Anforderungen, was in dieser Studienarbeit auch gezeigt wird.

2.3. Begriff Szenegraph

Zunächst kurz zur Begriffserläuterung: In der Software-Architektur eines computer-graphischen Systems wird zwischen zwei Typen der Szenenpräsentation unterschieden:

1. hierarchische Szenenpräsentation (Szenegraph)

2. sequentielle Szenenpräsentation (Liste von Szene-Objekten).

Beide Systeme die in dieser Studienarbeit behandelt werden, gehören zum 1.Typ, der daher in diesem Abschnitt näher erläutert wird.

Der Szenengraph, ein gerichteter azyklischer Graph (DAG), ist die häufigste Form der Szenenrepräsentation in computergraphischen Systemen. Ein Szenegraph hat nur einen Wurzelknoten; dieser enthält alle Objekte, die die Szene betreffen. Ein Szenegraph besteht aus einer unbestimmten Anzahl von Knoten (engl. node), die je nach Typ verschiedene Informationen enthalten, wie z.B. Form eines Körpers (Kegel, Kugel, etc.), Material und Transformation.

Blattknoten repräsentieren Szenengeometrien, während innere Knoten graphische Attribute, geometrische Transformationen und Subgraphen verwalten. Der Szenengraph unterstützt die hierarchische geometrische Modellierung und den Ausdruck gemeinsamer graphischer Attribute.

Während einer Traversierung wird insbesondere die Kohärenz in der Attributierung und Transformation der Szenenobjekte ausgenutzt, die zur Optimierung des Renderings wesentlich beitragen. Allerdings muss der Szenengraph als Modellierungsstruktur unterschieden werden von der Struktur der internen Objektdarstellung.

Erfolgt die hierarchische Modellierung auf der Basis einer Baumstruktur, kann die gemeinsame Nutzung von Submodellen nicht direkt ausgedrückt werden; die multiple Nutzung von Szenen-Teilgraphen ist jedoch eine wesentliche Eigenschaft, da sie eine kompakte Spezifikation des Gleichartigen in komplexen Szenen ermöglicht.

Der Szenegraph von Open Inventor und von Java 3D wird in den Einzel- und jeweiligen Besonderheiten sehr ausführlich in den nachfolgenden 2 Kapiteln erläutert.

3. Open Inventor

3.1. Einführung

Open Inventor ist ein von Silicon Graphics entwickeltes objektorientiertes Toolkit zur Programmierung von dreidimensionalen Graphiken, das in C++ geschrieben ist. Es basiert auf OpenGL und ist auch für andere Plattformen, z. B. Hewlett Packard, SUN und Microsoft-Windows verfügbar. Eine ebenfalls existierende Java-Implementation ist weitgehend plattform-unabhängig.

Die Open Inventor-Klassenbibliothek ist eine Sammlung von Objekten und Methoden für high-level 3D-Graphik-Programmierung.

3.1.1. Möglichkeiten von Open Inventor

Open Inventor bietet eine Vielzahl an Funktionen zur Berechnung, Transformation und Darstellung von 3D-Graphik-Objekten. Durch die objektorientierte Kapselung ist eine maximale Ausnutzung der Fähigkeiten von 3D-Graphikhardware mit minimalem Programmieraufwand möglich.
Open Inventor bildet praktisch fast alle Grafikprimitive von OpenGL irgendwie nach. Die Grafikprimitive ähneln sich sehr stark, so daß die Performance und sonstige Eigenschaften von OpenGL optimal ausgenutzt werden können.
Neben der reinen Darstellung bietet Open Inventor Möglichkeiten, Szenen zu animieren und eine Interaktion zwischen dem Benutzer und den Objekten einer 3D-Szene zu unterstützen.
Open Inventor ermöglicht das Rendering von Szenen, implementiert ein flexibles Speicherformat und stellt zahlreiche Möglichkeiten für eine Echtzeitinteraktion mit der dargestellten Szene zur Verfügung. So können etwa die angezeigten Szenen intuitiv mit der Maus bewegt und gedreht werden, mit einfachen Konstrukten Animationen programmiert werden oder Positionsparameter und Materialeigenschaften über spezielle Fenster direkt in der Szene verändert werden.

3.1.2. Komponenten

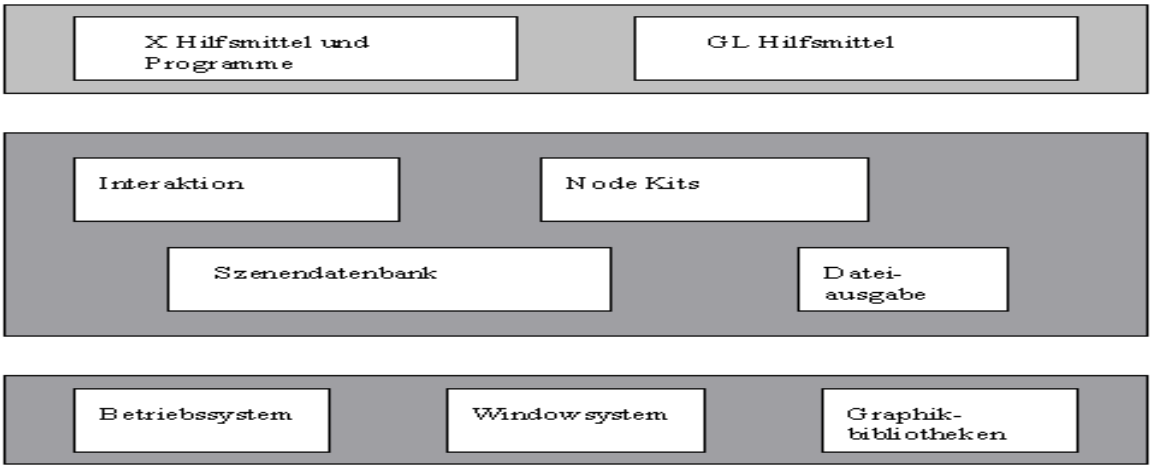
Open Inventor kann in eine portable Kernkomponente, die auf OpenGL fußt, und in eine betriebssystemspezifische Komponente (z.B. für Win32, X11, Motif) unterteilt werden. Ursprünglich war sie für UNIX-Betriebssysteme oder deren Derivate, wie IRIX, konzipiert worden. Somit war X11 bzw. Motif das zugehörige Window-System.

3.1.3. Aufbau des Toolkit

Das Open Inventor Toolkit besteht aus drei Hauptbereichen, wie die Abbildung 1 zeigt.

Auf Applikationslevel beinhaltet Open Inventor einige Hilfsprogramme wie zum Beispiel Editoren für Farbe, Material, Lichtquellen, sowie verschiedene Viewer, die spezielle Interaktionen des Benutzers ermöglichen.

Abbildung 1: Aufbau von Open Inventor



3.1.4. Dateiformat

Zum Austausch mit anderen Programmen und Prozessen existiert ein offenes Dateiformat, das es ermöglicht, Szenengraphen zu laden und zu speichern. Mit Dateien ist es möglich, Szenengraphen zwischen Prozessen und Programmen auszutauschen. Die Konvertierung von anderen Grafik-Dateiformaten ist möglich. In Open Inventor gibt es ein binäres Dateiformat und ein lesbares ASCII-Dateiformat.
Aus heutiger Sicht erscheint das Format nicht mehr zeitgemäß, XML wäre da sicher eine mögliche Alternative oder Erweiterung, vor allem weil es dafür validierende Parser, spezialisierte Editoren und andere nützliche Tools gibt. Denkbar wäre zum Beispiel auch, einfachere Operationen am Szenengraphen als XSL-Transformationen zu definieren, denn XSL ist eine auf XML basierende Sprache, mit der man Transformationen von XML-Dokumenten definieren kann.

3.2. Grundlegendes Konzept: Der Szenegraph

3.2.1. Der Szenegraph

Open Inventor speichert die 3D-Szene mit all ihren Objekten in einer speziellen Struktur, der Baumstruktur, genannt Szenegraph (scene graph). Mit Objekten sind hier die 3D-Geometrien als Oberflächenmodelle mit zusätzlichen Beleuchtungsinformationen gemeint.

Der Szenegraph dient dazu, eine komplexe Szene auf dem Bildschirm modellieren und anzeigen zu können. Er ist ein gerichteter azyklischer Graph und besteht zum einen aus Knoten (nodes), die die einzelnen Elemente der Szene und deren Eigenschaften modellieren, und zum anderen aus Gruppen (groups), die die Knoten verknüpfen.

Der Szenegraph bildet zusammen mit einigen Zusatzinformationen die Szenendatenbank (Scene Database), siehe auch Abbildung 1.

Es existiert in einem Szenegraphen immer ein Objekt "Renderarea", also ein Fenster, in dem die Szene dargestellt wird. Dieses Objekt kümmert sich um ein automatisches Redraw, um eine Event-Transformation aus dem Window-System in das Toolkit-System und um andere nützliche Dinge.

3.2.2. Die Szenendatenbank

Die Szenendatenbank kann einen oder mehrere Szenengraphen verwalten. Die Bezeichnung „Datenbank“ ist eigentlich nicht das richtige Wort, da die minimale Anforderung für Datenbanken, nämlich Daten persistent zu verwalten, bei der Szenendatenbank nicht zutrifft. Auch andere für Datenbanken übliche Konzepte gibt es in Open Inventor nicht, wie z.B. Transaktionen. Vielmehr ist die Szenendatenbank eine komplexe objektorientierte Datenstruktur im Hauptspeicher, in dem Objekte als Knoten (nodes) gespeichert werden. Von diesen Knoten gibt es verschiedene Klassen für geometrische Objekte, für die Eigenschaften dieser Objekte und für Transformationen.

Auf den damit definierten Graphen lassen sich speziell implementierte Methoden, sogenannte Aktionen (actions) anwenden, die im Abschnitt 3.2.2.14. näher erläutert werden.

3.2.2.1. Die Basis-Datenstruktur der Szenendatenbank

Die Basis-Datenstruktur der Szenendatenbank ist der Knoten, von dem eine Vielzahl von Klassen abgeleitet sind. Ein Knoten wird als Teil des Szenengraphen von einem oder mehreren anderen Knoten referenziert und er kann selbst einen oder mehrere Knoten referenzieren. Je nach Typ des Knotens enthält ein Knoten Informationen z.B. über die Form eines 3D-Körpers (Kugel, Quader, Kegel, etc.), das Oberflächenmaterial, Transformation, Licht oder Kamera.
Auch interaktive Elemente, die Manipulatoren und sogar einfache Animationen werden mit Hilfe solcher Knoten abgebildet. Jeder Knoten hat also seine Aufgabe, im nächsten Abschnitt werden die wichtigsten vorgestellt.

Des Weiteren existieren Objekte zur vereinfachten Erzeugung von Animationen. Der Bereich der Interaktion beinhaltet Klassen für Events und sogenannte „smart nodes". Eventklassen dienen zum Beispiel zur Verwaltung von Maus-Button-Events oder reagieren auf bestimmte Tasten-betätigungen. Smart nodes enthalten spezielle Intelligenz, die sie zur Ausführung bestimmter Funktionen benötigen. So kann zum Beispiel ein Knoten zur Selektion von Objekten über Maus-Events das Picking überwachen und die selektierten Objekte farblich hervorgehoben darstellen.

So genannte „Node Kits“ stellen sinnvoll zusammengefasste Mengen von Knoten dar, die den Aufbau von komplexen und doch konsistenten Szenen ermöglichen. Sinnvolle Arten von Knoten werden zu Subgraphen kombiniert. Fügt man also ein Node Kit ein, so werden zum Beispiel automatisch ein Knoten für ein geometrisches Objekt, ein Materialknoten und eine Transformation gleichzeitig eingefügt, siehe Anschnitt 3.2.2.7.

In den nachfolgenden Abschnitten erfolgen nähere Erläuterungen zu den Bestandteilen der Datenstrukturen, soweit diese zum Grundverständnis von Open Inventor notwendig sind.

3.2.2.2. Knoten

Jeder Knoten in der Szenendatenbank hat, wie bereits erwähnt, eine spezielle Funktion.

- Shape Knoten repräsentieren geometrische oder physikalische Objekte, siehe Abschnitt 3.2.2.2.3.
- Property Knoten beschreiben vielfältige Attribute dieser Objekte, siehe Abschnitt 3.2.2.2.3.
- Group Knoten verbinden andere Knoten zu Graphen oder Subgraphen.

Es existieren aber auch andere Arten von Knoten wie zum Beispiel Kameras oder Lichtquellen

Einige Beispiele aus den einzelnen Klassen:

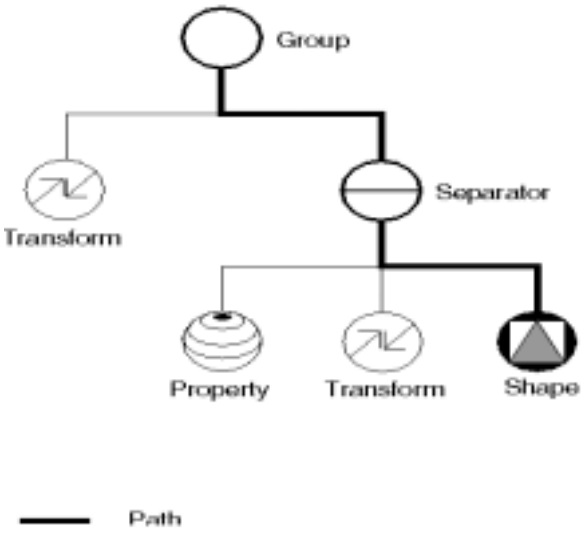
Shape nodes	Cone, Cube, Faceset, TriangleStripSet
Light/camera nodes	Perspective Camera, PointLight
Group nodes	Group, Seperator
Property nodes	BaseColor, Material, Transform, Font

Jede Instanz einer Objektklasse beinhaltet natürlich bestimmte Nutzdaten. Diese Daten werden innerhalb des Objektes in Subobjekten, sogenannten Fields, gespeichert.

Fields haben einen bestimmten Werttyp, der die Art der Daten beschreibt. So kann zum Beispiel ein Objekt aus der Klasse Zylinder zwei Fields vom Typ "float" haben: Radius und Höhe. Zu diesen Fields existieren konsistente Methoden zum Editieren, Abfragen, Lesen, Schreiben und überwachen dieser Instanzdaten. Bestimmte Arten von Daten können sich von mehreren Objekten geteilt werden, um Speicherplatz zu sparen. Unter diese Kategorie von Daten fallen zum Beispiel Koordinatenangaben oder Normalvektoren.

Open Inventor stellt eine ganze Anzahl von Knotentypen zur Verfügung, um verschiedene Primitive und ihre Eigenschaften zu modellieren. Im Folgenden sollen die wichtigsten kurz aufgeführt werden.

Abbildung 2: Ein einfacher Szenengraph

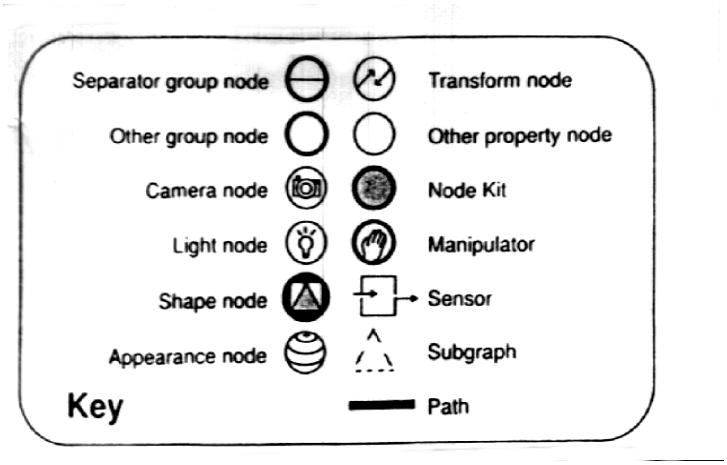


3.2.2.2.1. Group Nodes

Mehrere Knoten können zu einer Gruppe zusammengefasst werden. Eine Gruppe ist ein spezieller Knoten, der Verweise auf einen oder mehrere andere Knoten hat.

Abbildung 2 zeigt einen einfachen Szenengraphen mit sechs Knoten, zwei davon sind Gruppen. An die Knoten sind die Typen (Klassen) der Knoten angeschrieben. Die verwendeten Knoten-Symbole sind die in der Inventor-Referenz benutzten.

Abbildung 3: Knoten-Symbole



Die Gruppenknoten verbinden somit Knoten in Graphen. Jede GroupNode-Klasse bestimmt, ob und in welcher Weise ihre Knoten abgearbeitet werden und wie der Status der Abarbeitung an Nachfolgeknoten weitergegeben wird. Solche Informationen, die aktuelle Materialdaten, Ausgabeeigenschaften, usw. enthalten, werden für gewöhnlich von oben nach unten im Baum weitergereicht. Es werden aber nicht nur Nachfolgerknoten weiter unten im Baum versorgt, sondern auch Knoten, die sich auf gleicher Höhe im Baum befinden. Einige wenige Group Nodes sind auch in der Lage, Informationen an Vorgängerknoten zu vererben.

Group Nodes sind also die elementaren Bausteine, um komplexe Szenengraphen aufzubauen. Open Inventor unterscheidet zwischen Separator-Knoten und Group-Knoten. Erstere speichern alle Eigenschaften, bevor die darunter liegenden Knoten besucht werden, und stellen sie wieder her, wenn die Traversierung abgeschlossen wurde. Dies ist ein bequemer Mechanismus, Eigenschaften lokal für ein Objekt zu definieren, siehe nächster Abschnitt.

Group-Knoten dienen nur zum logischen Gruppieren des Graphen, speichern aber keine Eigenschaften.

3.2.2.2.2. Seperator Nodes

Seperator Nodes speichern den Zustand der Darstellungsweise, wenn sie in der Abarbeitung des Szenegraphen an die Reihe kommen, und stellen diesen Zustand wieder her, wenn alle ihre Nachfolgerknoten abgearbeitet wurden. Auf diese Weise wird der Effekt der nachfolgenden Knoten auf die Szene isoliert. Zum Beispiel können Seperator Nodes die Ergebnisse von Bounding-Box-Berechnungen cachen. Die dadurch erzielte Beschleunigung ist groß genug, um ein dynamisches Highlighting unter einem sich bewegenden Mauscursor zu ermöglichen.

Um einen Szenengraphen flexibel verändern zu können, gibt es Schalter, genannt Switch-Knoten. Sie werden wie ein Separator-Knoten verwendet, besitzten jedoch ein Feld, mit dem entschieden werden kann, welcher der angeschlossenen Knoten traversiert werden soll.

Weitere Optionen ermöglichen es, alle Knoten oder keinen Knoten zu traversieren. Ein Switch Node ist somit eine spezielle Art von Seperator Node. Er wählt nur einen seiner Nachfolgerknoten zur Abarbeitung aus. Diese Methodik kommt zum Beispiel sinnvoll zum Einsatz, wenn verschiedene Detailstufen in der Darstellung implementiert werden sollen.

Eine andere Art von Seperator Node ist ein Array Knoten. Diese Knotenklasse führt ihre Nachfolgerknoten mehrmals aus, wobei sie jedes Mal eine bestimmte Transformation auf den Subgraphen ausübt.

3.2.2.2.3. Weitere Knoten

Um ein Objekt an einer bestimmten Stelle in der Szene zu platzieren, wird immer ein Transformationsknoten benötigt. Es ist in Open Inventor möglich, einen Teilgraphen einer Szene mehrfach aufzunehmen. Auf diese Weise müssen komplexere Strukturen, die in einer Szene mehrfach vorkommen, nur einmal gespeichert werden. Neben den Knoten ist die Reihenfolge der Traversierung angegeben, einige Knoten werden mehrfach besucht.

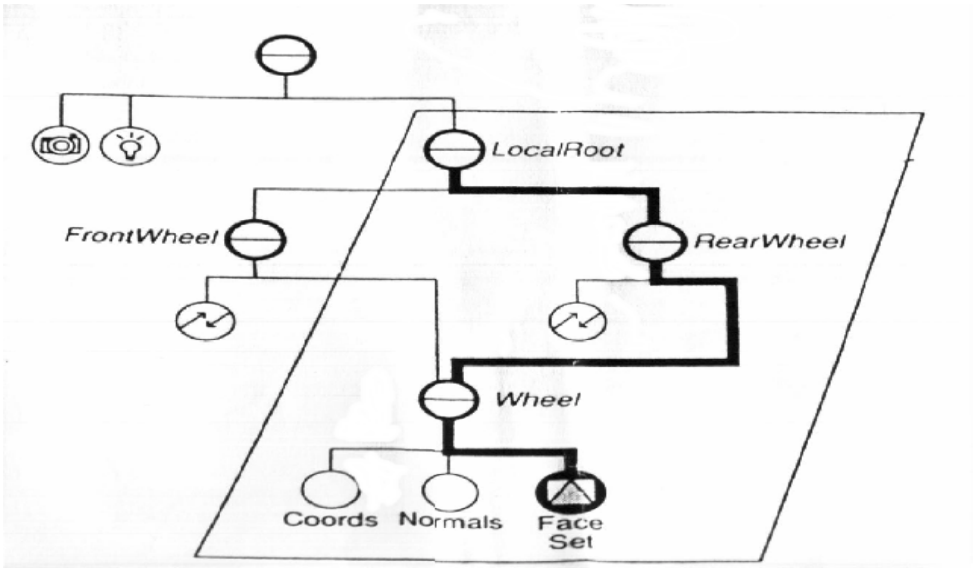
Der Szenengraph wird beim Rendern wie folgt interpretiert: der Szenengraph wird von oben nach unten und von links nach rechts traversiert. Jeder Knoten wird typabhängig gerendert. Das Transformations-Objekt „Transform“ repräsentiert zum Beispiel eine geometrische Transformation und führt beim Rendern zu einer Veränderung der Transformationsmatrix des sogenannten Render-Status. Der Render-Status wird beim Rendern der in der Traversierungs-reihenfolge folgenden Formen angewendet, so dass immer die zuletzt gesetzte Eigenschaft das Rendern der Form beeinflusst.

Das Property-Objekt stellt eine hier nicht näher definierte Eigenschaft wie z.B. eine Material- oder Beleuchtungseigenschaft dar. Beim Rendern werden bei einem Vorkommen eines Property-Objekts die entsprechenden Eigenschaften des Render-Status überschrieben. Das Shape-Objekt repräsentiert eine geometrische Form und wird beim Traversieren des Graphen unter Berücksichtigung des aktuellen Render-Status gerendert.

3.2.2.3. Pfade

Knoten können im Allgemeinen mehrere Vorgänger haben. Durch diese Erweiterung lässt sich Wiederverwendbarkeit erreichen. Soll zum Beispiel ein Auto dargestellt werden, so muss ein Rad nur einmal konstruiert werden, kann dann aber viermal mit einer geeigneten Transformation als Subgraph eingesetzt werden. Dieses Konzept führt zu einer überschaubareren und kompakten Szenenrepräsentation. Daraus ergibt sich aber auch ein gravierender Nachteil: es ist nicht mehr möglich, mit der Angabe eines Knotens ein bestimmtes Objekt zu spezifizieren. Um hier Abhilfe zu schaffen, wurde das Konzept der Pfade eingeführt. Pfade zeigen immer von einem Ausgangsknoten nach unten durch den Baum weisend auf einen bestimmten Knoten. So kann zum Beispiel eine Pick-Aktion einen Pfad vom Rootknoten bis zum ausgewählten Objekt liefern.

Abbildung 4: Beispiel (Szenengraph - Fahrzeug) für einen Pfad



Der "Wheel" Knoten in Abbildung 4 wird mehrfach instanziiert, zumal er ein Nachfolgerknoten von "FrontWheel" und "RearWheel" ist. Der Pfad (die dicke Linie) zeigt eindeutig vom Root-knoten auf das Objekt, welches das Hinterrad definiert. Innerhalb des Parallelogramms liegt der Subgraph, der von dem Pfad definiert wird.

Pfade werden also verwendet, um ein bestimmtes Objekt einer 3D-Grafik zu isolieren. Da der Weg von der Wurzel zu einem bestimmten Knoten des Szenengraphen nicht eindeutig ist, wie das in einem Baum der Fall wäre, muss dieser Pfad durch eine Aufzählung der Knoten entlang des Pfades spezifiziert werden.
Pfade werden durch Auswahl- oder Such-Aktionen berechnet und zurückgegeben. Klickt der Benutzer ein Objekt am Bildschirm mit der Maus an, so wird das Objekt ausgewählt. Der Selection-Knoten verwaltet eine Liste von Pfaden der gegenwärtig ausgewählten Objekte. Auf Pfade können bestimmte Aktionen angewendet werden, siehe Abschnitt 3.2.2.14.

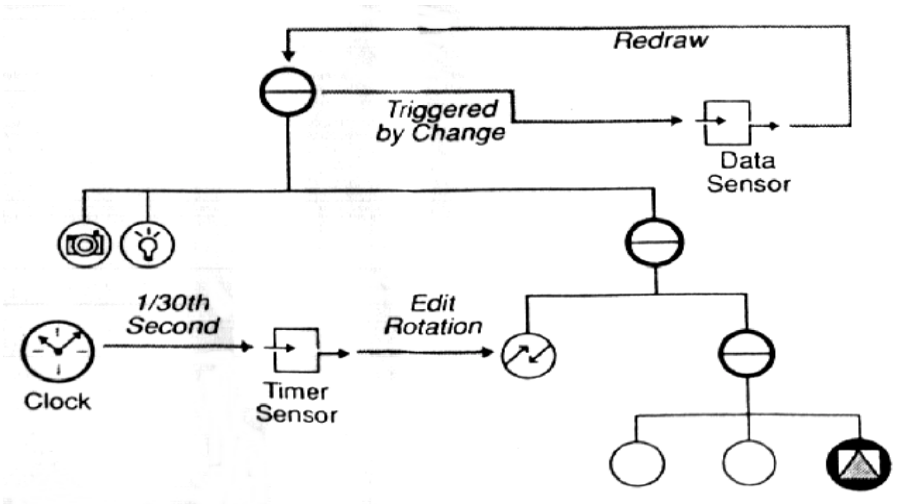
3.2.2.4. Sensors

Sensoren überwachen Veränderungen von Knoten oder werden zur Implementierung von Animationen verwendet. Es gibt zwei Arten von Sensoren: Data und Time Sensors.

Data Sensors werden ausgelöst, wenn sich die Daten des angeschlossenen Knoten (oder die Daten seiner Kinder) ändern. Über solch einen Data Sensor kann zum Beispiel ein interaktiver Sensor Sliders oder andere Anzeigen ändern, wenn die zugehörigen Objekte von anderen Programmobjekten verändert wurden. Wenn ein Data Sensor am Rootknoten angeschlossen wird, so erkennt er Veränderungen in der ganzen Szene. Dieses Verhalten kann zum Beispiel für ein bedingtes Redraw genutzt werden, bei dem die Szene nur neu gezeichnet werden muss, wenn sich etwas verändert hat.

Time Sensors werden zu einem bestimmten Zeitpunkt oder in einem regelmäßigen Intervall ausgeführt. Über diese Funktionalität lassen sich leicht Animationen erzeugen.

Abbildung 5: Ein Szenengraph, der Sensors verwendet.



Zur Darstellung in Abbildung 5:

Der Data Sensor beim Rootknoten wird ausgelöst, wann immer sich etwas an einem der nachfolgenden Knoten ändert. Er wird verwendet, um ein erforderliches Neuzeichnen der Szene auszulösen. Der Time Sensor, welcher mit einer Uhr verbunden ist, triggert in regelmässigen Abständen und animiert derart die Rotation des Objektes rechts unten.

3.2.2.5. Events

Das Open Inventor Toolkit verwendet einen simplen Algorithmus, um User Events aus dem Windowing System auf die Knoten zu verteilen. Ein Event im Windowing System resultiert aus einer User Aktion, also zum Beispiel einer Mausbewegung, einem Tastendruck, usw. Sollte solch ein Event in einem Window der Renderarea aufgetreten sein, so wird der Event an dieses Objekt übergeben. Das Renderarea-Objekt kann den Event entweder selbst verarbeiten, beispielsweise könnte ein Viewer die Kamera mit der Mausbewegung verschieben, oder den Event in einen Toolkit-Event übersetzen und ihn an die Objekte in der Szenendatenbank übergeben, indem ein Handle-Event-Objekt mit dem Rootknoten verbunden wird. Der Event wird dann durch den Baum weitergereicht und verteilt. Jeder interessierte Knoten kann auf den Event reagieren und den Event-Status gegebenenfalls auf "handled" (=bearbeitet, keine weitere Bearbeitung nötig) setzen. Nur Knoten, deren Status noch nicht "handled" ist, werden durch den Baum weitergereicht.

3.2.2.6. Manipulatoren

Open Inventor wurde speziell im Hinblick auf die Unterstützung von Interaktivität entworfen. Ein wichtiges Element dabei sind Manipulatoren, spezielle interaktive Knoten, die auf Benutzerschnittstellen-Ereignisse reagieren und direkt vom Benutzer editiert werden können. Manipulatoren können einfach über das Eventmodell integriert werden.

Die Manipulatoren sind eine Ableitung der Transformationsknoten. Sie haben eine Reihe von Kindern in Form von Handle-Objekten. Manipulatoren sind, wenn sie im Szenegraphen integriert werden, in der Lage Events aufzunehmen und zu verarbeiten. Visuell repräsentieren sie sich in Form kleiner Kugeln, Linien oder Kegel. Klickt man eines dieser Manipulatorobjekte an und hält es mit der Maus fest, so sind durch ziehen direkte Veränderungen an dem zugehörigen Objekt möglich. Der Anwender kann die Szene intuitiv mit Manipulatoren verändern.

Manipulatoren werden typischerweise in der Szene gerendert, z.B. als greifbare Box. Der Benutzer kann durch entsprechende Interaktionen mit dem Manipulator ein Objekt drehen, verschieben und skalieren, wie beispielsweise mit dem Trackballobjekt. Dieses legt eine unsichtbare Bounding Sphere um das angeschlossene Objekt, welches es manipuliert. Über diese Kugel werden Mausbewegungen in rotatorische Veränderungen für das Objekt übersetzt. Drei zylindrische Streifen um die Kugel erleichtern das Spezifizieren von Bewegungen um die Hauptachsen. Andere simple Manipulatoren führen zum Beispiel nur einfache Translationen in einer Dimension durch oder werden zu komplexeren Manipulatoren wie dem Trackball kombiniert. Die Änderungen an solchen Manipulatoren werden sofort umgesetzt in entsprechende Änderungen im Szenengraphen. In der Folge werden betroffene Teile des Szenengraphen invalidiert und neu gerendert, so daß die Wirkung der Interaktion sofort am Bildschirm nachvollzogen werden kann.

3.2.2.7. Node Kits

Die Szene-Graph-Bibliothek ist recht allgemein gehalten, um optimale Performance und Flexibilität zu gewährleisten; aber dadurch kann sie auf unerfahrene User verwirrend wirken. Es gibt keine Beschränkungen beim Erstellen von Graphen und so kann es oft vorkommen, daß bizarre und sinnlose Konstruktionen entstehen, wenn nicht bestimmte strukturelle Richtlinien eingehalten werden. Node Kits vereinfachen diese Problematik, indem sie eine konsistente Art der Graphkonstruktion erzwingen. Mit Node kits kann man eine Menge von Knoten zu einem Knoten zusammenfassen. Von Außen betrachtet ist ein Node kit selbst ein Knoten, der wie andere Knoten in den Szenengraph eingefügt werden kann. Ein Node kit kann als eine Black box gesehen werden, die einen ganzen Subgraphen hinter einem Knoten verbirgt. Der Zweck von

Node kits ist, einer Gruppe von Knoten zusätzliche Methoden zu geben, und Information zu kapseln und zu verstecken, ähnlich wie bei der objektorientierten Vererbung.

Jedes Node Kit enthält also einen strukturierten Subgraphen von Datenbankknoten. Ein Template, welches zum Node Kit gehört, gibt an, welche Knoten angefügt werden können, und an welchen Stellen das geschehen sollte. Als Beispiel sei hier das Sphere Kit angeführt, welches neben der Objektknotenklasse einer Kugel auch ein Template enthält, welches die Anbindung von Materialknoten, geometrischen Transformationen und anderen Eigenschaftsknoten an den richtigen Stellen gestattet.

Ein Node kit enthält normalerweise bei der Instanziierung schon einige Knoten, und es können bei Bedarf weitere Knoten hinzugefügt werden. Die Art der Knoten, die hinzugefügt werden können, wird durch das Node kit definiert. Die Anordnung der Knoten wird vom Node kit organisiert. Ein Node kit kann auch weitere Node kits enthalten, so daß hierarchische Strukturen mit Node kits realisierbar sind.

Node kits bieten Methoden zur Vereinfachung der Erzeugung von Knoten des Node kits. Damit wird der Code kürzer und besser lesbar. Eigene Node-kit-Klassen können durch Vererbung erstellt werden. Obwohl auf node kits verzichtet werden könnte, sind sie in der Praxis ein wichtiges Hilfsmittel zur Strukturierung von Szenengraphen, und es wird stark empfohlen, diese zu verwenden.

3.2.2.8. Kameras und Lichtquellen

Open Inventor stellt zwei Kameratypen zur Verfügung: PerspectiveCamera für übliche Zentralperspektive und OrthographicCamera für eine orthographische Projektion.

Die zwei wesentlichen Lichtquellen in Open Inventor sind zum einen DirectionalLight, es definiert den Einfall von parallelen Lichtstrahlen auf die gesamte Szene, ähnlich etwa direktem Sonnenlicht und zum anderen PointLight, es definiert eine punktförmige Lichtquelle an einer bestimmten Stelle, die gleichmäßig in alle Richtungen strahlt. Eine Variante davon ist SpotLight, das vorwiegend in eine Richtung strahlt.

Es gibt keine räumlich ausgedehnten Lichtquellen in Open Inventor. Und wichtig in der Anwendung ist: Lichtquellen werfen keine Schatten.

3.2.2.9. Einfache Grundkörper

Open Inventor stellt eine Vielzahl vorgefertigter Knoten für Würfel, Kugeln, Zylinder und Kegel zur Verfügung, die jedoch nicht weiter modifiziert werden können und so für ernsthafte Anwendungen ungeeignet sind. Es können zudem fertige 3d-Buchstaben aus PostScript-Zeichensätzen erzeugt werden. Alle Klassen, deren Objekte direkt in den Szenegraphen eingehängt werden können, beginnen mit dem Kürzel So (Scene-Object), z.B. SoCone oder SoSphere. Die zusätzlichen Informationen eines Knotens, wie die Lage, Größe oder das Material, werden in den dem Knoten zugehörigen Feldern (engl. field) gespeichert. So enthält ein Kegel unter anderem die Felder bottomRadius und height.

3.2.2.10. Komplexe Körper

Die Modellierung von komplexeren Körpern kann mit verschiedenen Methoden geschehen:

FaceSet	ermöglicht die Definition von mehreren Flächen über definierende Punkte;
QuadMesh	stellt ein verformbares quadratisches Gitter zur Verfügung;

TriangleStripSet	stellt verformbare Streifen aus Dreiecken zur Verfügung.
NurbsSurface	ermöglicht die Definition von gerundeten Körpern mittels NURBS-Kurven.

Die Angabe von Eckpunkten und Oberflächennormalen geschieht jeweils über spezielle Knoten.

3.2.2.11. Eigenschaften

Material	ermöglicht die Angabe von Materialparametern;
Transform	definiert eine Transformationsmatrix durch Spezifikation von Position, Größe und Orientierung.
DrawStyle	ermöglicht das Umschalten auf Wireframe- oder Eckpunktdarstellung.
Complexity	ermöglicht es, zu bestimmen, wie genau das Rendering zu erfolgen hat, um für subjektiv unwichtigere Objekte weniger Rechenzeit aufzuwenden.

3.2.2.12. Texturen

Durch Definition eines Texture2-Knotens lassen sich Objekte mit zweidimensionalen Texturen versehen. Die Texturen können nur Farbe und Transparenz der Objekte modifizieren. Jedes Objekt kann nur eine Textur besitzen. Es ist möglich, reflection textures zu verwenden.

3.2.2.13. Engines

Eine weitere interessante Eigenschaft von Open Inventor ist die Möglichkeit, einzelne Felder, d.h. einzelne Parameter eines Knotens, mit anderen zu koppeln. Dies geschieht durch die so bezeichneten „engines“. Dies sind spezielle Knotentypen, die nicht in die normale Hierarchie des Szenengraphen eingeordnet sind, sondern mittels einer zweiten Relation, die einen Datenfluß beschreibt, mit der Szene verbunden werden. Diese zweite Relation wird in den Grafiken durch gepunktete geschwungene Linien symbolisiert.

Engines haben normalerweise Eingänge und Ausgänge, die über einen definierbaren Berechnungsprozeß miteinander gekoppelt sind. Mit Engines können Eigenschaften bestimmter Objekte miteinander verkoppelt werden. So könnte etwa die Transparenz eines Objekts mit seiner Position verknüpft werden. Bewegt man dann das Objekt durch irgendeinen Mechanismus, so verändert sich gleichzeitig die Transparenz.

Als Quellen für die Eingänge einer Engine stehen sowohl die Felder anderer Knoten als auch globale Felder zur Verfügung. Globale Felder sind keinem bestimmten Knoten zugeordnet und können als Schnittstellenparameter für einen veränderbaren Szenengraphen interpretiert werden.

Ein spezielles globales Feld ist bereits definiert und stellt eine laufende Uhrzeit zur Verfügung. Verwendet man dieses als Eingang für eine Engine und verbindet den Ausgang mit einem geeigneten Feld eines Knotens, so lassen sich Animationen definieren.

Man beachte den Unterschied zwischen Knoten und Engines. Engines sind ein Beispiel für Objekte der Szenendatenbank, die keine Knoten sind. Dies bedeutet insbesondere, daß Engines nicht Teil des Szenengraphen sind. Für die Praxis bedeutet das, daß man beim Überprüfen, ob der Szenengraph azyklisch ist, auf die Betrachtung der Engines verzichten kann. Außerdem spielen Engines auch keine Rolle bei der Traversierung des Szenengraphen, was zum Beispiel beim Rendern geschieht.

3.2.2.14. Aktionen

Bisher wurde gezeigt, wie ein Szenengraph konstruiert ist und welche Bestandteile er hat. Wie aber wird in Szenengraph verarbeitet, wenn mehr als ein Knoten dabei betrachtet werden muss? Das trifft zum Beispiel für das Rendern des Szenengraphen zu, wo jeder einzelne Knoten des Szenengraphen traversiert und gerendert werden muss.
Dazu gibt es in Open Inventor so genannte Aktionen (actions). Eine Aktion ist ein Objekt, das einen komplexen Algorithmus auf der Basis eines Teilgraphen definiert. Eine Aktion wird auf einen Knoten oder einen Pfad angewendet.

Ein Pfad ist, wie bereits im Abschnitt 3.2.2.3. erläutert, ein Weg von der Wurzel eines Szenengraphen zu einem bestimmten Knoten. Bei Anwendung einer Aktion auf einen Knoten wird der Teilgraph, dessen Wurzel dieser Knoten ist, traversiert. Die Reihenfolge der Traversierung ist vom Typ der Aktion abhängig, meist wird aber von oben nach unten und von links nach rechts traversiert, d.h. die Traversierung ist tiefenorientiert. Dabei verwaltet die Datenbank einen Traversierungsstatus, der während der Traversierung durch die Knoten verändert werden kann, abhängig von deren speziellen Verhaltensweise für diese Aktion. Der Objekttyp des Traversierungsstatus wird vom Typ der Aktion bestimmt. Praktisch alle Operationen, die eine Traversierung des Szenengraphen erforderlich machen, sind in Open Inventor Aktionen.

Es gibt folgende Aktionen, die auf Knoten angewendet werden können:

- zum Rendering der Struktur (render action)
- zum Auswählen, Suchen, Ändern von bestimmten Objekten
- zur Berechnung von Bounding Boxes, also zum Bestimmen der Größe der Szene (get bounding box action)
- zur Verarbeitung von events
- zum Abspeichern (write action), Lesen von und Schreiben in eine Datei
- zur Dateiausgabe und zum Drucken
- für die Ausführung von Callbacks einzelner Knoten

Aktionen sind also Objekte, die den Szenegraphen in bestimmter Weise abarbeiten, als Beispiele wird hier das Rendering und das Picking behandelt.

Rendering: Die Darstellung der Szene durch eine „Render Action“ sollte in Echtzeit ablaufen. Die Rendering-Methoden der Knotenklassen müssen also auf Performance optimiert sein. Bestimmte Teile der Rendering-Informationen für Subgraphen (wie zum Beispiel „Display Lists“) können in einem Cache-Speicher aufgehoben werden, wo sie für den nächsten Render-Durchgang dann wieder zur Verfügung stehen. Des weiteren kann der Rendering-Ablauf an beliebiger Stelle unterbrochen werden, wenn für die geforderte Darstellung in Echtzeit nicht genug Rechenzeit zur Verfügung steht, und anfällige User-Events nicht rechtzeitig bearbeitet werden könnten.

Picking: Das Picking ist ein elementarer Bestandteil von interaktiven Applikationen. Ein Knoten namens "RayPickAction" liefert zum Beispiel einen Pfad vom Rootknoten zum ersten Objekt, welches unter einem Maus-Cursor liegt, wenn die linke Maustaste gedrückt wurde.

3.2.2.15. Zusammenfassung zur Struktur

Der gegebene Überblick stellt lediglich eine Einführung dar, in der folgende Konzepte aufgezeigt wurden: Open Inventor verwendet Knoten, die in einer baumartigen Strukur

organisiert werden, um Szenen zu repräsentieren. Diese Knoten können für geometrische Objekte, für Eigenschaften, für Transformationen oder andere Dinge stehen. Transformationen und Materialeigenschaften bleiben bei der Abarbeitung der einzelnen Knoten erhalten. Ist dieser Effekt unerwünscht, so müssen Seperator Nodes eingefügt werden. Spezielle Arten von Knoten sind:

Events -	reagieren auf Events aus dem Windowing-System
Actions -	arbeiten Szenenbaum ab, z.B. Rendering
Sensors -	reagieren auf Veränderungen
Manipulators -	erlauben einfaches Verändern von Objekten.
Node Kits -	erlauben das gleichzeitige Einfügen einer Menge von semantisch sinnvoll zusammengehörenden Knoten, zum Beispiel geometrisches Objekt, Material und Transformation.

Einmal konstruierte Objekte können mehrfach wiederverwendet werden - normalerweise über Transformationen an verschiedenen Stellen im Raum. Bestimmte Objekte können dann über Pfade eindeutig identifiziert werden.

Open Inventor ermöglicht es, eigene Knoten durch in der Programmiersprache C++ geschriebene Programme zu definieren. Auf diese Weise können komplexere Primitive oder spezielle Eigenschaften leicht in das System integriert werden. Das Renderingsystem selbst kann allerdings nicht ohne weiteres verändert werden. Es ist also nicht möglich, eigene Shader zu schreiben.

3.2.3. Speicherverwaltung

Obwohl Open Inventor in C++ geschrieben wurde, ist eine komfortable Speicherverwaltung integriert worden, die ähnlich wie bei Java ein explizites Freigeben von Objekten unnötig macht. Der Programmierer sollte deshalb auf keinen Fall selbst Objekte mit „delete“ aus dem Speicher löschen. Der für die Speicherverwaltung verwendete Mechanismus beruht auf einer Verwaltung von Referenzzählern für Objekte. Die Art, wie Open Inventor den Speicher verwaltet, ist für die Anwendungs-Programmierung essenziell. Man muß beim Programmieren die Speicher-verwaltungsmechanismen genau kennen und berücksichtigen. Wichtig ist zu beachten, dass ein Wurzelknoten eines Szenengraphen durch keinen anderen Knoten referenziert wird und deshalb explizit im Anwendungsprogramm referenziert werden muss, siehe Beispielprogramm im Abschnitt 3.3.2. Ein nicht referenzierter Knoten kann nicht gerendert werden. Folgende Anweisung inkrementiert den Referenzzähler für den Wurzelknoten: root->ref();

3.2.4. Komponenten-Bibliothek

Um das Rendern der Szenen bewerkstelligen zu können, ist eine Integration von Open Inventor in ein Fenstersystem (z.B. X) nötig. Diese Integration wird durch die Inventor Component Library realisiert. Die Bibliothek hat folgende Teile:

- Fensterobjekt für das Rendern von Szenen
- Main loop und Initialisierungsroutinen
- Ereignis-Verarbeitung
- Komponenten: Editoren (material editor, directional light editor)
  und Betrachtungskomponenten (fly viewer, examiner viewer)

Die Bibliothek kann erweitert werden, um z.B. weitere Fenstersysteme zu unterstützen oder um weitere Eigenschaftseditoren hinzuzufügen.

3.3. Aufbau eines Programmes

Die einzelnen Schritte bei der Programmierung einer Szene sind im Allgemeinen die folgenden:

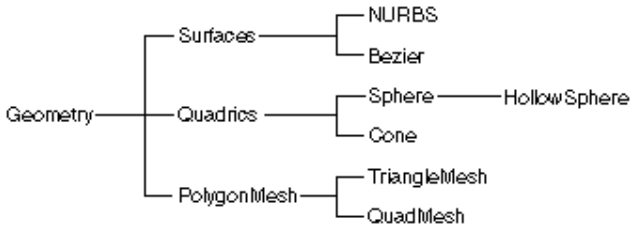
• Erzeugen eines Fensters und eines Renderbereichs, in dem die Szene gerendert werden soll

• Konstruktion des Szenengraphen

• Rendern des Szenengraphen innerhalb des Renderbereichs des Fensters

3.3.1. Open Inventor Klassenhierarchie

Open Inventor hat eine große Klassenbibliothek, die hierarchisch aufgebaut ist. Ein Beispiel für die Klassenbeziehungen zeigt das Beispiel in Abbildung 6, entnommen aus „Inventor Mentor“, das typisch für Kugel, Konus, Würfel u.a. geometrische Formen ist.



Zu diesem Beispiel: Funktionen und Variablen sind in der Klasse Geometry definiert, existieren also für alle Subklassen. Geometry hat eine Variable Bbox und eine Funktion getBbox(), so haben alle Subklassen von Geometry also auch Bbox und getBbox().

3.3.2. Programmbeispiel Konus

Ein kurzes aber komplettes Programmbeispiel in C++, das einen Konus rendert, soll die Programmierung mit Open Inventor zeigen. Es wurde hier bewusst ein sehr einfaches Beispiel gewählt, dass nicht alle Möglichkeiten von Open Inventor aufzeigt, dafür aber nachvollziehbar ist.

Die Header-Dateien einbinden.

```
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/Win/SoWinRenderArea.h>
#include <Inventor/Win/SoWin.h>
```

Funktion schreiben

```
int main(int argc, char **argv)  {
```

Initialisierung der SoWin-Bibliotheken. Der Rückgabewert ist ein Fenster.

```
Widget myWindow = SoWin::init(argv[0]);
if (myWindow == NULL) exit(1);
```

Einen Szenegraphen mit rotem Konus erzeugen

Die Wurzel:

```
SoSeparator *root = new SoSeparator;
```

Knoten für Kamera und Material:

```
SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
SoMaterial *myMaterial = new SoMaterial;
```

Für die Speicherverwaltung von Open Inventor nötig: Referenzierung des Wurzelknotens

```
root->ref();
```

Einhängen der Knoten in den Szenegraphen:

```
root->addChild(myCamera);
root->addChild(new SoDirectionalLight);
myMaterial->diffuseColor.setValue(1.0,0.0,0.0);
root->addChild(myMaterial);
root->addChild(new SoCone);
SoWinRenderArea *myRenderArea = new SoWinRenderArea(myWindow);
```

Kameraposition setzen

```
myCamera->viewAll(root,
myRenderArea->getViewportRegion());
```

Szenengraph mit dem Renderbereich verknüpfen

```
myRenderArea->setSceneGraph(root);
```

Fenstertitel setzen

```
myRenderArea->setTitle("Hello Cone");
myRenderArea->show();
```

Anzeigen des Fensters

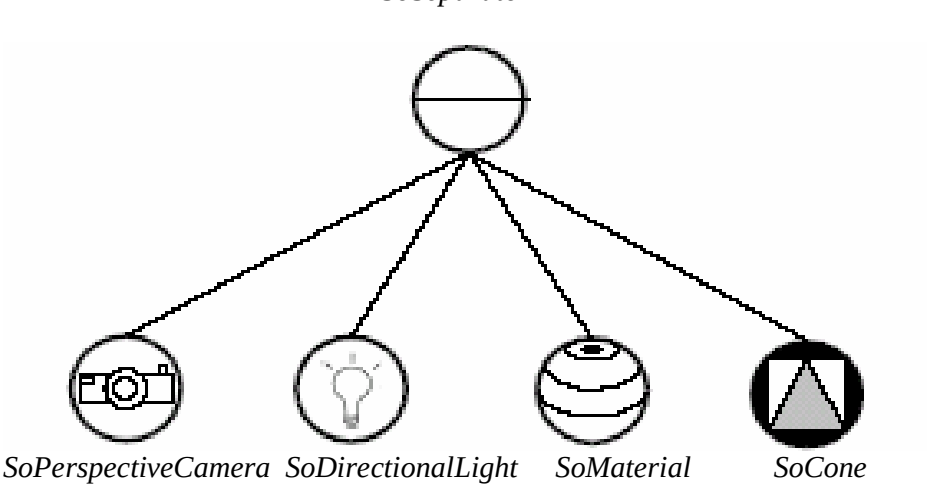
```
SoWin::show(myWindow);
```

Inventor-Ereignisschleife

```
SoWin::mainLoop(); }
```

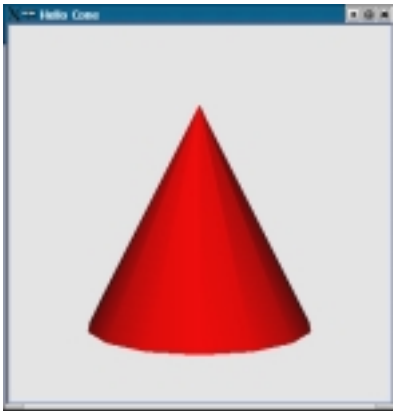
Zunächst wird SoWin initialisiert, womit implizit auch die Szenendatenbank initialisiert wird. Anschließend wird der Wurzelknoten erzeugt (vom Typ SoSeparator, abgeleitet von SoGroup). Dieser Gruppe werden mit der Methode addChild ein Kamera-, Licht-, Material- und der Konus-Objekt hinzugefügt. Das Programm erzeugt den in der Abbildung 7 dargestellten Szenengraphen.

Abbildung 7: Szenegraph des Beispielprogramms



Anschließend wird die Szene in einer SoWinRenderArea gerendert, wie in Abbildung 8 gezeigt. Bei der Konstruktion einer Szene sind Kamera und Licht von großer Bedeutung.

Abbildung 8: Bildschirmkopie des Beispiels – Konus



Die Kameraposition bestimmt, von welcher Position aus und mit welcher Perspektive man die Szene betrachtet. In diesem Beispiel wird die Kamera so positioniert, dass die ganze Szene sichtbar ist. Ohne Beleuchtung ist eine Szene praktisch unsichtbar, deshalb sollte in jedem Szenengraphen mindestens ein Beleuchtungsobjekt existieren. Das Material-Objekt wird in dieser Szene dazu benutzt, den Konus rot einzufärben.

3.2. Erweiterung des Beispiels um eine Animation

Es soll gezeigt werden, wie eine in Open Inventor als Engine bezeichnete Animationskomponente für eine kontinuierliche Rotation des Konus verwendet werden kann. Dazu wird in den Szenengraph ein **SoRotationXYZ**-Objekt eingefügt und dessen Attribut angle, das den Winkel einer Rotation enthält, mit einem Engine-Objekt, hier **SoElapsedTime,** verknüpft. Beide Objekte sind in Abbildung 9 dargestellt.
SoElapsedTime reagiert auf Veränderungen der Systemzeit und ändert als Reaktion darauf das verknüpfte Attribut angle. Der Winkel wird dabei ständig erhöht, so daß die Veränderung eine kontinuierliche Drehung des Konus bewirkt. Bei jeder Veränderung dieses Attributs wird die Szene automatisch neu gerendert. Im Allgemeinen wird nur der Teil neu gerendert, der sich durch die Veränderung des Szenengraphen tatsächlich verändert hat.
Einfache Animationen können in Open Inventor also mit Standard-Objekten erzeugt werden. Das hat unter anderem auch den Vorteil, dass die Animation als Teil des Szenengraphen in Dateien geschrieben werden kann. Außerdem ist für einfache Animationen relativ wenig Code notwendig. Die entsprechenden Code-Sequenzen:

Zunächst erfolgen die gleichen Schritte, hier wird daher das Includieren der Header-Dateien ausgenommen:

```
main(int , char **argv) {
Widget myWindow = SoWin::init(argv[0]);
if (myWindow == NULL) exit(1);
SoSeparator *root = new SoSeparator;
root->ref();
SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
root->addChild(myCamera);
root->addChild(new SoDirectionalLight);
```

Rotations-Objekt einfügen

```
SoRotationXYZ *myRotXYZ = new SoRotationXYZ;
root->addChild(myRotXYZ);
```

Rotation um X-Achse definieren

```
myRotXYZ->axis = SoRotationXYZ::X;
SoElapsedTime *myCounter = new SoElapsedTime;
```

Engine mit Attribut “angle” verbinden

```
myRotXYZ->angle.connectFrom(&myCounter->timeOut);
```

Materialknoten definieren

```
SoMaterial *myMaterial = new SoMaterial;
myMaterial->diffuseColor.setValue(1.0,0.0,0.0);
```

Einhängen der Knoten in den Szenegraphen:

```
root->addChild(myMaterial);
root->addChild(new SoCone);
SoWinRenderArea *myRenderArea = new SoWinRenderArea(myWindow);
```

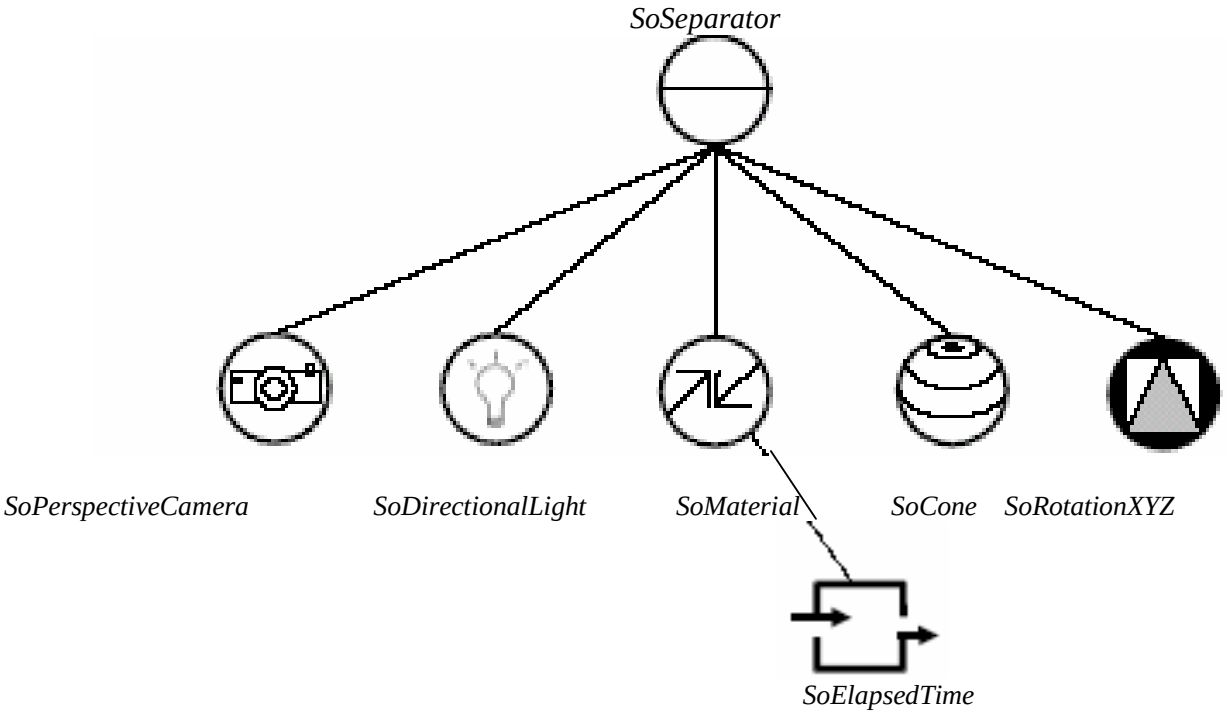
Kameraposition setzen

```
myCamera->viewAll(root, myRenderArea->getViewportRegion());
```

Dann die abschließenden Schritte

```
myRenderArea->setSzenegraph(root);
myRenderArea->setTitle("Engine Spin");
myRenderArea->show();
SoWin::show(myWindow);
SoWin::mainLoop(); }
```

Abbildung 9: Szenegraph des erweiterten Beispiels – Konus mit Rotation



Anmerkung:

Eine Zusammenfassung zum Kapitel über Open Inventor erfolgt im Kapitel 5 in Form des Vergleichs der Renderingsysteme.

4. Java 3D

4.1. Einführung

Java 3D ist eine von Sun Microsystems spezifizierte Schnittstelle zur Programmierung von Anwendungen mit der Java Plattform zur Darstellung von dreidimensionalen Informationen.
Java 3D ist nur eine Schnittstelle und stellt an sich keine Funktionalität zur Verfügung. Um Java 3D real nutzen zu können, benötigt man eine Implementierung dieser Schnittstelle, die oftmals plattformabhängig ist. Somit ist es möglich, Code zu schreiben, der plattformunabhängig ist, aber auf dem jeweiligen System eine angepasste Java3D-Implementierung als Aufsetzpunkt benötigt.

Die API ist mit folgenden Zielen entwickelt worden:

- Verfügbarkeit einer möglichst großen Menge an Features, ohne jedoch das System dadurch zu überladen
- Möglichkeit der schnellen und einfachen Entwicklung auf der Java3D-Plattform
- umfassende Erweiterbarkeit hinsichtlich der zu ladenden Datenformate, um möglich viele 3D-Formate zu unterstützen. Diese Erweiterbarkeit erlaubt es Drittanbietern, Module für beliebige Dateiformate zu erstellen.

4.1.2. Java 3D in Applikationen

Java 3D bietet sich für kleinere 3D-Elemente in Java-Anwendungen an, bei denen Performance nicht so wichtig ist wie Kompatibilität und Uniformität der Entwicklungsumgebung. Hier sind z.B. folgende Anwendungen möglich: kleinere 3D Spielereien wie drehende Logos, dreidimensionale Stadtpläne, Gebäudeorientierungsapplikationen und durchaus auch technische und medizinische Visualisierungen, falls neben der Performance schnelle Entwicklungszeiten und Plattformunabhängigkeit gefragt sind, wie beispielsweise die Visualisierung mechanischer Phänomene für Lehrzwecke.

4.1.3. Java 3D in Applets

Hier gibt es verschiedene Anwendungen, wobei zu beachten ist, dass im Browser ein Java-Plug-In und Java 3D installiert sein muss. Es bieten sich in der Praxis beispielsweise Anwendungen zur Darstellung von Waren und Produkten an, wobei Java mit seiner größeren Flexibilität hier auf die Konkurrenz von 3D-Formaten wie VRML trifft. Es bieten sich ebenso Online-Darstellungen als Anwendungsfeld an, da hier ebenfalls eine nicht hohe Performance erforderlich ist.
Eventuell bietet sich Java 3D auch für MIDlets, also Java Applets für Mobile Devices an, um dort die für UMTS angekündigten Features zu realisieren.

4.1.4. Existierende Implementierungen

Sun selbst hat eine Implementierung von Java 3D erstellt, die direkt auf Open GL aufsetzt und stellt diese zum kostenlosen Download in der Version 1.2 für Windows und Solaris zur Verfügung (J3DOPENGL1.2.) Auch die ältere Version 1.1.3 ist für die selben Plattformen (J3DOPENGL1.1.3) verfügbar. Unter LINUX steht eine Implementierung von Java 3D, die auf OpenGL aufbaut, zur Verfügung. Unter DIRECT3D stellt Sun eine Implementierung der Java 3D API in der Version 1.2 für MS Windows zur Verfügung, die auf Direct3D aufsetzt.

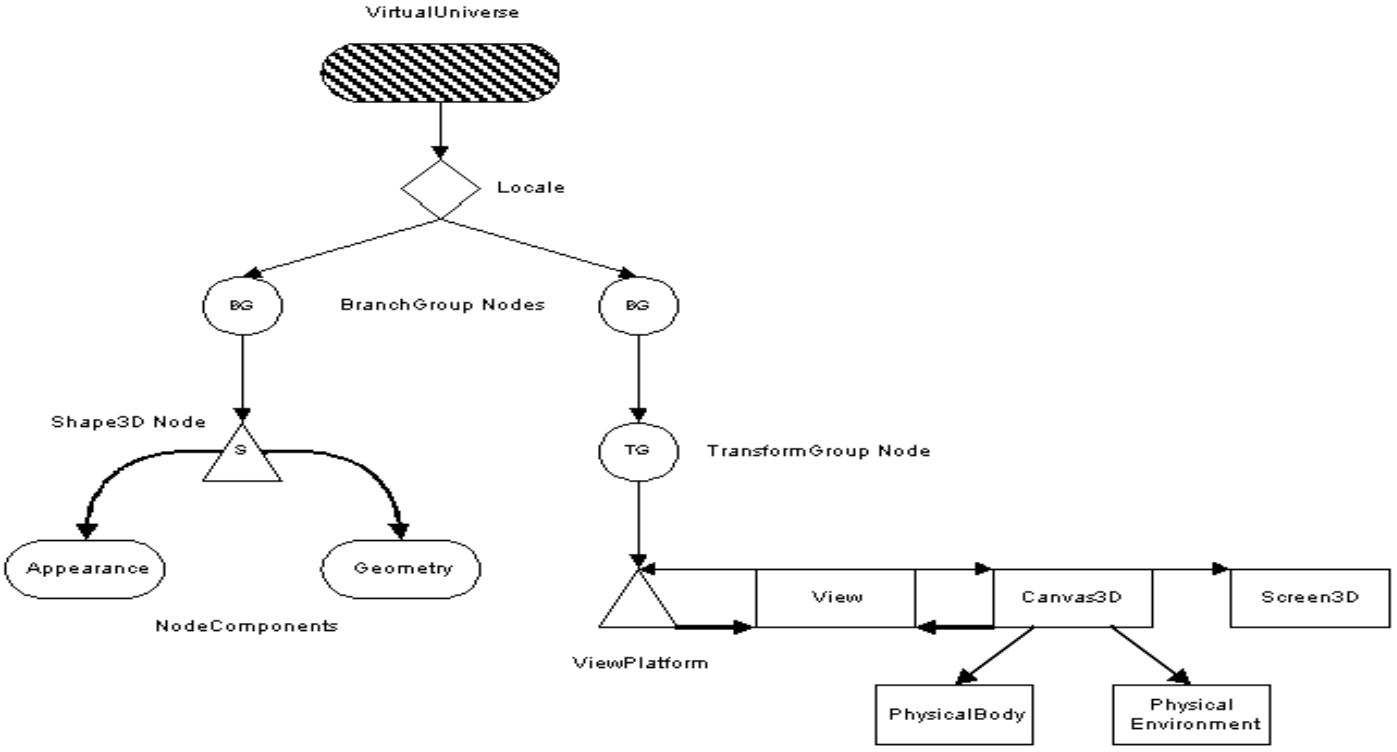
4.1.5. Dateiformate

Formate verschiedener Hersteller werden mit Hilfe von „formatloader“ unterstützt:
3DS (3D-Studio), PDB (Protein Data Bank), DEM (Digital Elevation Map), IOB (Imagine), COB (Caligari trueSpace), OBJ (Wavefront), DXF (Drawing Interchange File), AutoCAD, VRML 97, PLAY, X3D, OpenFLT, LWS (Lightwave Scene Format), LWO (Lightwave Object Format), OBJ (Wavefront), NFF (WorldToolKit), VTK (Visual Toolkit), VRT.

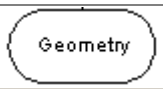
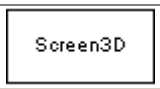
4.2. Grundlegendes Konzept: Der Szenegraph
4.2.1. Struktur
4.2.1.1. Aufbau des Szenegraphen

Der Szenegraph ist das Hauptkonzept, um 3D-Szenen mit Java3D darzustellen. Er dient der Modellierung der 3D-Information und enthält alle für die Darstellung der Szene notwendigen Objekte. Der Szenegraph in Java 3D ist ein gerichteter Graph ohne Zyklen und somit ein Baum, bei dem zwischen Knoten und Blättern unterschieden werden kann. Die Zustände verschiedener Äste dieses Baumes sind unabhängig voneinander. Diese Struktur erlaubt es dem Java3D-Renderer, Objekte der Szene völlig getrennt voneinander abzuarbeiten und zu zeichnen. Die Hierarchie des Szenegraphen unterstützt die örtliche Gruppierung der einzelnen Objekte und ist ein sinnvolles Konzept, um Dinge wie Kollisionserkennung oder die Auswahl der für die View-Plattform jeweils sichtbaren Objekte zu ermöglichen. Der Zustand eines Blattes, also des Objektes, welches letztendlich grafisch auf dem Bildschirm erscheint, ergibt sich, bis auf wenige Einschränkungen, aus der Traversierung des Pfades von diesem Blatt zur Wurzel des Baumes (Locale bzw. virtual Universe). Dies ist ein Vorteil im Vergleich zu älteren APIs, da nun ein Parallel-Rendering möglich ist.

Abbildung 1: Beispiel eines einfachen Szenengraphs mit Legende



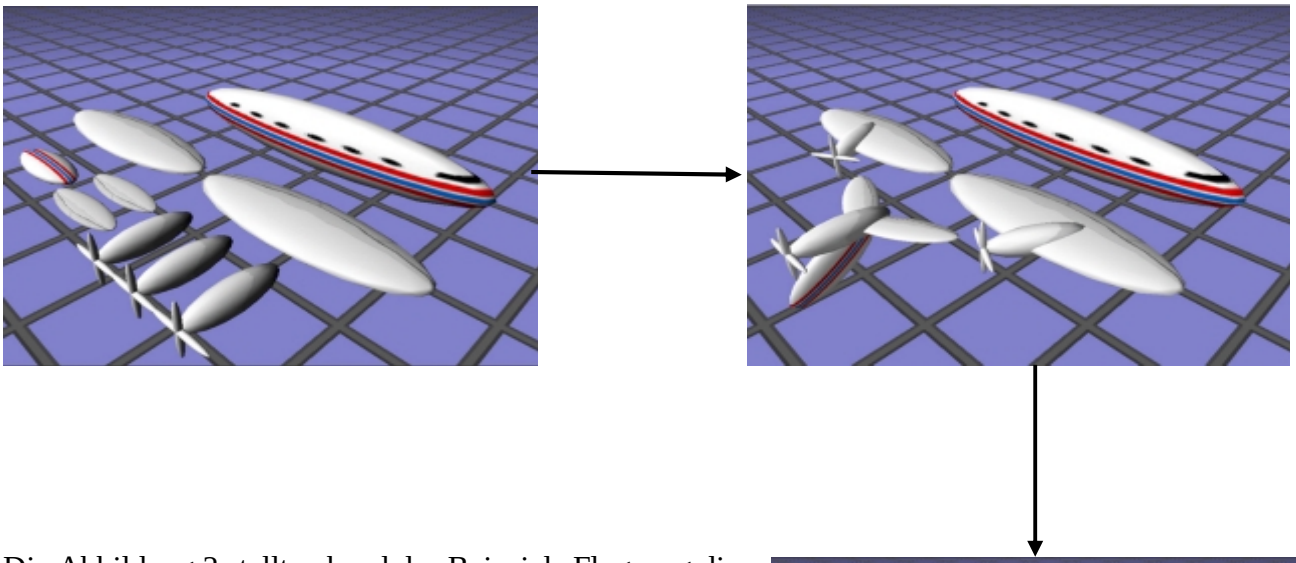
Legende

Symbol	Bedeutung	
schraffierte Fläche	VirtualUniverse Objekt	
Quadrat	Locale Objekt	
Kreis	Group Objekt	
Dreieck	Leaf Objekt	
Ellipse	NodeComponet Objekt	
Rechteck	andere Objekte	
Pfeil (dünn)	Eltern-Kind-Verbindung	
Pfeil (dick)	Verweis	

Die Abbildung 1 zeigt ein Beispiel für den Aufbau eines typischen Szenegraphen und die in der Legende darunter erläuterten Zeichen werden standardmäßig in Java 3D für die Darstellung des Szenegraphen verwendet.
Ausgehend vom VirtualUniverse-Objekt verzweigt sich der abgebildete Graph in unterschiedliche Richtungen.
Das Locale-Objekt verwaltet dabei die einzelnen Äste des Graphen, die als Wurzelelement eine BranchGroup besitzen. Eine Locale kann grundsätzlich „n“ Äste verwalten, üblich ist aber die Aufteilung in zwei Äste, wobei der linke Ast den Inhalt (Content) und der rechte Ast die Ansicht (View) der 3D-Welt repräsentieren.
Der Schwerpunkt dieses Dokuments liegt dabei auf der Gestaltung von Inhalt (Content), wie z. B. das Erzeugen von Geometrien und Animationen.
Der Ansichts-Ast (View Branch) legt fest, wie das virtuelle Universum dargestellt werden soll (Rendering). Java 3D verfolgt dabei die Philosophie des 'write-once-run-anywhere' Prinzips. Ohne den Inhalt zu ändern, passt sich der View den physikalischen Gegebenheiten der realen Welt an. So wäre es z.B. möglich, durch Aufbau eines entsprechenden View-Astes ein HMD (Head Mounted Display) zu unterstützen.
Der View-Ast in Abbildung 1 ist dagegen einfach gehalten. Die Geometrie in dieser 3D-Welt wird durch Zentralprojektion auf einer 2-dimensionalen Zeichenfläche (Canvas) dargestellt.

Die Projektion von 3D-Körpern auf eine 2D-Fläche ist notwendig, um 3D-Objekte auf dem Bildschirm zu realisieren. Der Betrachter des virtuellen Universums schaut die z-Achse hinab direkt auf den Ursprung des 3D-Koordinatensystems.

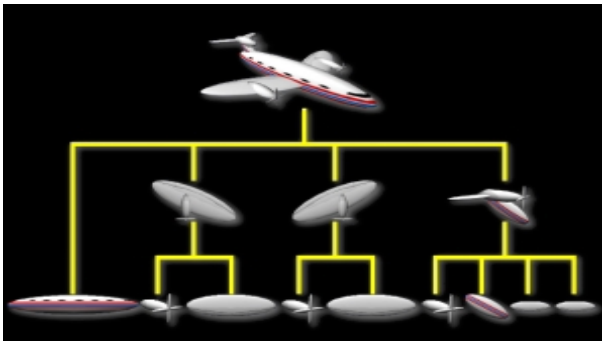
Abbildung 2: Beispiel Flugzeug



Die Abbildung 2 stellt anhand des Beispiels Flugzeug die Zerlegung des Ganzen in einzelne Komponenten und diese wiederum nach Zusammengehörigkeit entsprechender Funktionsmerkmale gruppiert. Entsprechend dieser Struktur wird der dazugehörige Baum, der Szenegraph erstellt, wie Abbildung 3 verdeutlicht.

Abbildung 3: Szenengraph für das Flugzeubeispiel

Ein Szenengraph enthält:



- Shapes (geometry and appearance)
- Groups und transforms
- Lights
- Fog und backgrounds
- Sounds und sound environments
- Behaviors
- View platforms (viewpoints)

4.2.1.2. Rendering

Java 3D rendert den Szenengraphen, dieser spezifiziert den Inhalt, nicht die Reihenfolge, wie gerendert wird. Die Rendering-Reihenfolge übernimmt Java3D.

Java3D basiert auf asynchronen Threads, die unabhängig voneinander arbeiten:

- Graphik-Rendering
- Sound-Berechnung
- Animation der Eigenschaften (behavior)

Die drei Rendering-Modi von Java 3D

Java 3D bietet die Möglichkeit, mit drei verschiedenen Rendering-Modi ("rendering modes") zu arbeiten: Immediate-Mode, Retained-Mode und Compiled-Retained-Mode.
Jeder Modus ermöglicht, mit einem unterschiedlichen Grad an Automatisierung zu arbeiten und somit unterschiedliche Optimierungen der 3D-Ausgabe durch das Java 3D System durchzuführen. Hier eine kurze Beschreibung der einzelnen Modi:

Der Immediate-Mode: Dieser Modus lässt wenig Freiraum für globale Optimierungen des Szenengraphen durch das Java3D-System. Der Entwickler arbeitet recht "hardwarenah" und bedient sich elementarer Ausgabefunktionen. Eine Applikation muss in diesem Fall eine Ausgabemethode mit einer Menge von Punkten, Linien und Dreiecken implementieren, wobei diese direkt durch den Java 3D Renderer dargestellt wird. Der Entwickler kann diese Liste von Punkten, Linien und Dreiecken beliebig konstruieren und verwalten. Der Immediate-Mode ist von der softwaretechnologischen Seite am wenigstens interessant.

Der Retained-Mode: In diesem Modus muss der Entwickler einen Szenengraphen konstruieren. Dabei wird dieser Graph mit bestimmten Elementen gefüllt, die visuelle Objekte und ihr Verhalten in der Szene (z.B. Animationen) beschreiben. Man hat dabei den vollen Zugriff auf die Struktur des Szenengraphen und seine Objekte und kann diese jederzeit verändern, löschen oder sogar neue hinzufügen. Der Retained-Mode kann als eine Mischung zwischen dem Immediate-Mode und dem Compiled-Retained-Mode angesehen werden.

Der Compiled-Retained-Mode: Auch in diesem Modus muss man einen Szenengraphen konstruieren und mit den dazugehörigen Elementen füllen. Der Unterschied zu dem einfacheren Retained-Mode besteht in dem beschränkten Zugriff auf die Objekte im Szenengraphen, der durch bestimmte "Zugriffsparameter" geregelt wird. Dadurch wird eine effiziente Optimierung der 3D-Ausgabe durch das Java3D-System ermöglicht, wobei der Szenengraph in eine interne Repräsentation konvertiert wird – der Vorgang der Kompilierung. Im Compiled-Retained-Mode kann somit die höchste Leistungsstufe der Optimierungen durch Java 3D erreicht werden.

4. 2.1.3. Terminologie von Java 3D

Ein Szenengraph besteht aus Knoten, Komponenten und Kanten.

Knoten: Datenelement, Instanz einer Java3D-Klasse, unterteilt in Group-Nodes und Leaf-Nodes
Komponenten: definieren Eigenschaften von Knoten
Kante: Verbindung zwischen den Datenelementen: Eltern-Kind-Beziehung, Referenz
Leaf nodes: Knoten ohne Kind, z.B. Shapes, Lights, Sounds, etc., Animation behaviors
Group nodes: Knoten mit Kindern, z.B. Transforms, Switches, etc.
Node component: Eine Menge von Attributen, die ein Knoten haben kann: z.B. Geometrie eines Shape, Farbe eines Shapes, Sound-Daten, die zu spielen sind

4.2.2. Szenegraph-Objekte

Szenegraph-Objekte sind von der abstrakten Klasse SceneGraphObject abgeleitet und bilden im Zusammenschluss den Szenegraph. Sie können über set- und get-Methoden manipuliert werden und unter einem schon vorhanden Scenegraph-Objekt in den Graphen eingehängt werden.
Bevor man die 3D-Szene dem Virtual Universe hinzufügt und somit sichtbar macht, besteht die Möglichkeit einer Geschwindigkeitsoptimierung durch Kompilieren des Subgraphen in ein internes Format.
Der dynamische Zugriff auf den Szenegraphen nach der Kompilierung ist nur möglich, wenn hierfür vorher bestimmte capability bits gesetzt worden sind. Jede Instanz eines Szenegraph-Objektes hält seine individuellen capability bits. Diese müssen so gesetzt sein, dass die zur Laufzeit anfallenden Manipulationen, wie z.B. Translationen oder Geometrieänderungen, vom Java3D-Renderer zugelassen werden.

4.2.3. Szenegraph-Superstructure-Objekte

Java 3D definiert zwei Szenegraph-Superstructure-Objekte, das Virtual Universe und das Locale.
Das Virtual Universe enthält eine Liste aller Locale-Objekte und stellt somit die höchste Aggregationsstufe dar. Es umfasst den kompletten modellierten 3D-Raum für eine Anwendung. Die physikalischen Ausmaße erstrecken sich bis zu mehreren hundert Mrd. Lichtjahren, wobei die kleinsten beschreibbaren Objekte eine geringere Größe als ein einzelnes Proton aufweisen können. Dies ist durch ein hochauflösendes Koordinatensystem mit 256 Bit pro Achse möglich, in welches ein oder mehrere kleine lokale Koordinatensysteme, die Locales, eingehängt werden.
Locales dienen als Container für Subgraphen, welche über einen BranchGroup-Knoten dem Locale hinzugefügt werden. Die Koordinaten eines Locales sind relativ zu den hochauflösenden Koordinaten, an denen das Locale in das Virtual Universe eingehängt ist. Der Wert 1.0 ist in Java 3D generell als exakt 1 Meter definiert.

4.2.4. Szenegraph-Viewing-Objekte

Es existieren 5 Viewing-Objekte, welche nicht Bestandteil des Szenegraphen sind, jedoch über die View-Plattform mit ihm verbunden sind. Sie dienen in erster Linie als Schnittstelle in die physikalische Welt.

Das **Canvas3D-Objekt** ist die Zeichenfläche, auf der der Java3D-Renderer die Projektion des 3D-Models ausgibt.
Das **Screen3D-Objekt** kapselt alle mit dem physikalischen Bildschirm verbundenen Parameter wie Höhe, Breite und Raster-Auflösung.
Das **PhysicalBody-Objekt** beschreibt die notwendigen physischen Daten des menschlichen Körpers wie z.B. Kopfposition und Position der Augen.
Das **PhysicalEnvironment-Objekt** kapselt zusätzliche Informationen wie Kalibrierungswerte für Headtracker oder Datenhandschuhe.
Das **View-Objekt** ist das zentrale Java3D Objekt, welches alle Aspekte der Sicht auf das Modell spezifiziert bzw. referenziert.

Java 3D unterstützt auch mehrere simultan aktive Viewing-Objekte, die wiederum jeweils mehrere Zeichenflächen (Canvas3D) ansteuern können.

4.2.5. Szenegraph-Knoten im Detail

Es existieren verschiedene Gruppierungsknoten, welche der Zusammenfassung beliebig vieler Subgraphen dienen. Allen gemeinsam ist die Eigenschaft mittels der Methoden setChild, insertChild, addChild oder removeChild eine einheitliche Manipulationsweise bereitzustellen. Weiterhin ist es möglich, für jeden Gruppierungsknoten bestimmte Kollisionsgrenzen zu definieren.
BranchGroup-Knoten sind einfache Wurzelknoten zur Aggregation von Objekten zu einem kompilierbaren Subgraphen. Über den BranchGroup-Knoten können Teile des Szenegraphen zur Laufzeit eingehängt (addChild) und auch wieder entfernt werden (detach).
Im Normalfall werden BranchGroup-Knoten dem Locale hinzugefügt, können aber auch Bestandteil eines Subgraphen auf niedrigerer Ebene sein.
Der **TransformGroup-Knoten** dient zur örtlichen Verschiebung bzw. Positionierung, Skalierung und Drehung von Objekten bzw. Subgraphen. Über die Methoden setTransform und getTransform wird dem Transformationsknoten eine Transformation (double-precision matrix) zugeordnet.
Die beste Berechnungs-Performance wird erreicht, wenn alle Matrizen nicht willkürliche Transformationen sondern Rotation, Translation und einheitliche Skalierung spezifizieren.
Der Effekt von Transformationen entlang eines Pfades ist kumulativ (composite model transformation = CMT); das bedeutet, dass alle Transformationen in diesem Pfad konkateniert und somit die lokalen Koordinaten des Blattes in globale Koordinaten der virtuellen Welt umgerechnet werden.
Mittels eines **DecalGroup-Knoten** ist es möglich zu bestimmen, in welcher Reihenfolge die Kinder des Knotens gerendert werden sollen. Dies ist nützlich, wenn in der 3D-Szene ansonsten z-Buffer-Kollisionen auftreten würden, z.B. bei Dingen die flach auf einer Tischoberfläche liegen.
Der **SwitchGroup-Knoten** macht es möglich, während der Laufzeit dynamisch aus einer Anzahl von Subgraphen diejenigen auswählen zu können, die vom Renderer gezeichnet werden sollen. Ein SwitchGroup-Knoten spezifiziert jedoch nicht die Darstellungsreihenfolge.
Der **SharedGroup-Knoten** bietet einen Mechanismus an, um einen Subgraph mittels eines Link-Knoten an verschiedenen Stellen des Baumes zu verwenden und so redundante Modellierung unnötig zu machen.

4.2.6. Blätter

Blätter definieren atomare Bestandteile des Szenegraphs, wie z.B. geometrische Objekte, Licht und Sounds.
Der wichtigste Blattknoten in Java 3D ist das **Shape3D**, welches jeweils ein oder mehrere geometrische Beschreibungen der selben Äquivalenzklasse sowie zusätzlich ein Appearance-Objekt enthält. Die Appearance umfasst Attribute wie Farbe, Material und Textur.
Der **OrientedShape3D-Knoten** ist eine Spezialform der normalen Shape3D's, bei dem die positive z-Achse des geometrischen Objektes immer zum Betrachter zeigt.
Das **Background-Blatt** definiert eine Hintergrundfarbe oder ein Hintergrundbild, welche bei Durchdringung des Aktivierungsvolumens des Knotens durch den Sichtzylinder der View-Plattform (view frustum) angezeigt werden.
Die Blätter **Clip** und **ModelClip** dienen der Begrenzung des sichtbaren Bereiches des Modells.

Über die Blätter **ExponentialFogNode** und **LinearFogNode** ist es möglich, Nebel beliebiger Farbe zu simulieren.
BehaviorNode-Blätter sind integraler Bestandteil von Java 3D und dienen zur automatisierten, dynamischen Manipulation des Szenegraphen zur Laufzeit.
Diese Verhaltensreaktionen umfassen Dinge wie Tastatureingaben, Mausereignisse, Selektion oder Kollision von Objekten und einiges mehr.

4.2.7. Beleuchtung

4.2.7.1. Lichtquellenarten

Zur **Beleuchtung** einer Szene existieren im Wesentlichen vier verschiedene Lichtquellenarten:

1. ambientes Licht (AmbientLightNode)
2. gerichtetes Licht mit Ursprung im Unendlichen (DirectionalLight)
3. ungerichtete Punktlichtquelle (PointLightNode)
4. Strahler, welcher eine kreisförmige Ausleuchtung realisiert (SpotLightNode)

Aufgrund der Bedeutung für das realistische Aussahen einer 3D-Szene und den interessanten Möglichkeiten bei Java 3D möchte ich im Gegensatz zur Beschreibung von Open Inventor hier näher auf dieses Thema eingehen.

1. **Ambient Light (Umgebungslicht):** Dieses Licht hat die gleiche Intensität an allen Orten und in alle Richtungen. Es handelt sich um das Licht, welches von anderen Objekten in der Szene reflektiert wird. Java 3D erlaubt ausschließlich die Festlegung der Farbe dieser Lichtart und natürlich, ob die Lichtquelle an- oder ausgeschaltet werden soll, z.B. über den Konstruktor: AmbientLight(boolean lightOn, Color3f color)

2. **Directional Light (Umgebungslicht):** Diese Lichtart modelliert Lichtquellen, die sehr weit entfernt sind (z.B. die Sonne). Alle Lichtstrahlen haben die gleiche definierte Richtung, sie sind parallel. Die Lichtstärke variiert nicht, egal wie weit ein Objekt von der "Lichtquelle" entfernt ist. Die einstellbaren Parameter sind hier die Lichtfarbe und die Richtung des Lichts, z.B. im Konstruktor: DirectionalLight(boolean lightOn, Color3f color, Vector3f direction)

3. **Point Light (Punktlicht):** Ein Punktlicht hat im Gegensatz zu den beiden vorgenannten Lichtarten eine definierte Lichtquelle. Das Licht strahlt in alle Richtungen aus und wird außerdem mit der Entfernung von der Lichtquelle schwächer. Einstellbare Parameter sind

- Lichtfarbe
- Position der Lichtquelle im Raum
- Abschwächung

Ein Konstruktor: PointLight(Color3f color, Point3f position,Point3f attenuation)

4. **Spot Light:** Diese Lichtklasse ist von PointLight abgeleitet. Sie fügt zu den geerbten Eigenschaften noch eine definierte Richtung und einen Konzentrationsparameter hinzu. Damit entsteht ein kegelförmiger Lichtbereich, wobei das Licht zu den Rändern des Kegels hin in Abhängigkeit vom Konzentrationsparameter schwächer wird. Mit SpotLight lassen sich vom Menschen geschaffene Lichtquellen wie z.B. Lampen modellieren. Ein Konstruktor:
SpotLight (Color3f color, Point3f position, Point3f attenuation, Vector3f direction,
floatspreadAngle, float concentration)

4.2.7.2. Licht und Schatten in Java 3D

Java 3D schattiert visuelle Objekte basierend auf einer Kombination von Materialeigenschaften und den Lichtern im virtuellen Universum. Die Schattierung resultiert aus der Anwendung eines Lichtmodells auf die visuellen Objekte im Beisein von Lichtquellen. Da viele Aspekte des Java3D-Licht-Schatten-Modells auf OpenGL basieren, können auch in den OpenGL Dokumentationen entsprechende Informationen nachgelesen werden.

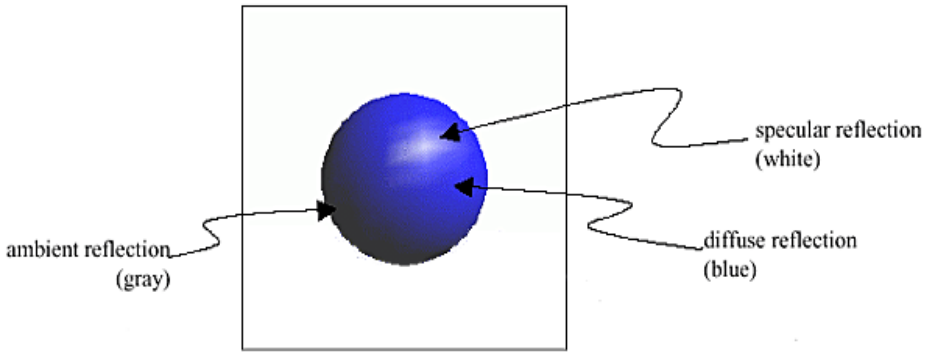


Abbildung 4: Typen von Reflektionen

Die Abbildung 4 zeigt eine Kugel, die in Java 3D gerendert wurde. Die drei Typen von Reflektion können hier beobachtet werden: Der dunkelste Teil der Kugel reflektiert nur das Umgebungslicht, welches keine spezifische Richtung hat, sondern gleichermassen im ganzen Raum vorhanden ist (ambient reflection). Das Zentrum der Kugel wird von diffusem und Umgebungslicht erhellt. Bei einer blauen Kugel und einem weissen Licht ist die diffuse Reflektion (diffuse reflection) blau. Der hellste Teil der Kugel ist das Ergebnis einer spiegelnden Reflektion (specular reflection) mit Umgebungslicht- und diffusen Reflektionen.

4.2.7.3. Vorgehen zur Beleuchtung von Objekten

Eine Reihe von Schritten ist erforderlich, um die Beleuchtung von visuellen Objekten im virtuellen Universum zu erreichen. Neben der Erzeugung und Anpassung der Licht-Objekte muß jedes Licht zum Szenengraphen hinzugefügt werden und einen Wirkungsbereich (bounds) haben. Jedes visuelle Objekt, das beleuchtet werden soll, muss Oberflächennormalen und Materialeigenschaften haben.

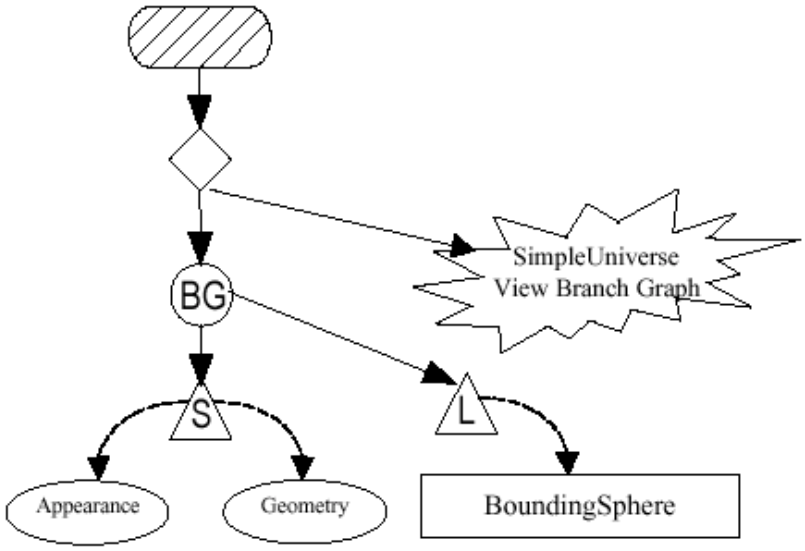


Abbildung 5: Szenegraph einer Kugel mit Standard-Materialeigenschaften

Abbildung 5 zeigt, daß im Szenengraphen die Licht-Objekte prinzipiell an beliebiger Stelle eingefügt werden können. Man muss nur beachten, dass der zugewiesene Wirkungsbereich sich auf das lokale Koordinatensystem des Group-Objektes bezieht, dem das Licht-Objekt zugewiesen wurde.

4.2.7.4. Materialeigenschaften

Entscheidend für das Erscheinungsbild eines Objektes in einer beleuchteten 3D Szene sind neben den vorhandenen Lichtquellen auch die folgenden Materialeigenschaften, die hier der Vollständigkeit des Themas Beleuchtung halber genannt werden:

- "ambient color" bestimmt die Farbe des reflektierten Lichtes für "ambient reflection".
- "diffuse color" bestimmt die Farbe des reflektierten Lichtes für "diffuse reflection".
- "specular color" bestimmt die Farbe des reflektierten Lichtes für "specular reflection".
- "emissive color" bestimmt die Farbe des Lichtes, das das Material "aus eigener Kraft" aussendet, d.h. für den Fall, dass es leuchtet.
- "shininess" ist der Glanz der Materialoberfläche.

Sämtliche Parameter lassen sich zum Beispiel über den Konstruktor einstellen:
Material(Color3f ambientColor, Color3f emissiveColor, Color3f diffuseColor, Color3f specularColor, float shininess)

4.2.8. Animation und Interaktion

4.2.8.1. Behavior-Klasse

Die zentrale Klasse für die Realisierung von Interaktion, Animation und Kollisionserkennung ist die Java3D-Klasse Behavior (Behavior, *engl.* = Verhalten). Die Behavior-Knoten werden „wach“, sobald ein bestimmtes Ereignis eintritt. Diese verändern dann mit einer speziellen Methode den Szenegraphen oder ermitteln z.B. selektierte Objekte. Ereignisse, die durch den Benutzer ausgelöst wurden, der Ablauf einer gewissen Zeitspanne und andere Kriterien dienen als Aufwachkriterien.
Die Behavior-Klasse kann zur Animation von visuellen Objekten verwendet werden, ermöglicht aber praktisch, jedes Attribut eines visuellen Objektes zu verändern.
Genauso ermöglichen Behaviors das interaktive Verändern von Eigenschaften von Szenen und Objekten. Dabei kann man die vordefinierten Java3D-Behaviors verwenden, aber auch eigene entwerfen. Wird ein Behavior mit einem visuellen Objekt assoziiert, dann werden die Attribute wie z.B. Position, Orientierung und Farbe automatisch angepasst, so dass der Entwickler sich um diese nicht mehr kümmern muss. Im Allgemeinen spricht man von Animationen, wenn das Behavior in Abhängigkeit von der Zeit aktiviert wird und von Interaktion, wenn dieses aufgrund von Eingaben des Benutzers aktiviert wird.
Jedes visuelle Objekt in einer 3D-Szene kann sein eigenes Behavior haben. Es ist aber auch möglich, einem Objekt mehrere verschiedene Behaviors zuzuordnen. Zu beachten ist hierbei, daß Behaviors meist recht rechenintensiv sind und die Rendering-Ausgabe erheblich verlangsamen können. Um diesen Problem entgegenzuwirken, ermöglicht Java 3D die Definition von bestimmten Grenzen für die Behaviors. Man spricht in diesem Kontext von dem sog. Wirkungsbereich ("scheduling region") der Behaviors. Ein Behavior wird erst dann aktiv, wenn sich der Aktivierungsbereich ("activation volume") der ViewPlatform mit dem Wirkungsbereich des Behaviors schneidet. Diese Vorgehensweise entspricht dem Prinzip "Das was man nicht sieht, gibt es nicht". Bei diesem Vorgehen kann die 3D-Darstellung eines virtuellen Universums mit vielen Behaviors viel effizienter erfolgen.

Die Behavior-Klasse unterstützt des Weiteren die Auswahl von Objekten. Dies geschieht über die Definition eines eigenen Behavior-Objekts, das sich für den Empfang von einfachen Fenster-Events registriert.
Mit Hilfe der Methode processStimulus() kann dann das ausgewählte Objekt ermittelt werden. Da es auch möglich ist, den Pfad des Objekts vom Locale bis zu seiner Position zu bestimmen, kann nach dem Auswählen auch seine Position und Orientierung in der virtuellen Welt verändert werden. Es existieren auch drei vorgefertigte „Picking“-Klassen, die dem Entwickler die Arbeit mit der Auswahl von Objekten sehr vereinfachen, siehe Abschnitt 4.3.7.

4.2.8.2. Animation mit Interpolatoren

Die sog. Interpolatoren sind eine Anzahl von vordefinierten Kernklassen aus dem Java3D-Package, die von der Behavior-Klasse abgeleitet sind. Ein solcher Interpolator manipuliert bestimmte Parameter eines Szenengraphobjektes. Eine Übersicht über die wichtigsten Interpolatoren:

- RotationInterpolator für Rotationen
- PositionInterpolator für Translationen
- ScaleInterpolator für Skalierungen
- ColorInterpolator für Farbveränderungen
- TransparencyInterpolator für Transparenzveränderungen

Zu beachten ist hierbei, daß all diese Interpolatoren sich auch durch die gewöhnliche Behavior-Klasse konstruieren lassen, jedoch vereinfachen Interpolatoren die Implementierung von Behaviors erheblich. Außerdem ist es möglich, mehrere Interpolatoren zu kombinieren.

4.2.8.3. Interaktion

Die Java3D-API enthält mehrere fertige Behavior-Klassen, mit denen der Benutzer interaktiv über die Tastatur oder Maus auf die dargestellte 3D-Szene einwirken kann:

KeyNavigatorBehavior: Ermöglicht unter anderem die Steuerung von Rotation und Translation über die Tastatur.
MouseBehavior mit Unterklassen: Die Unterklasse MouseRotate ermöglicht über "Mouse-Dragging" (Maustaste halten und Maus ziehen) mit der linken Maustaste die Rotation eines Objektes an der Stelle an der es sich befindet. MouseTranslate bietet über "Mouse-Dragging" mit der rechten Maustaste die Verschiebung (Translation) parallel zur "image plate", also parallel zur Zeichenfläche. MouseZoom realisiert über "Mouse-Dragging" mit einer Maustaste und der Alt-Taste oder einfach nur der mittleren Maustaste (falls vorhanden) eine Zoomfunktion durch die Verschiebung orthogonal zur "image plate".
PickMouseBehavior mit Unterklassen: Die Unterklassen von PickMouseBehavior bieten die gleiche Funktionalität wie die Unterklassen von MouseBehavior, mit dem Unterschied, dass der Benutzer mit dem Drücken der Maustaste über einem bestimmten Objekt in der Szene dieses für die gewünschte Transformation auswählt. Die durch das Behavior implementierte Transformation wird also nur auf das gewählte Objekt angewendet.
Bei Verwendung eines der genannten Behaviors ist der entscheidende Schritt die Zuweisung eines TransformGroup-Objektes, auf das die Transformation angewendet werden soll. Wichtig ist, dass für dieses TransformGroup-Objekt die notwendigen Capabilities gesetzt worden sind.

4.3. Neue Features in Java 3D

Die neue Version Java 3D 1.2 bietet einige neue und interessante Features, deren Erläuterung hier wesentlich dazu beiträgt, zu zeigen, was dieses Renderingsystem leisten kann.

4.3.1. Offscreen Rendering

Man kann Canvas3D beliebig in Screen Designs einfügen, d.h. es ist möglich durch die Software festzustellen welches Pixel welche Farbe angenommen hat. Mit der Methode BufferedImage getImage() kann man an die echten Bilddaten gelangen. Laut der Javadoc-Dokumentation von Canvas 3D wird dieses Feature, wenn immer möglich, genauso wie das Onscreen-Rendering von Hardware oder nativ implementierten Bibliotheken übernommen.
Zu diesem sinnvollen Feature ist zu sagen, daß die Anwendungsgebiete beschränkt sind. Es ist eher unwahrscheinlich, dass ernsthaftes Rendering von Videos etc. in Java realisiert werden wird, und die für Java 3D am ehesten geeigneten kleineren Echtzeit-3D-Systeme von diesem Feature nicht viel Profitieren werden. Da dieses Feature in Java 3D, Version 1.1 fehlte, war es beispielsweise nicht möglich, durch Java 3D erstellte Bilder automatisch in ein Internetforum zu posten.

4.3.2. Model Clipping Planes

Seit Version 1.2 enthält Java 3D die Klasse javax.media.j3d.ModelClip. Diese erlaubt es, durch 6 Ebenen einen Raum zu definieren. Sobald sich der Betrachter in diesem Raum befindet, werden nur noch Objekte dargestellt, die sich auch innerhalb der 6 Ebenen befinden. Sollten sich mehrere dieser Räume überlappen, soll der dem Betrachter räumlich nächste gewählt werden. Der genaue Algorithmus ist hier implementierungsabhängig.
Des Weiteren ist es möglich, einen Scope für das Clipping festzulegen. Entweder ModelClip enthält eine leere Liste von GroupNodes, was zur Folge hat, dass das Clipping auf alle Objekte des Universums angewandt wird, oder im Fall einer nicht leeren Liste sind nur die Elemente der Liste betroffen.
Dieses Feature erlaubt es dem Entwickler, die Performance seiner Anwendung extrem zu steigern, wenn er z.B. bereits beim Design festlegen kann, daß ein bestimmter Bereich einen Raum darstellt und somit Sichtkontakt nach außen eventuell ausgeschlossen werden kann. Dies erspart der 3D-Engine unter Umständen aufwendiges Back-Face-Removal.

4.3.3. Graphics2D-Operationen

Seit der Version 1.2 von Java3D bietet die Klasse javax.media.j3d.Canvas3D eine Methode public J3DGraphics2D getGraphics2D(), welche ein Objekt vom Typ J3DGraphics2D zurück liefert. Dieses erbt über java.awt.Graphics2D, welches seit Java 1.2 Teil der Standard-API von java.awt.Graphics ist. In Version 1.1 von Java 3D war es nur möglich über die von java.awt.Component geerbte Methode public Graphics getGraphics() an eine Referenz auf ein Objekt zu kommen, mit dem man den Canvas3D direkt pixelweise bemalen kann.
In der Version 1.2 der Java Plattform wurde Java2D zwar in die Standardbibliothek integriert, aber java.awt.Component besitzt leider keine Methode, die ein Graphics2D-Objekt zurückliefert. Somit stellt es eine direkte Erweiterung von Canvas3D dar, denn man kann mit dem moderneren Java2D direkt darauf malen. Dies eröffnet die Möglichkeit Technologien wie das Arbeiten mit

Translationen, Skalierungen, Rotationen, Shapes und GlyphVektoren direkt auf der Darstellungsebene.

4.3.4. OrientedShape3D Node

Java 3D 1.1 kennt das Billboard. Dies ist eine von javax.media.j3d.Behavior erbende Klasse, deren Objekte dafür Sorgen, daß eine untergeordnete TransformGroup sich automatisch nach vordefinierte Kriterien ausrichtet. Die Ausrichtung erfolgt relativ zum primären Betrachter. Hier ist es möglich, eine Ausrichtung an einer Achse oder an einem Punkt zu realisieren. Dadurch ist es möglich, optische Täuschungen, wie die Textur einer Baumkrone zu verbessern, indem diese immer rechtwinklig zum Betrachter steht.
Eine andere sehr interessante Anwendung sind Texte, die bestimmten Punkten in der 2D-Welt zugeordnet sind, auf den Betrachter auszurichten. So könnte z.B. ein Text, der ein Objekt beschreibt, automatisch zum Benutzer ausgerichtet werden, so dass der Text aus allen Perspektiven sichtbar ist. Da es sich beim Billboard um ein Behavior handelt, wird die TransformGroup aber real bewegt, was dazu führt, dass die Ausrichtung in einem System mit mehreren Betrachtern immer nur für den primären Betrachter funktioniert. Hier greift das in Java 3D 1.2 neue Objekt OrientedShape3D. Dies bietet dieselbe Funktionalität wie das Billboard, ist aber als Kind von Shape3D realisiert. Da Objekte vom Typ OrientedShape3D im Gegensatz zu Billboard-Objekten auch zu SharedGroup-Objekten hinzugefügt werden können, erlauben sie eine Ausrichtung gleichzeitig auf alle Views im System. Dieses neue Feature ist sehr interessant, wenn man beispielsweise zwei Views für Stereovision benötigt oder Welten mit mehren Benutzer erstellen möchte.

4.3.5. Mehrere Geometries in einer einzelnen Shape3D Node

Objekte vom Typ Shape3D, die ihrerseits zu Transform Groups gehören, enthalten in Java 1.1 ein Objekt vom Typ Geometry. Von Geometry erben alle primitiven Formen wie Linien, Dreiecke und 3D-Text. Durch Shape3D wird dieses Geometry-Objekt mit einem Objekt vom Typ Appearance verbunden. Dieses Appearance-Objekt legt viele für das Rendering wichtige Eigenschaften wie Transparenzen, Farben, Texturen und Materialeigenschaften fest.
In Java 3D ist es damit möglich, einem Objekt vom Typ Shape3D beliebig viele Geometry-Objekte hinzuzufügen, d.h. die Appearance direkt mit all diesen Objekten zu verbinden.
Aus Performancegründen gibt es aber die Einschränkung, dass die zugefügten Objekte vom exakt selben Typ sein müssen, also keine Vererbung möglich ist. Es besteht nur eine Ausnahme, dass alle Subklassen von GeometryArray, die einer der Äquivalenzklassen Punkte, Linien und Polygone angehören, als typgleich behandelt werden.

4.3.6. Picking Utilities

Java 3D bietet die Möglichkeit durch Picking Objekte, die sich in einem bestimmten Bereich befinden, auszuwählen. Hierzu existieren verschiedene Picking Utilitys. Diese Utilities sind alle von PickShape abgeleitet. BranchGroup und Locale stellen folgende Methoden zur Verwendung dieser Utilities bereit:

- public SceneGraphPath[] pickAll(PickShape pickShape)
- public SceneGraphPath[] pickAllSorted(PickShape pickShape)
- public SceneGraphPath pickClosest(PickShape pickShape)

- public SceneGraphPath pickAny(PickShape pickShape)

Der Rückgabetyp SceneGraphPath stellt den Pfad von dem entsprechenden Locale Objekt zum selektierten Objekt dar. Diese Funktionalität ist sehr hilfreich, z.B. wenn die Position der Maus im Koordinatensystem des Canvas3D in 3D-Koordinaten umgewandelt wurde, festzustellen, welche Objekte der Benutzer im Fall eines Klicks selektieren möchte, oder um andere Funktionalitäten an die Mausposition zu koppeln. Es wäre auch möglich, alle in einer bestimmten Perspektive sichtbaren Objekte auf diese Weise zu ermitteln. In Java 1.2 gibt es folgende Picking Utilities:
PickBounds, PickPoint, PickRay und PickSegment. In der Version 1.2 wurden folgende Utilitys hinzugefügt: PickCone, PickConeRay, PickConeSegment, PickCylinder, PickCylinderRay und PickCylinderSegment.

4.3.7. MediaContaintype

Objekte vom Typ javax.media.j3d.MediaContainer repräsentieren Audiodaten, die in der 3D Welt verwendet werden. Diese können von Java3D durch eine URL oder einen Pfad ins Dateisystem referenziert werden. In Java3D ist die Möglichkeit, Audiodaten direkt über einen InputStream zu referenzieren gegeben. Dies ermöglicht es, die Audiodaten aus beliebigen Quellen zu laden wie z.B. ByteArrayInputStreams oder direkt über den seriellen Port mit java.comm. Siehe hierzu auch den nächsten Abschnitt über Sound.

4.3.8. Java 3D Sound

4.3.8.1. Motivation

Um das Erlebnis von virtuellen Welten zu perfektionieren, ist es wichtig, neben der visuellen Modellierung auch die Akustik zu integrieren. Während auf dem Gebiet der 3D-Grafik sehr schnell Fortschritte erzielt wurden, verharrte die Akustik lange auf dem gleichen Stand: Statische Geräusch-Samples wurden plump im Stereofeld platziert und ohne Beachtung von auralen Einflussgrößen abgespielt. Aufwendigere Verfahren blieben den Profis vorbehalten, wie z.B. Geräuschsimulation für Architekten. Mit Java 3D soll sich dies nun ändern. Da Computer in jüngster Zeit solche Berechnungs-Geschwindigkeiten erreichen, die genügen, um neben aufwendiger 3D-Grafik zusätzlich ein komplexes Sound-Modell zu rendern, ist es nun möglich, ein von der Umgebung abhängiges Geräuschmodell aufzubauen, welches völlig in das grafische Modell integriert wird und konsistent zu diesem ist. Um diesen bedeutenden Schritt in der Entwicklung der 3D-Graphik thematisch nicht zu übergehen, und mit dieser Studienarbeit auf diese neuen Möglichkeiten, mit Sound zu arbeiten, aufmerksam zu machen, habe ich dieses Feature bewußt in dieses Kapitel mit aufgenommen.

4.3.8.2. Sound-Knoten

Da der Java3D-Szenegraph das komplette Modell der virtuellen Welt darstellt, umfasst dieser natürlich auch die akustische Modellierung. So existieren drei verschiedene Arten von Sound-Knoten, welche je nach gewünschtem Effekt analog zu den grafischen Knotenobjekten in den Szenegraph eingehängt werden.
Jeder Sound-Knoten hat eine Referenz auf die Sample-Daten, einen Koeffizienten für die Grundlautstärke, eine Abspielpriorität, ein Flag, welches angibt, ob der Sound in jedem Fall bis

zum Ende abgespielt werden soll, eine Wiederholungsanzahl, einen Aktivierungs-Zustand sowie eine Aktivierungs-Region. Sobald sich der Zuhörer innerhalb der Aktivierungsregion befindet, ist der Sound potentiell hörbar.
Vorraussetzung hierfür ist natürlich, daß dem Sound vorher Sampledaten zugeordnet wurden, der Sound-Knoten in den Szenegraphen eingehängt wurde, ein aktiver View im virtuellen Universum existiert, sowie das Audioausgabegerät dem PhysicalEnvironment-Objekt bekannt gemacht wurde.
Die Sample-Daten werden über ein Java3D-MediaContainer-Objekt mit dem Sound-Knoten assoziiert und können gepuffert oder als Streaming-Audio abgespielt werden, siehe vorheriger Abschnitt.
Wenn die Anzahl der abzuspielenden Kanäle die Möglichkeiten des Audioausgabegerätes übersteigt, entscheidet die individuelle Priorität jedes Sounds, wie wichtig er für das Gesamtklangbild ist und ob er möglicherweise nicht oder nur teilweise abgespielt wird. Bei gleicher Priorität von Sounds entscheidet der Zufall.

4.3.8.3. BackgroundSound-Knoten

Der BackgroundSound-Knoten ist eine Spezialisierung des abstrakten Sound-Knotens, bei dem der Sound weder eine Position noch eine Richtung aufweißt und völlig unbearbeitet dem Gesamtmix zugeführt wird. Dies ist nützlich, um einer Szene Hintergrundmusik in Form eines Mono- oder Stereomusikstückes oder auch einen ambienten Sound-Effekt wie z.B. Vogelgezwitscher hinzuzufügen. Im Gegensatz zum grafischen Pendant können beliebig viele BackgroundSound-Knoten simultan aktiv sein.

4.3.8.4. PointSound-Knoten

Die zweite Form eines Sound-Knoten stellt der PointSound-Knoten dar, bei dem der Schall radial von einer bestimmten Position gleichförmig in alle Richtungen abgestrahlt wird.
Zusätzlich zu den Attributen eines normalen Sound-Knoten besitzt der PointSound-Knoten die Möglichkeit einer abstandsabhängigen Lautstärkeabschwächung, die individuell definiert werden kann. Dazu muss der Modellierer bestimmte Entfernungs-Lautstärke-Paare angeben, zwischen denen dann zur Laufzeit anhand der berechneten Entfernung der ViewPlatform zur Schallquelle die resultierende Lautstärke linear interpoliert wird.

Die Entfernungsangaben gelten im lokalen Koordinatensystem der Schallquelle. Wenn zur Laufzeit der Entfernungswert kleiner ist als die kleinste spezifizierte Distanz, wird der erste Lautstärke-Wert im Array angewendet.
Bei zu großer Entfernung des Hörers wird der letzte Wert benutzt.

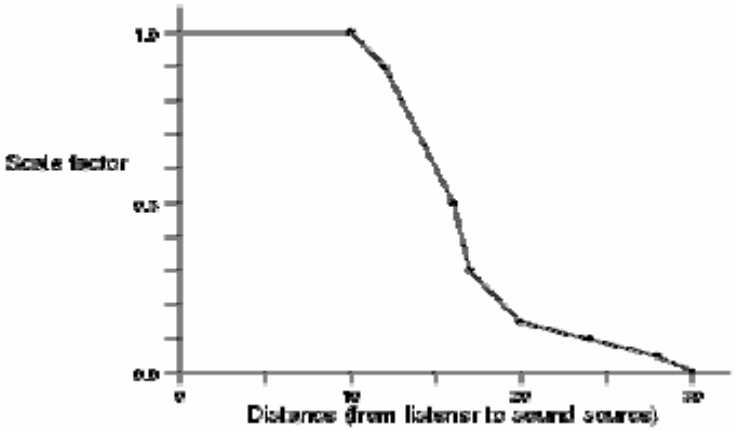


Abbildung 12: Distanzarray

4.3.8.5. ConeSound-Knoten

Die dritte und spezialisierteste Form eines Sound-Knoten ist der ConeSound-Knoten. Dieser hat die zusätzliche Eigenschaft, dass er seinen Schall kegelförmig entlang eines bestimmten Vektors abgibt. Je weiter die Hörerposition von dieser Achse abweicht, desto leiser wird der empfangene Schall. Zusätzlich zur Lautstärkeabschwächung ist es möglich, per LowPass-Filter den Sound dumpfer klingen zu lassen.
Möchte man ein ellipsoides Abschwächungsgebiet für seine Schallquelle erreichen, so muß man zwei DistanceGainAttenuation-Arrays füllen, jeweils eines für die Front-und Backdistance. Insgesamt kann man für das ConeSound also 6 Arrays bestücken:

1. frontDistance[] Entfernungswerte in m
2. frontDistanceGain[] Amplitudenkoeffizienten für pos. Abstände entlang der Achse
3. backDistanceGain[] Amplitudenkoeefizienten für neg. Abstände entlang der Achse
4. angle[] Winkelwerte (Abweichung vom Achsenvector im Bogenmaß)
5. angularGain[] Amplitudenkoeffizienten für Winkel in angle[]
6. frequencyCutoff[] Filterfrequenz in Hz

4.3.8.6. Soundscape-Knoten

Der Soundscape-Knoten definiert akustische Attribute (“Aural Attributes”) für bestimmte Regionen innerhalb einer 3D-Szene. So ist es z.B. möglich, beim Wechsel von einem Raum zum anderen, innerhalb eines Gebäudes die spezifische Raumakustik für jeden Raum individuell zu simulieren und so dem visuellen Eindruck anzupassen.

4.3.8.7. AuralAttributes-Objekt

Das AuralAttributes-Objekt als Komponente des Soundscape-Knotens definiert die Audio-Parameter einer bestimmten Umgebung. Dazu gehören ein Skalierungsfaktor für die Gesamtlautstärke aller Sounds, die Abweichung von der normalen Luftgeschwindigkeit, verschiedene Filter- und Hallparameter sowie Einstellungen für den Dopplereffekt.

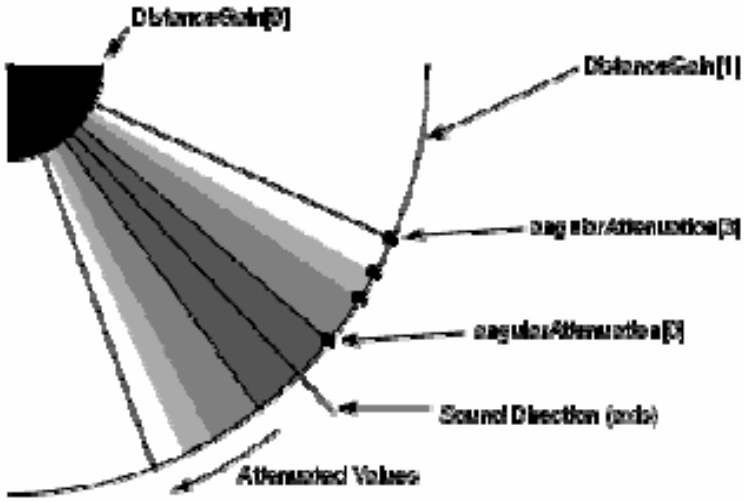


Abbildung 13:
Kegelförmige Sound-Abschwächung

Luftgeschwindigkeit: Um eine Abweichung der Schallgeschwindigkeit von der Luftgeschwindigkeit bei Zimmertemperatur (344 m/s) zu bewirken, ist es möglich einen “rolloff” zu definieren, welcher als Koeffizient in die Berechnung mit eingeht. Werte >1.0 erhöhen und Werte <1.0 verringern die Geschwindigkeit des Schalls. Ein rolloff von 4,31 würde z.B. die Schallgeschwindigkeit im Wasser bei 20° C simulieren.

Nachhall: Da Hallberechnungen in Echtzeit sehr rechenintensiv sind, beschränkt sich die Java3D-Hallkomponente auf ein relativ einfaches Modell mit drei Parametern. Die **Delay time** (Predelay) gibt an, wie lange das von der Schallquelle ausgesandte Signal benötigt, bis es nach Reflektion an einer Oberfläche den Zuhörer erreicht. Dies ist auch als Backwall-Effekt bekannt und macht eine akustisch wahrnehmbare, räumliche (Tiefen-)Positionierung der Schallquelle im Raum über die zeitliche Verschiebung des Nachhalls gegenüber dem Direktschall möglich.
Der **Reflexionskoeffizient** bestimmt die Menge des von einer Oberfläche zurückgeworfenen Schalls. Metallische Objekte haben einen hohen Reflexionskoeffizient, währenddessen z.B. Stoffe eine hohe Absorption und somit einen eher kleinen Reflexionskoeffizienten aufweisen.
Der Parameter **Feedback loop** gibt an, wie viele Reflexions-Iterationen berechnet werden sollen. Ein Wert von –1 bedeutet, dass so viele Reflexionen berechnet werden, bis die Amplitude den Wert von –60dB bzw. 1/1000 der Originalamplitude unterschreitet. Währenddessen erzeugt ein Wert von 1 z.B. nur ein einzelnes Echo.

Doppler-Effekt: Der Doppler-Effekt ist jedem durch das Sirenengeräusch von vorbeifahrenden Polizeiwagen bekannt und kann genutzt werden, um eine bessere Wahrnehmung von Schallquellenbewegungen zu gewährleisten. Dabei wird abhängig von der relativen Geschwindigkeit zwischen Hörer und Schallquelle die Tonhöhe bzw. Frequenz des empfangenen akustischen Ereignisses erhöht oder gesenkt. Die Ursache hierfür liegt in der Stauchung bzw. Streckung der Wellenberge des Schalls.
Der “velocityScaleFactor” bestimmt den Einfluss der kalkulierten Differenzgeschwindigkeit auf die Tonhöhenänderung. Der “frequencyScaleFactor” kann getrennt vom Doppler-Effekt dazu genutzt werden, die Tonhöhe des Originalsignals zu verändern.

DistanceFilter: Analog zur Filterung beim ConeSound-Objekt ist es mit dem DistanceFilter möglich, abhängig von der Entfernung des Hörers zu einer beliebigen Schallquelle in der Region eine Low-Pass-Filterung vorzunehmen. Hiermit könnte man z.B. das Grollen eines entfernten Gewitters bzw. das krachende höhenbetonte Donnern eines nahen Gewitters simulieren.

4.3.8.8. AudioDevice Interface

Das AudioDevice Objekt spezifiziert eine abstrakte Schnittstelle, die individuell für jedes Audio-Ausgabegerät implementiert sein muss.
Die Anwendung muss prüfen, wie viele Ausgabegeräte zur Verfügung stehen und welche Ressourcen diese bereitstellen. Die Auswahl geschieht durch Aufruf der Methode “setAudioDevice” des PhysicalEnvironment-Objektes.
Das Initialisieren und Freigeben der jeweiligen Audioressource geschieht über die Methoden “initialize” und “close” des AudioDevice-Objektes.
Des Weiteren enthält dieses Objekt Informationen über die Anzahl der Kanäle, den Abspielmodus (Mono, Stereo oder Kopfhörer) und den Winkel zwischen Mittenvektor und Lautsprechervektor im Stereofeld.

4.3.8.9. AudioDevice3D Interface

Die Schnittstellenklasse AudioDevice3D ist abgeleitet von AudioDevice und bildet die Grundlage einer individuellen hardwaremäßigen Implementierung der Java3D-Sound-Features für Entwickler von Audioausgabegeräten.

Am Ende des Kapitels 4 gibt es zur Information ein Java3D-Sound-Beispiel.

4.4. Aufbau eines Programmes

4.4.1. Java 3D Klassenhierarchie

Der generelle Aufbau:

javax.media.j3d

- VirtualUniverse
- Locale
- View
- PhysicalBody
- PhysicalEnvironment
- Screen3D
- Canvas3D (extends java.awt.Canvas)
- SceneGraphObject
  - Node
    - Group
    - Leaf
  - NodeComponent
- Transform3D

Die Java 3D-API-Klassenhierarchie besteht aus zwei Typen von Klassen:

- den Kernklassen ("core classes")
- den Zubehörklassen ("utility classes")

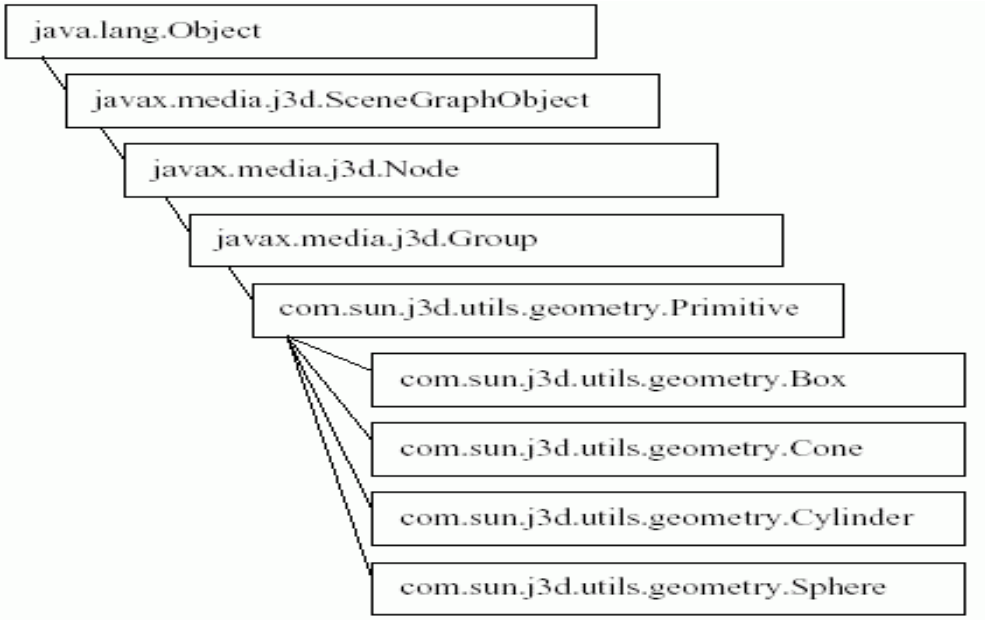
Jede Java 3D-Applikation baut - zumindest zu einem Teil - auf den Kernklassen auf. Die Ansammlung dieser Klassen beschreibt das virtuelle Universum bzw. die 3D Szene, die dargestellt wird. Die Kernklassen finden sich in dem javax.media.j3d.* Package. Zu beachten ist hierbei, daß obwohl es weit mehr als 100 Java 3D Kernklassen gibt, es dabei trotzdem möglich ist, eine einfache funktionsfähige 3D-Applikation mit schon sehr wenigen Klassen zu implementieren. Neben dem oben erwähnten Kernklassen-Package, können zusätzlich dazu auch noch andere Packages mit sehr nützlichen Zubehörklassen zur Programmierung von Java 3D Applikationen verwendet werden. Dies ist z.B. das com.sun.j3d.utils.* Package. Die darin enthaltenen "utility classes" bilden eine Erweiterung der elementaren Kernklassen und ermöglichen so, schneller und einfacher 3D-Programme zu entwickeln.
Die Zubehörklassen sind in die folgenden Kategorien unterteilt:

- Ladefunktionen für Dateien mit 3D-Informationen ("content loader")
- Vereinfachungen zur Konstruktion des Szenengraphen ("scene graph construction aids")
- Geometrieklassen ("geometry classes")
- Allgemeine vereinfachende Klassen ("convenience classes")

Zu erwähnen ist hierbei, dass in zukünftigen Versionen von Java 3D neue Zubehörklassen hinzukommen werden aber auch einige aktuelle Zubehörklassen in das Kernpackage übernommen werden könnten.
Weitere Java (Zubehör-) Klassen, die in den meisten Java 3D Applikationen ebenfalls verwendet werden, sind:

- AbstractWindowToolkit(AWT)-Klassen (java.awt.*) zur Erzeugung eines 3D-Ausgabefensters
- Vektormathematik-Klassen (javax.vecmath.*) für mathematische Objekte wie z.B. Punkte, Vektoren, Matrizen usw.

Abbildung 6: Aufbau der Klassenhierarchie



4.4.2. Szenengraphen erstellen

Knoten definieren: Instanziieren von Java 3D Klassen
Shape3D myShape1 = new Shape3D(myGeom1, myAppear1);
Shape3D myShape2 = new Shape3D(myGeom2);

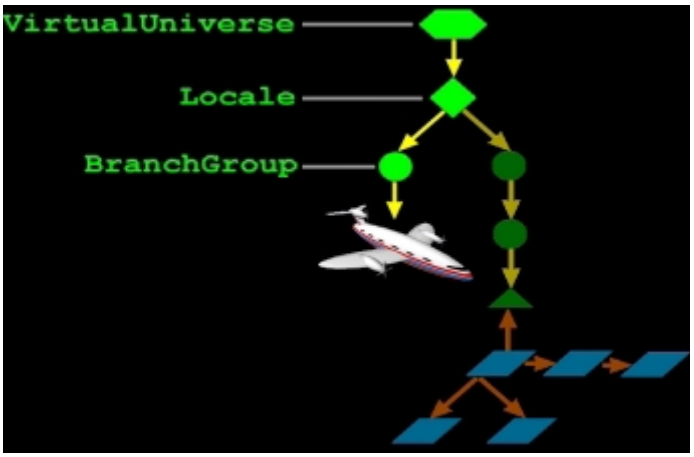
Ändern von Knoten: Methodenaufruf
myShape2.setAppearance(newAppear2);

Gruppieren der Knoten
Group myGroup = new Group( );
myGroup.addChild(myShape1);
myGroup.addChild(myShape2);

Man definiert die Komponenten zunächst separat. Dann werden die Komponenten zu einem gemeinsamen Container zusammengefasst (Virtual universe).

Dies ist ein Weg, um Szenengraphen zusammenzufassen und eine Möglichkeit für die Bestimmung des Root-Knoten des Szenengraphen.

Abbildung 7: Gruppenknoten des Flugzeugbeispiels



Virtual universe: ist die Wurzel eines Szenengraphen.
Ein Szenengraph besitzt nur eine Wurzel, typischer Weise: ein Universe pro Applikation.

Locale: ist eine Art Startpunkt, eigentlich der Ursprung des Koordinatensystems, zu dem die visuellen Objekte des Universe relativ positiv sind. Es gibt eine Locale pro Universe.

Branch graph: ist die Wurzel eines Subgraphen des Locale Baumes. Typisch sind mehrere Branchgraphs pro Locale.

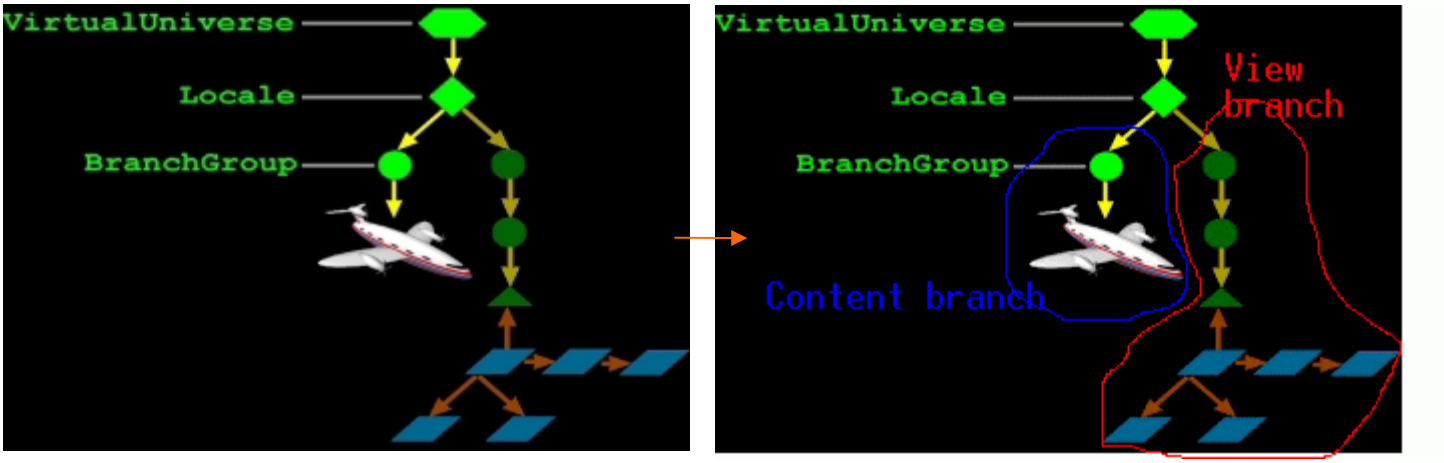
Abbildung 8: Knoten im Programmcode

4.4.3. Unterteilung von Szenengraphen

Content branch: beschreibt die 3D-Welt, d.h. die Geometrie, Aussehen, Positionierung, Verhalten und Beleuchtung der Objekte.

View branch: spezifiziert, wie die Welt angezeigt wird, z.B. Positionierung der Kamera. Es Existiert ein View branch pro universe.

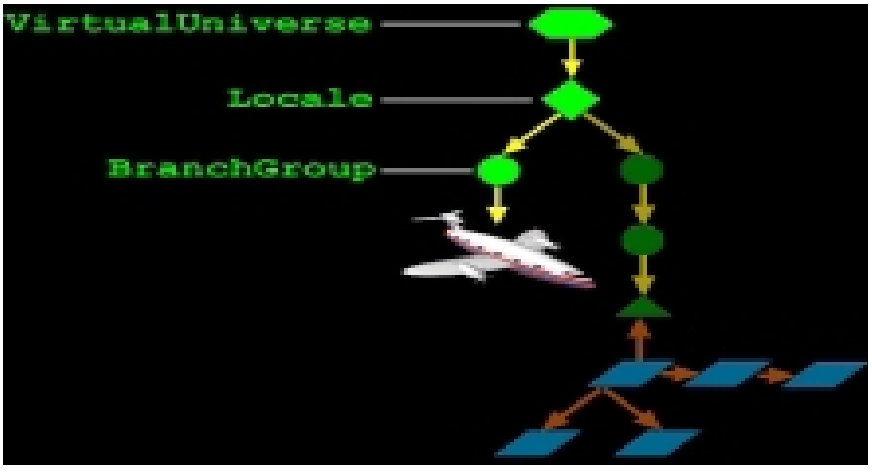
Abbildung 9: Unterteilung von Szenengraphen am Flugzeug-Beispiel



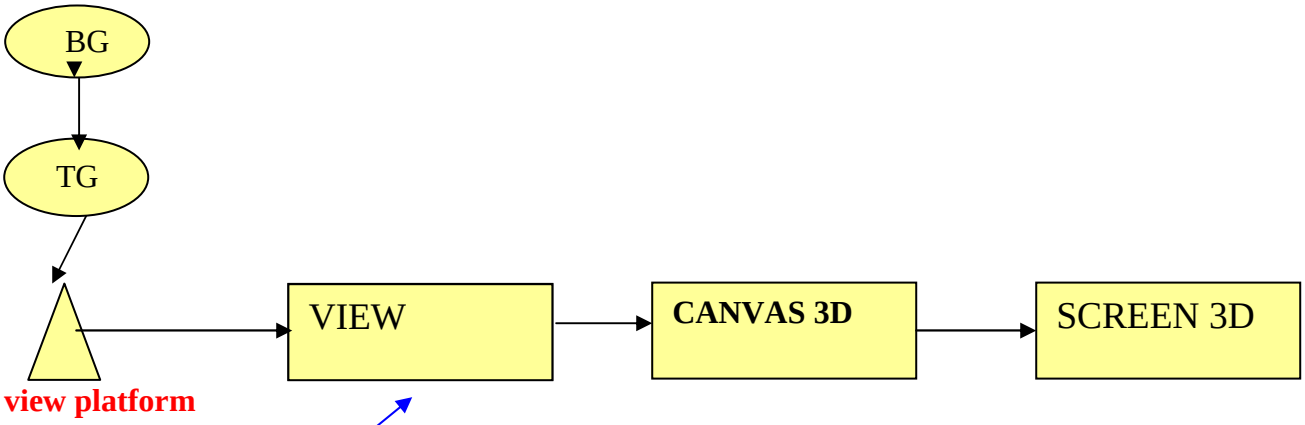
Optimale Unterteilung: Inhalt- und View- Informationen werden getrennt

Erstellen des Wurzelknotens (universe)
Erstellen von universe:
VirtualUniverse myUniverse = new VirtualUniverse( );
Erstellen von *locale*:
Locale myLocale = new Locale(myUniverse);
Erstellen von branch group:
BranchGroup myBranch = new BranchGroup( );
Erstellen von nodes und groups of nodes
Shape3D myShape = new Shape3D(myGeom, myAppear);
Group myGroup = new Group( );
myGroup.addChild(myShape);
Hinzufügen zur branch group
myBranch.addChild(myGroup);
Hinzufügen des branch graph zum locale
myLocale.addBranchGraph(myBranch);

Abbildung 10: Branchgroup-Knoten



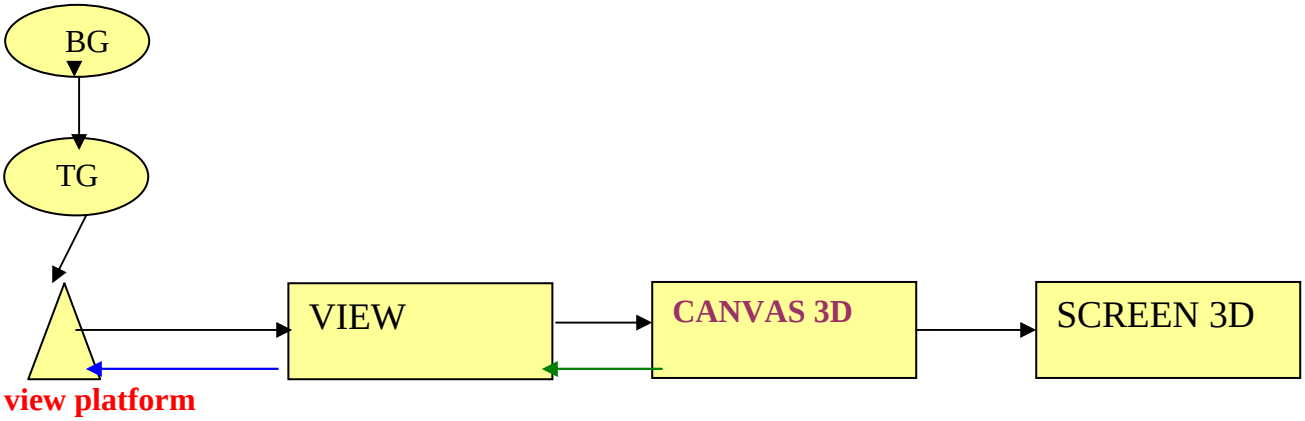
Im Weiteren werden die Knoten anhand von zwei Skizzen näher erläutert, sowohl in ihrer Anordnung in der Baumstruktur, als auch in ihrer Funktion und Bedeutung.



ViewPlatform-Knoten: Spezifiziert den Blickwinkel auf die Szene, pro Plattform ein View-Objekt, mehrere View-Plattformen sind möglich, mit dem TransformGroup-Knoten über View platform kann die Position Orientierung und Skalierung des Blickwinkels auf die Szene verändert werden

View-Objekt: koordiniert alle Aspekte des Renderingprozesses, enthält alle Informationen, die für die Darstellung der Szenen in das Windows der entsprechenden Canvas3D-Objekte nötig sind. Mehrere aktive View-Objekte sind möglich.

PhysicalBody-Objekt: **s**pezifiziert Kalibrierungsinfos über Körper des Benutzers, Positionierung der Augen, Pupillenabstand, etc., Umgebungen mit Head Tracking (z.B. HMD), Normalerweise reicht eine Einstellung (Monitor).



PhysicalEnvironment-Objekt: **s**pezifiziert Kalibrierungsinformationen für die Ausgabegeräte, z.B. Optik des HMDs. Für die normale Umgebung reicht die default-Einstellung.

Canvas3D-Objekt: Oberfläche, auf die die View ein Bild der 3D Szene projiziert, pro View ist nur ein Canvas3D Objekt erlaubt. Eine Anwendung kann natürlich mehrere Canvas3D-Objekte enthalten. Canvas3D-Objekte enthalten die Referenz auf Screen3D, das die physikalischen Eigenschaften des Anzeigefensters enthält.

Content-Zweig: beschreibt die virtuelle Welt, CB wird „lebendig“, sobald er unter einen Locale-Knoten gehängt wird. „Lebendige“ Objekte werden gerendert, d.h. im Anzeigefenster dargestellt. Die Parameter von „lebendigen“ Objekten können während der Laufzeit des Programms nur geändert werden, wenn die entsprechenden Capabilities gesetzt wurden.

Kompilierte BranchGroup-Knoten lassen sich effizienter rendern. Daher ist es ratsam: bevor man BG unter den Locale Knoten hängt, kompilieren.

4.4.4. Schreiben eines Programmes

Erzeugen einer Unterklasse der Klasse Applet

public class HelloJava3D extends Applet

Erzeugen eines Konstruktors

public HelloJava3D

Bereitstellen einer Methode für das Erzeugen des Content- Zweiges

public BranchGroup createSceneGraph()

Aufruf des Applets in einem MainFrame-Objekt in der main-Methode des Applets:

public static void main (String[] args) {
Frame frame = new MainFrame (newHelloJava3D(), 256, 256);
}

Erzeuge Canvas3D Objekt

Canvas3D canvas3D = new Canvas3D(null)

Erzeuge eine SimpleUniverse Objekt

SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
simpleU.
getViewingPlatform().setNominalViewingTransform();

Erzeuge Content-Zweig

BranchGroup scene = createSceneGraph()

Kompiliere Content-Zweig

scene.compile();

Content-Zweig in simpleUniverse hängen

simpleU.addBranchGroup(scene);

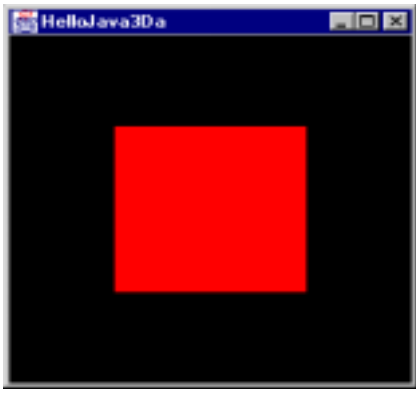


Abbildung 11:

Erstes Implementierungsbeispiel

„Hello Java3D“

4.4.5. Beispielprogramm HelloWorld

Einbinden der Klassen aus der Java-Bibliothek

```
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.Text2D;
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.GraphicsConfiguration;
```

HelloWorld zeigt den Schriftzug 'Hello World!' im 3D Universum

```
public class HelloWorld extends Applet {
  public HelloWorld() {
    setLayout(new BorderLayout());
    GraphicsConfiguration graphConfig =
      SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas3D = new Canvas3D(graphConfig);
    add("Center", canvas3D);
```

SimpleUniverse vereinfacht das Erstellen des Universums

```
    SimpleUniverse simpleUni = new SimpleUniverse(canvas3D);
    BranchGroup sceneGraph = createSceneGraph();
```

ViewPlatform wird so eingestellt, daß man Objekt betrachten kann

```
    simpleUni.getViewingPlatform().setNominalViewingTransform();
    simpleUni.addBranchGraph(sceneGraph);
    }
```

Ende Konstruktor HelloWorld

Erzeugen des Content-Branch des Szenengraphen

```
  public BranchGroup createSceneGraph() {
    BranchGroup objRoot = new BranchGroup();
```

Anhängen des Demo-3D Objekt: den Schriftzug 'Hello World!' (Farbe weiss, Schrift Helvetica, Grösse 18, Normalschrift) an den Ast

```
    objRoot.addChild(
      new Text2D("Hello World !", new Color3f(1f, 1f, 1f), "Helvetica", 18, 0));
```

Optimierungen am Ast des Szenengraphen

```
    objRoot.compile();
    return objRoot;
  }
```

Ende Methode createSceneGraph

Programm sowohl als Applet wie auch als Applikation laufen lassen

```
    public static void main(String[] args) {
```

Erzeugen des Applikationsfenster der Groesse 800x600

```
        MainFrame mainFrame = new MainFrame(new HelloWorld(), 800, 600);
    } Ende Methode main
} Ende Klasse HelloWorld
```

Verwendete Klassen:
BranchGroup (Konstruktor, compile, addChild)
Canvas3D (Konstruktor)
SimpleUniverse (Konstrutor, setNominalViewingTransform)
ColorCube (Kantenlänge von 2 Metern im Nullpunkt

Als weiteres Beispiel für Java 3D sei an dieser Stelle für Interessierte und der Vollständigkeit halber das am Ende von Abschnitt 4.3.8. erwähnte Java3D-Sound-Beispiel gegeben:

4.4.6. Sound-Beispiel

Das folgende kompakte Code-Beispiel “SoundExample” spielt einen Sound “Hello Universe” unendlich oft ab.

```
import java.applet.Applet;
import java.net.URL;
import java.awt.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import java.io.File;
import javax.media.j3d.*;
import javax.vecmath.*;

public class SoundExample extends Applet {
        private static URL url;
        public BranchGroup createSceneGraph() {
                try { url = new URL("file:hello_universe.au"); } catch (Exception e) {}
// Wurzel des Subgraphen erzeugen
        BranchGroup objRoot = new BranchGroup();
// PunktSchallquelle mit Kugelbegrenzung erzeugen
        PointSound sound1 = new PointSound();
        BoundingSphere soundBounds =
        new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        sound1.setSchedulingBounds(soundBounds);
        objRoot.addChild(sound1);
// Sample laden und abspielen
        MediaContainer sample1 = new MediaContainer();
        sample1.setCapability(MediaContainer.ALLOW_URL_WRITE);
        sample1.setCapability(MediaContainer.ALLOW_URL_READ);
        sample1.setURL(url);
```

```
sample1.setCacheEnable(true);
sound1.setLoop(-1);
sound1.setSoundData(sample1);
sound1.setEnable(true);
objRoot.compile();
return objRoot;
}
public SoundExample() {
setLayout(new BorderLayout());
GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
Canvas3D c = new Canvas3D(config);
add("Center", c);
SimpleUniverse u = new SimpleUniverse(c);
AudioDevice audioDev = u.getViewer().createAudioDevice();
BranchGroup scene = createSceneGraph();
u.addBranchGraph(scene);
}
public static void main(String[] args) {
new MainFrame(new SoundExample(), args, 64, 64);
}
}
```

Anmerkung:

Eine Zusammenfassung zum Kapitel über Java 3D erfolgt im Kapitel 5 in Form des Vergleichs der Renderingsysteme.

5. Vergleich von Java 3D mit Open Inventor

5.1. Allgemeine Einschätzung

Nach der Einarbeitung in beide Graphiktools, dem Studium der Tutorials, vieler Internet- und Buch-Veröffentlichungen, sowie ersten Programmierversuchen fällt ein Vergleich nicht leicht, denn Open Inventor und Java 3D entsprechen sich in wesentlichen Punkten. Die - meiner Meinung nach - wichtigsten zehn sind hier aufgeführt, nach der folgenden Auflistung, deren Reihenfolge nicht als Bewertungsmodus gilt:

- Plattformunabhängigkeit und Betriebssystem
- Erweiterbarkeit
- Dateiformate
- Konzept und Aufbau
- Programmierung
- Optimierungen
- Animation und Benutzerinteraktion
- Features
- Kollisionserkennung

Open Inventor ist wesentlich älter, Java 3D ist erst in den letzten Jahren entwickelt worden.
Beide Renderingsysteme sind sowohl von der Struktur her gleich, als auch in der Anwendung ohne wesentliche Unterschiede zu bewerten. Was die beiden Graphiktools wirklich können, stellt sich nach der Einarbeitung und ersten Programmierversuchen heraus. Ich habe den Schluß gezogen, dass Open Inventor und Java 3D auch hier in sehr vielen Punkten überein stimmen, jedoch die Unterschiede im Detail zu finden sind. Ein wesentlicher Unterschied in der Benutzung des Graphiktools ist die anzuwendende Programmiersprache.

Dieser Vergleich soll sich jedoch aufgrund dieser zusammenfassenden Einschätzung nicht mehr mit den Details beschäftigen, da diese ausführlich in den jeweiligen Abschnitten über die beiden Renderingsysteme behandelt worden sind und tatsächlich sehr schwer direkt vergleichbar sind. Diese Feststellung ist auch als eines der Resultate dieser Studienarbeit zu werten.

5.2. Plattformunabhängigkeit und Betriebssystem

Open Inventor und Java 3D sind auf verschiedensten Plattformen verfügbar, wie unten näher erläutert, und sind somit plattformunabhängig.
Open Inventor ist für Hewlett Packard, SUN (Linux) und MS-Windows verfügbar; eine ebenfalls existierende Java-Implementation ist weitgehend plattformunabhängig. Open Inventor bietet also systemunabhängigen Zugriff auf die Benutzeroberfläche.
Java 3D ist geeignet für Integration von plattformunabhängiger 3D-Grafik in Java-Anwendungen; wobei gilt: "Write Once, Run Anywhere™"
Java 3D gibt es für IRIX, HP-UX, Solaris, Linux und Windows (NT, Windows 98, Windows 2000). Es gibt eine Version, die auf DirectX basiert und eine, die auf OpenGL basiert. Java3D-Programme können als alleinstehende Applikationen und als Java Applets implementiert werden. Um Java3D-Programme ausführen zu können, muss allerdings die Java3D-Erweiterung in der entsprechenden Virtuellen Maschine (VM) installiert sein.

Während der Studienarbeit wurden beide Systeme unter Windows genutzt:

Java 3D in „Eclipse“ und Open Inventor (Coin) in „Microsoft Visual Studio 7.0“, wobei es bei letzterem immer wieder Probleme mit dem Includieren diverser Bibliotheken gab und somit die Anwendungen dann nur mit viel Mühe zum Laufen gebracht werden konnten.
Der Erfahrungsaustausch mit Studenten unserer Studienrichtung, die in dieser Zeit parallel ein Projektpraktikum auch unter Verwendung von Open Inventor absolvierten, ergab offensichtlich, daß Open Inventor unter Linux mühelos zu installieren ist und die Programme problemlos laufen. Beide Systeme sind als Open Source downzuloaden, also kostenlos zu beziehen, URL's siehe Quellenverzeichnis.

Die Erfahrung mit der Installation und Programmierung während der Studienarbeit lässt den persönlichen Schluß zu, dass zum Einen die als erstes erlernte und gewohnte Programmiersprache – in diesem Falle Java – dem entsprechenden System auch den Vorzug gibt, obwohl Open Inventor das offensichtlich professionellere System ist, und zum Anderen, daß die Anwendersoftware und das Betriebssystem, in der man die Renderingsysteme zum verwendet, ebenfalls für eine optimale Nutzung entscheidend sein können.

5.3. Erweiterbarkeit

Beide Systeme können durch den Hintergrund der Klassenbibliothek jederzeit erweitert werden und sind mit anderen Tools zu kombinieren:
Open Inventor kann relativ einfach mit OpenGL kombiniert eingesetzt werden, so dass die Vorteile beider Bibliotheken innerhalb einer Anwendung ausgenutzt werden können.
Java 3D ist mit Swing und Java-AWT leicht kombinierbar, was empfehlenswert ist, um eine noch höhere Performance zu erreichen.

5.4. Speicherverwaltung

Obwohl in C++ geschrieben worden, ist in Open Inventor eine komfortable Speicherverwaltung integriert, die ähnlich wie bei Java, ein explizites Freigeben von Objekten unnötig macht. Der für die Speicherverwaltung verwendete Mechanismus beruht wie bei Java auf einer Verwaltung von Referenzzählern für Objekte.

5.5. Dateiformate

In Java 3D werden Formate verschiedener Hersteller mit Hilfe von „formatloader“ unterstützt: 3DS (3D-Studio), PDB (Protein Data Bank), DEM (Digital Elevation Map), IOB (Imagine), COB (Caligari trueSpace), OBJ (Wavefront), DXF (Drawing Interchange File), AutoCAD, VRML 97, PLAY, X3D, OpenFLT, LWS (Lightwave Scene Format), LWO (Lightwave Object Format), OBJ (Wavefront), NFF (WorldToolKit), VTK (Visual Toolkit), VRT.
In Open Inventor dagegen gibt es ein binäres Dateiformat und ein lesbares ASCII-Dateiformat. Es ist anzumerken, daß das Dateiformat nicht mehr ganz dem aktuellen Stand entspricht, XML wäre hier angebrachter, Ausführungen dazu im Abschnitt 3.1.4.
Dies ist meiner Meinung nach für die Invertierbarkeit mit anderen Programmen ein Nachteil.

5.6. Konzept und Aufbau

Als universeller Typ von Szenenrepräsentation hat sich in heutigen computergraphischen Systemen der Szenengraph durchgesetzt. Den Szenengraphen haben beide Systeme als zentrales Konzept. Der Szenengraph als ein gerichteter azyklischer Graph (DAG) mit nur einem Wurzelknoten, der alle Objekte, die die Szene in irgendeiner Weise beeinflussen, enthält.

Der Szenengraph erlaubt es, kompakt Szenen zu modellieren, Gemeinsamkeiten in der Attributierung und in den geometrischen Transformationen auszudrücken und kann direkt durch eine Benutzerschnittstelle editiert werden. Hier gibt es allerdings Unterschiede in Aufbau und Terminologie der beiden Renderingsysteme.
In Open Inventor wird die 3D-Szene mit all ihren Objekten in einer Baumstruktur mit allen Informationen über die Szene gespeichert, wobei mit Objekten hier die 3D-Geometrien als Oberflächenmodelle mit zusätzlichen Beleuchtungsinformationen gemeint sind. Der Szenengraph bildet zusammen mit einigen Zusatzinformationen die Szenendatenbank (Scene Database).
In Java 3D dagegen ist der Szenengraph etwas anders aufgebaut als in Open Inventor und die Terminologien sind teilweise anders, natürlich an die Sprache Java angelehnt.
Für jede Szene existiert genau ein Szenengraph, der die Daten enthält: Kinder sind z.B. Modelle, Lichter, Sounds. Väter sind Gruppen, die Kinder evtl. weitere Väter.
Die Knoten dieser Baumstruktur beinhalten weitere Java3D-Objekte, die z.B. geometrische Formen oder bestimmte Ausgabe-Einstellungen dieses virtuellen Universums beschreiben.

5.7. Programmierung

Grundstein der Programmierung für beide Renderingsysteme ist die jeweils umfangreiche Klassenbibliothek und das Prinzip der Objektorientiertheit, teils – wie beschrieben auch in Open Inventor mit dem Vererbungsprinzip versehen.
Der wesentliche Unterschied der Systeme ist, wie bereits erwähnt, die Programmiersprache.
Ein großer Vorteil von Open Inventor ist, daß es als objektorientiertes Toolkit in C++ formuliert ist und somit eine state-of-the-art–Implementierung ermöglicht. Die Sourcen sind jedoch, genau wie in Java 3D, nicht zugänglich.
Ein Programmierer kann die zur Verfügung gestellten Klassen und Methoden benutzen und so das Grafiksystem flexibel in seine Applikationen einbinden. So können komplexe Grafik-anwendungen leicht und übersichtlich implementiert werden.
Java 3D ist, wie der Name schon sagt, ebenfalls eine 3D-Grafikbibliothek, die vollständig in Java geschrieben ist und bietet daher eine Java-API. Die Klassenbibliothek ist sehr umfangreich und bietet einige besondere Features, auf die in dieser Studienarbeit ausführlich eingegangen wird.
Die Akzeptanz in Programmierkreisen wird sich wohl auch aufgrund der gewohnten Sprache wiederfinden.
Die API von Open Inventor wird als high level bezeichnet, die von Java 3D als low level.

Java 3D und Open Inventor kann man jeweils als eine Sammlung von Bausteinen, aus denen man ganze Applikationen zusammensetzen kann, verstehen. Diese Bausteine bedienen sich schneller Hardwareunterstützung und sind sehr systemnah programmiert. Obwohl diese Bausteine sehr umfangreich sein können, bringt ihre Verwendung nur minimalen Programmieraufwand mit sich.

5.8. Optimierungen

Open Inventor macht intern sehr viele Optimierungen implizit und unterstützt einige weitere Optimierungen; man muß sich also praktisch kaum darum kümmern.
In Java 3D gibt es auch eine Automatisierung der Rendering-Optimierung. Der Java3D-Renderer verwendet beim Durchlaufen der Blattknoten automatisch die optimale Reihenfolge, um die effizienteste 3D-Ausgabe zu erreichen. Dieser Vorgang wird ebenfalls intern und transparent durchgeführt. Unterschied zu Open Inventor ist hier jedoch, dass mit drei verschiedenen Rendering-Modi gearbeitet wird: Immediate-Mode, Retained-Mode und

Compiled-Retained-Mode. Jeder Modus ermöglicht, mit einem unterschiedlichen Grad an Automatisierung zu arbeiten und somit unterschiedliche Optimierungen der 3D-Ausgabe durch das Java3D-System durchzuführen. Im Kapitel über Java 3D wird daher eine kurze Beschreibung der einzelnen Modi gegeben. Für Optimierungen wird vorwiegend der Compiled-Retained-Mode genutzt, da er von der softwaretechnologischen Seite am interessantesten ist.

5.9. Animation und Benutzerinteraktion

Open Inventor basiert auf OpenGL, das wiederum die speziellen Hardwareeigenschaften der von Silicon Graphics hergestellten Systeme optimal unterstützt. Auf diese Weise lassen sich auch mit Open Inventor Echtzeit-Animationen programmieren. Zusätzliche Komponenten für grafische Oberflächen erweitern die Fähigkeiten von Open Inventor, denn es steht nicht nur das gerenderte Bild im Vordergrund, vielmehr soll auch das interaktive Arbeiten mit dem visualisierten Modell unterstützt werden. Szenen werden zunächst als Szenengraph konstruiert, und erst dann gerendert oder anderweitig weiterverarbeitet. Der Szenengraph kann zu jedem Zeitpunkt, d.h. auch nach dem Rendern, geändert werden. Dies ist auch die Grundlage für die einfache Implementierung von Interaktionen und Animationen in Open Iventor. Alle Knoten exportieren ausgewählte Objektattribute in Form von sog. Fields. Eine Anwendung manipuliert einen OpenInventor-Knoten mit Hilfe sog. Sensoren, die bestimmte Fields modifizieren, z.B. zum Zweck der Animation. Manipulatoren sind spezielle Objekte, die Interakionstechniken bereitstellen. Dazu werden nähere Informationen im Abschnitt 3.2.2.6. gegeben.

In Java 3D gibt es ebenfalls Engines für Animationen, und Interaktionen werden ähnlich komfortabel unterstützt. Es gibt eine zentrale Klasse für die Realisierung von Interaktion, Animation und Kollisionserkennung - die Behavior-Klasse, ausführlich beschrieben im Abschnitt 4.2.8.
Die Behavior-Klasse kann zur Animation von visuellen Objekten verwendet werden, ermöglicht aber praktisch, jedes Attribut eines visuellen Objektes zu verändern. Genauso ermöglichen Behaviors das interaktive Verändern von Eigenschaften von Szenen und Objekten. Dabei kann man die vordefinierten Java3D-Behaviors verwenden, aber auch eigene entwerfen. Es gibt verschiedenen Interpolatoren, die bestimmte Parameter eines Szenengraphobjektes manipulieren. Die Java3D-API enthält mehrere fertige Behavior-Klassen, mit denen der Benutzer interaktiv über die Tastatur oder Maus auf die dargestellte 3D Szene einwirken kann, siehe ebenfalls Abschnitt 4.2.8.

5.10. Features

Die generellen Features eines 3D-Graphikprogrammes sind selbstverständlich in beiden Systemen vorhanden, aber die Details sind oft unterschiedlich. In den beschreibenden Kapiteln 3 und 4 wird daher detailliert auf die Einzelheiten von Open Inventor und Java 3D eingegangen. Es hat sich zudem als recht schwierig erwiesen, in beiden Systemen vorhandene Features direkt zu vergleichen, denn es sind eigentlich die Besonderheiten, die den Unterschied ausmachen; hier nur einige Beispiele:

Texturen

Es ist in beiden Systemen die Nutzung von Texturen selbstverständlich mit bestimmten Möglichkeiten und zusätzlichen Optionen gegeben, wie beispielsweise diese:

In Open Inventor lassen sich durch Definition eines Texture2-Knotens Objekte mit zweidimensionalen Texturen versehen. Die Texturen können nur Farbe und Transparenz der Objekte modifizieren. Jedes Objekt kann nur eine Textur besitzen. Es ist aber möglich, reflection textures zu verwenden.

In Java 3D ist es möglich, optische Täuschungen, wie die Textur einer Baumkrone zu verbessern, indem diese immer rechtwinklig zum Betrachter steht. Hier ist es möglich eine Ausrichtung an einer Achse oder an einem Punkt zu realisieren. Die Ausrichtung erfolgt relativ zum primären Betrachter.

Licht und Schatten

Es gibt hier einen durchaus nennenswerten Unterschied. Es existieren keine räumlich ausgedehnten Lichtquellen in Open Inventor. Und wichtig in der Anwendung ist: Lichtquellen werfen keine Schatten. Lichtquellen in Open Inventor: DirectionalLight, PointLight, SpotLight

Lichtquellen in Java 3D: AmbientLight, DirectionalLight, PointLight und SpotLight Hier gibt es also auch das ambiente Licht. Ein großer Vorteil gegenüber Open Inventor ergibt sich dadurch: Java 3D schattiert visuelle Objekte basierend auf einer Kombination von Materialeigenschaften und den Lichtern im virtuellen Universum. Die Schattierung resultiert aus der Anwendung eines Lichtmodells auf die visuellen Objekte im Beisein von Lichtquellen. Daher wurde im entsprechenden Abschnitt 4.2.7. ausführlich auf dieses Thema eingegangen.

Besondere Möglichkeiten von Java 3D

Besonderheiten der API und neue Features, die dieses Programm so interessant machen, sind ausführlich im Abschnitt Java 3D erläutert. Beispielsweise ermöglichen die Blätter ExponentialFogNode und LinearFogNode, **Nebel** beliebiger Farbe zu simulieren.

Weiterhin ist eine große Stärke die Möglichkeit der Einbindung von **Sound**. Mit Java 3D Sound werden vergleichsweise umfangreiche Möglichkeiten geboten, um die audiophile Wahrnehmung dem visuellen Eindruck anzupassen. Da es für die Vollständigkeit der Darstellung von virtuellen Welten wichtig ist, neben der visuellen Modellierung auch die Akustik zu integrieren, ist im Kapitel über Java 3D ein spezieller Abschnitt eingefügt, der mit solchem Umfang nicht unbedingt zur vergleichenden Studie gehört, aber den Informationsgehalt dieser Arbeit für an Java 3D Interessierte bedeutend erhöht.
Der genannte Abschnitt 4.3.8. ist in gewissem Sinne auch als ein – wohl unerwartetes - Ergebnis dieser Studienarbeit zu werten, da vorher nichts über die akustischen Möglichkeiten von Renderingsystemen bekannt war.

5.11. Kollisionserkennung

Java 3D ist auch hier offensichtlich ein Stück weiter bei der eleganten Modellierung und Darstellung von 3D-Welten. Die Hierarchie des Java3D-Szenegraphen unterstützt die örtliche Gruppierung der einzelnen Objekte so, daß **Kollisionserkennung** ermöglicht wird, wobei diese Studienarbeit diesen Bereich nicht detailliert behandelt.

Um dieses Kapitel „Vergleich der beiden Renderingsysteme“ möglichst aussagekräftig zu gestalten, habe ich dafür noch einmal recherchiert. In mir zugänglichen Veröffentlichungen zu Open Inventor konnte ich den Bereich der Kollisionserkennung nicht finden, bin jedoch auf verschiedene Ansätze im Internet gestoßen; beispielsweise in einer Arbeit auf dem Gebiet der

Chemie, in der es heißt „Es gibt bereits eine Kollisiosnerkennung unter Open Inventor für konvexe Objekte.“ Dazu verweise ich Interessierte allerdings auf die Webseite http://www.uni-paderborn.de/~lst/VISMDOCK/NewDock.htm#Collision.

5.12. Zusammenfassung

Abschließend möchte ich die Aussage bekräftigen, dass meiner Meinung nach beide Renderingsysteme gleichwertig sind. Es wird wohl bei der Auswahl der Anwendung einerseits vorwiegend die jeweilige Programmiersprache, also C++ oder Java, entscheidend sein oder aber andererseits einzelne besondere Features, die man unbedingt nutzen möchte, werden den Vorzug erhalten – je nach Ziel und Möglichkeiten des geplanten Einsatzes.
In der Fachliteratur und in Internet-Veröffentlichungen finden sich Bewertungen, die Open Inventor als das 3D-Graphiksystem für die professionelle Anwendung bezeichnen. Allerdings hat das Renderingsystem von Open Inventor selbst eine Reihe von visuellen Schwächen, deren Hervortreten durch kreativen Einsatz vermieden werden muß, wie zum Beispiel bereits erwähnt, dass es im Bereich der Darstellung von Beleuchtungen einer Szene keine Schatten gibt.
Begeistert sind sicher alle 3D-Entwickler von den genannten besonderen Features von Java 3D, sowie der hervorragenden Möglichkeiten der Netzwerkfähigkeit und der Integration im Internet. Letzteres ist recht einfach, denn insbesondere für Java3D-Applets existieren ja bereits Plug-In’s von Netscape bzw. Internet Explorer.

Literaturverzeichnis

H. Sowrizal, K. Rushforth, M. Deering, The Java 3D API Specification. 2.Edition, Addison-Wesley, 2000

Wernecke, Josie. The Inventor Mentor: Programming Object-Oriented 3D Graphics with OpenInventor, Release 2. Addison-Wesley Publishing Company, 1994

G. Ward Larson, R. Shakespear. Rendering with Radiance. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, 1998

Open Inventor Architecture Group. Open Inventor C++ Refernce Manual:
The Official Reference Document for Open Inventor, Release2. Addison-Wesley Publishing Company, 1994

Wernecke, Josie. The Inventor Toolmaker: Extending OpenInventor, Release 2. Addison-Wesley Publishing Company, 1994

Das Apple ColourBook, Apple Computer, Marketing Publishing,Ismaning, Juni 1993

Foley, James D. van Dam; Andries, Feiner, Steven K. and Hughes, John F. “Computer Graphics -Principles and Practice “, The System Programming Series, 2nd edition, Addison Wesley, USA, 1990

Internet-Seiten

JAVA 3D
http://www.javasoft.com/products/javamedia/3D/index.html allgemein
http://java.sun.com/products/java-media/3D Firmenseite von SUN
http://www.java.de Java User Group Deutschland e.V.

J3DOPENGL1.2: Java 3DTM 1.2 API
http://java.sun.com/products/javamedia/3D/download.html

J3DOPENGL1.1.3: Java 3DTM 1.1.3 API
http://java.sun.com/products/javamedia/3D/1.1.3/download.html

J3DSPEC: Java 3D Specification 1.1 und 1.2
http://java.sun.com/products/javamedia/3D/

LINUX: Java 3D for Linux
http://www.blackdown.org/java-linux/jdk1.2-status/java-3d-status.html

SGI: Java 3DTM v 1.1.3 API for SGI IRIX® MR Release
http://www.sgi.com/developers/devtools/languages/java3d113.html

HP: SUN INKS AGREEMENT WITH HEWLETT-PACKARD
http://java.sun.com/pr/1999/08/pr990811-01.html

DIRECT3D: Java 3DTM 1.2 Beta API for Direct3D
http://developer.java.sun.com/developer/earlyAccess/java3D/directx/

FLA: File Loader Archives
http://www.j3d.org/utilities/loaders.html

NCSA: NSCA Portfolio
http://www.ncsa.uiuc.edu/~srp/Java3D/portfolio/index.html

SENSOR: Sensor Archives
http://www.j3d.org/utilities/sensors.html

VOL: Java 3D Volume Rendering
http://www.j3d.org/utilities/volumepro.html

DOOM: Keyboard Navigation
http://www.j3d.org/utilities/keyboard.html

J3DSV: Java 3D Scenegraph Viewer
http://www.tornadolabs.com/News/J3dTree_Home/j3dtree_home.html

BUGS: Java 3DTM 1.2 API Known Issues/Bugs
http://java.sun.com/products/java-media/3D/java3d-bugs.html

PHIGS: Programmer's Hierarchical Interactive Graphics System (PHIGS)
http://delonline.cern.ch/HELP/PHIG

INVENTOR: SGI - Open GL Inventor
http://www.sgi.com/software/inventor/faq.html#WhatIsOpen

SCENEGRAPH: Scene Graph Basics
http://www.j3d.org/tutorials/raw_j3d/chapter1/scene_graph_basics.html

CAMERA: Cameras and Viewing
http://www.j3d.org/tutorials/raw_j3d/chapter1/viewing.html

Studie zu 3D-Systemen
http://www.studierstube.org/media/documents/MarxThesis.pdf

Kollisionserkennung unter Open Inventor
http://www.uni-paderborn.de/~lst/VISMDOCK/NewDock.htm#Collision)