

**Ein konzeptioneller Vergleich der 3D-Graphik-Schnittstellen  
OpenGL, Java3D und VRML hinsichtlich ihrer Eignung  
für die Erstellung von Animationen**

**Diplomarbeit in Informatik**

**von**

**Christian Stein**

**Anleiter:**

**Prof. Dr. Manfred Sommer**

**Philipps-Universität Marburg**

**Fachbereich Mathematik und Informatik**

**Abgegeben im Dezember 1998**

# Inhaltsverzeichnis

|   |    |
|---|----|
| 1. Historische Entwicklung der 3D-Graphik-Schnittstellen.....               | 3  |
| 1.1 Der Begriff: Schnittstelle.....   | 3  |
| 1.2 Der Begriff: Animation .....  | 4  |
| 1.3 Low-Level 3D-Graphik-Schnittstelle: OpenGL.....                         | 4  |
| 1.4 Low-Level 3D-Graphik-Schnittstelle: Java3D.....                         | 5  |
| 1.5 Very-High-Level 3D-Graphik-Schnittstelle: VRML .....                    | 5  |
| 2. Charakteristische Leistungsmerkmale der 3D-Graphik-Schnittstellen .....  | 6  |
| 2.1 OpenGL .....  | 6  |
| 2.2 Java3D .....  | 6  |
| 2.3 VRML.....   | 9  |
| 2.4 Zusammenfassung .....   | 10 |
| 3. Animation durch Programmiersprachen .....                                | 11 |
| 3.1 Die VRML-Schnittstelle zu Programmiersprachen.....                      | 11 |
| 3.2 Die OpenGL-Bilderzeugungsmechanismen .....                              | 13 |
| 3.3 Java3D und Java .....   | 15 |
| 3.4 Zusammenfassung .....   | 16 |
| 4. Das Animationsmodell von VRML .....                                      | 17 |
| 4.1 Variablen, Parameter, Felder und Nachrichten .....                      | 17 |
| 4.2 Der Route-Mechanismus .....   | 18 |
| 4.3 Sensoren .....  | 21 |
| 4.4 Interpolatoren .....  | 21 |
| 4.5 Der Script-Knoten .....   | 22 |
| 4.6 Optimierungen durch das Animationsmodell.....                           | 25 |
| 4.7 Zusammenfassung .....   | 26 |
| 5. Das Animationsmodell von Java3D.....                                     | 27 |
| 5.1 Rendermodi .....  | 27 |
| 5.2 Eingabegeräte und Sensoren.....   | 28 |
| 5.3 Der Scheduler .....   | 30 |
| 5.4 Interpolatoren und ihre Aktivierungsfunktion.....                       | 32 |
| 5.5 Die Verhalten- und die Auswahl-Klasse .....                             | 33 |
| 5.6 Optimierungen durch das Animationsmodell.....                           | 35 |
| 5.7 Zusammenfassung .....   | 36 |
| 6. Vergleich der Animationsmöglichkeiten der 3D-Graphik-Schnittstellen..... | 37 |
| 6.1 Verhalten-Objekte und Script-Knoten.....                                | 37 |
| 6.2 Unterschiede in der Interpolator-Realisierung .....                     | 40 |
| 6.3 Unterschiede bei den Ereignissen.....                                   | 43 |
| 6.4 Auswahl-Methoden und Drag-Sensoren .....                                | 47 |
| 6.5 Scheduleregionen und Bounding-Boxes .....                               | 48 |
| 6.6 Der Scheduler und der Route-Mechanismus.....                            | 51 |
| 6.7 VRML-Browser in Java3D.....   | 53 |
| 6.8 Zusammenfassung .....   | 55 |
| 7. Die abschließende Bewertung der 3D-Graphik-Schnittstellen .....          | 57 |
| 7.1 VRML.....   | 57 |
| 7.2 Java3D .....  | 58 |
| 7.3 OpenGL .....  | 59 |
| 7.4 Spiele und 3D-Grafik-Schnittstellen .....                               | 59 |
| 7.5 Schlußbemerkung .....   | 60 |

|   |    |
|---|----|
| Anhang A : Known Issues and Bugs in Java3D 1.1 Beta 2 Release.....            | 61 |
| Anhang B : Known Bugs (VRML mit Java3D).....                                  | 63 |
| Anhang C : Email von Henry Sowizral (Java3D).....                             | 65 |
| Anhang D : Email von Kevin Rushforth (Java3D) .....                           | 66 |
| Anhang E : Statistikwerte einer Umfrage zum Internet (Teilbereich Java) ..... | 67 |
| Abbildungsverzeichnis .....   | 68 |
| Tabellenverzeichnis .....   | 69 |
| Programmbeispielverzeichnis .....   | 70 |
| Literatur .....   | 71 |
| URLs .....  | 73 |

## 1. Historische Entwicklung der 3D-Graphik-Schnittstellen

Um einen Einblick in die Problematik der Arbeit zu bekommen, werden in diesem Kapitel die elementaren Begriffe Schnittstelle und Animation erläutert. Dabei wird nicht auf Begriffe der 3D-Graphik eingegangen, da dies den Umfang dieser Arbeit sprengen würde. Für eine allgemeine Einführung in die 3D-Graphik ist [Foley] zu empfehlen. Außerdem wird in diesem Kapitel die historische Entwicklung der drei Schnittstellen aufgezeigt.

### 1.1 Der Begriff: Schnittstelle

Im Zuge der sich auseinander entwickelnden Computerhardware ist es wichtig Standards zu finden. Dabei sollen Standards die Möglichkeiten verschiedener Hardware vollständig ausschöpfen. Der Versuch, einzelne Standards in eine Hardwareabstraktionshierarchie einzuordnen, bereitet jedoch Schwierigkeiten. Nicht einfach zu beantworten ist die Frage, was baut auf wen auf, oder was 'ruft' wen auf.

Ein Beispiel sind *OpenGL* und *Direct3D*. Momentan sind beide Standards konkurrierend, jedenfalls im Microsoft-Windows-Bereich, wobei OpenGL mehr von *CAD* (Computer-Aided Design)-Programmen und Direct3D mehr von Spielen genutzt wird. Microsoft will aber bei zukünftigen OpenGL-Versionen auf Direct3D aufbauen (siehe Abbildung 1), das heißt, OpenGL rückt in der Hardwareabstraktionshierarchie (ein wenig) nach oben. Der Vorteil für Microsoft ist klar, sie brauchen nur eine OpenGL-Version für ihre Betriebssysteme erstellen, da alle neuen Microsoft-Betriebssysteme Direct3D besitzen. Direct3D muß aber weiterhin pro Betriebssystem einmal erzeugt werden.

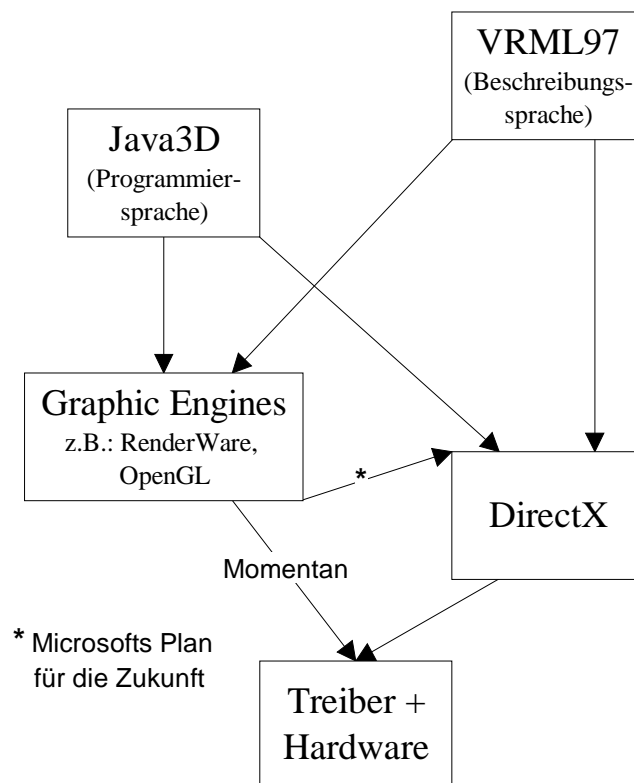


Abbildung 1: Hierarchie der 3D-Graphik-Schnittstellen  
(Umwandlung von [Hase, Seite 280] und [MS 98-1])

In der Zukunft wollen Microsoft und Silicon Graphics Inc. (SGI) eine gemeinsame 3D-Graphik-Schnittstelle erstellen mit dem Namen *Fahrenheit*. Dabei soll Fahrenheit die Vorzüge

von OpenGL und Direct3D bekommen. Der Zeitplan sieht allerdings vor, daß dies frühestens im Jahre 2000 geschehen soll [SGI 97-1].

Ein weiteres Beispiel liefert eine nicht offizielle OpenGL-Schnittstelle für Java. Diese Schnittstelle bietet Java-Programmen die Möglichkeit, auf OpenGL zurückzugreifen [VRML 98-1]. Es sei bemerkt, daß Java auf einer höheren Abstraktionsebene als OpenGL liegt.

Bei Schnittstellen wie *Java3D* und *VRML* existieren ähnliche Probleme. *Java3D* ist eine Programmiersprachenerweiterung und *VRML* eine Beschreibungssprache, das heißt *Java3D* ist eine 'Low-Level'-Schnittstelle und *VRML* eine High-Level-Schnittstelle. Beide Schnittstellenrealisierungen greifen aber auf dieselben, tiefer liegenden Hardwareabstraktionsschichten zu (siehe Abbildung 1). Es ist also interessant zu untersuchen, ob ein *VRML*-Browser mit *Java3D* geschrieben werden kann, oder ob die beiden 3D-Graphik-Schnittstellen so unterschiedlich in ihrem Aufbau sind, daß dies nicht sinnvoll ist. Dieses Thema beschäftigt zur Zeit auch die 'Java3D and VRML Working Group' [VRML 98-1]. Interessant sind hier besonders die verschiedenen Konzepte der beiden Schnittstellen (*VRML* und *Java3D*) zur Verwirklichung von Animationen in 3D-Welten.

## 1.2 Der Begriff: Animation

Es wirft sich die Frage auf, was unter dem Begriff 'Animation' zu verstehen ist. Wenn das Abbild einer 3D-Welt, z.B. auf dem Bildschirm sein Aussehen verändert, dann ist das Animation. Das heißt, auch die Veränderung des Blickwinkels des Betrachters auf eine 3D-Szene, die sogenannte Navigation durch die 3D-Szene, ist eine Animation. Diese Argumentation basiert darauf, daß das Abbild der 3D-Welt sich verändert, es ist also irrelevant, daß die 3D-Welt dabei die gleiche bleibt. Es sind somit zwei Arten von Animationen möglich, die Navigation durch die 3D-Szene und die Veränderung von Daten der Szene. Da es möglich ist, Animation mit allen 3 Schnittstellen zu erzeugen, kann ein Vergleich der Techniken, um Animationen zu erstellen, durchgeführt werden.

Bei den Schnittstellen gibt es verschiedene Möglichkeiten Animationen zuwege zu bringen, die nicht direkt durch deren Funktionen erzeugt wird. Das heißt, die Animation wird durch Veränderung des einzelnen Objektes erreicht, wobei die Veränderung nicht durch eine von dem 3D-Graphik-Schnittstellen bereitgestellten Funktion geschieht, sondern durch die entsprechende Programmiersprache. Diese Animationen haben dann allerdings den Nachteil, das sie nicht geschwindigkeitsoptimiert sind, da das 3D-Welt-Abbild, im ungünstigsten Fall, komplett neu berechnet wird. Inwieweit die Animationstechniken geschwindigkeitsoptimiert sind, wird auch Bestandteil dieser Diplomarbeit sein.

Nach der allgemeinen Einführung in die Begriffe Animation und Schnittstelle, folgt nun die Darstellung der historischen Entwicklung der 3D-Graphik-Schnittstellen für jede Schnittstelle einzeln.

## 1.3 Low-Level 3D-Graphik-Schnittstelle: OpenGL

OpenGL (Open Graphics Library) ist jetzt schon einige Jahre alt. Sie ist aus *IRIS GL* hervorgegangen, die wiederum in den 80'ern entstanden ist. 1992 wurde von dem OpenGL *Architecture Review Board* (ARB) die Version 1.0 erstellt [OpenGL 98-1, S. 225]. Damals galt es, erst einmal überhaupt realistisch wirkende 3D-Welt-Abbilder zu erstellen. Dementsprechend ist der Standard auf 'statische' 3D-Welten ausgelegt, das heißt, er war für CAD-Programme gedacht. Da OpenGL eine Programmiersprachenbibliothek ist, d.h. es wird durch Funktionsaufrufe die 3D-Welt verändert / erstellt, ist die Erstellung von Animationen möglich, wenn auch nicht sehr effektiv. In den neueren Versionen (1.1 und 1.2) hat sich an dem Konzept von OpenGL nichts verändert. Es sind nur einige neue Funktionen hinzugekommen, die sich hauptsächlich um Texturen (1.1 und 1.2) oder um die Darstellung von Farben (1.2) kümmern [OpenGL 98-1,

Anhang C und D]. Eine OpenGL-Library gibt es für fast jede Programmiersprache, die am häufigsten benutzte Programmiersprache ist C. C bietet Vorteile in der Plattformunabhängigkeit, wie OpenGL, und bei der Performance. Daher wird in dieser Arbeit immer davon ausgegangen, daß OpenGL in Verbindung mit C benutzt wird.

### **1.4 Low-Level 3D-Graphik-Schnittstelle: Java3D**

Java3D ist die neueste Schnittstelle (Dezember 1998) und hat die meisten, schon implementierten Möglichkeiten, Animationen zu erstellen. Da Java3D eine Programmiersprachenbibliothek für Java ab Version 1.2 ist, ist es wie bei OpenGL möglich, alleine durch die Programmiersprache, also Java, Animationen zu realisieren. Java3D hat aber ein anderes Ziel als OpenGL, denn es soll zur Visualisierung von Daten, Waren oder gar virtueller Welten (im Internet) dienen [Sun 98-1, Seite 1].

### **1.5 Very-High-Level 3D-Graphik-Schnittstelle: VRML**

1994 wurde VRML (Virtual Reality Modeling Language) entwickelt, damals hatte sie im WWW (World Wide Web) nur den Anspruch 3D-Objekte zu visualisieren. VRML ist im Gegensatz zu Java3D oder OpenGL keine API (Application Programming Interfaces), sondern eine Beschreibungssprache. Die Syntax für VRML 1.0 wurde von dem OpenInventor-Fileformat abgeleitet [Kloss, S. 34]. VRML ist also keine Programmiersprachenerweiterung, sondern vollkommen unabhängig von einer Programmiersprache. Diese Eigenschaft gibt ihr eine Sonderrolle unter den 3D-Graphik-Schnittstellen. Sie ist speziell für das WWW zur Darstellung von 3D-Welten definiert worden. Die Verwendung als 3D-Graphik-Schnittstelle stand bei der Definition von VRML nicht im Vordergrund. Zur Zeit gibt es Bestrebungen, den Leistungsumfang von VRML zu erweitern; durch die Leistungsumfangserweiterungen sind Animationen möglich, mit denen sogar virtuelle Welten erstellt werden können. Der erste Schritt in diese Richtung wurde mit VRML 2.0 unternommen. VRML 2.0 ist der direkte Vorgänger von VRML97, der aktuellen Version. Durch die Integration von Sensoren und sogenannten *Script-Knoten* sind nun Interaktionen möglich. Als sich die Script-Knoten nicht als mächtig genug herausgestellt haben, wurde eine zusätzliche Schnittstelle zu Programmiersprachen erstellt, das External Authoring Interface (EAI). Diese Schnittstelle ermöglicht, daß ein Applet unabhängig, das heißt, parallel zu einem VRML-Browser arbeitet und trotzdem in Kontakt mit dem VRML-Browser treten kann. Die Framerate leidet aber darunter, daß zwei verschiedene Programme benutzt werden müssen. Es handelt sich bei den Programmen um den VRML-Browser, inkl. einer 3D-Rendermaschine, und den Interpreter für die Sprache, z.B. Java [Kloss, S. 295-296].

## 2. Charakteristische Leistungsmerkmale der 3D-Graphik-Schnittstellen

In diesem Kapitel sollen die Merkmale der 3 Schnittstellen kurz umrissen werden, damit ein Eindruck ihrer Fähigkeiten entsteht. Es wird dabei so vorgegangen, daß die einzelnen Möglichkeiten Animationen zu erstellen, schnittstellenweise beschrieben werden.

### 2.1 OpenGL

Da OpenGL für CAD-Programme gemacht ist, das heißt, es wurde mehr Wert auf die Qualität der Darstellung gelegt, als auf Animationsmechanismen, sind die Animationsmöglichkeiten hier auf 'manuelle' Veränderung der 3D-Objekte beschränkt. Wenn sich ein Objekt drehen soll, muß für jedes Bild eine neue *Rotationsmatrix* angegeben werden und dann wird jedes Bild komplett neu berechnet. Die Liste der Möglichkeiten, Animation mit Hilfe einer Programmiersprache zu erstellen, ist lang. In dieser Arbeit geht es aber darum, die Möglichkeiten zu beschreiben, die Schnittstellen bieten, um Animationen zu erstellen und nicht um die Möglichkeiten, welche die Programmiersprachen bieten. Deshalb wird im 3. Kapitel nur ein kleiner Einblick in diese Thematik gegeben.

### 2.2 Java3D

Das Design von Java3D ist nicht ausschließlich auf statische 3D-Welten ausgelegt, sondern unterstützt explizit den Entwurf von dynamischen 3D-Welten. Java3D besitzt drei verschiedene Rendermodi. Der *immediate mode* hat keine vorgegebene Struktur, ähnlich wie OpenGL, ist aber auch nicht geschwindigkeitsoptimiert. Der *retained mode* hat eine Baumstruktur (siehe Abbildung 2), so daß er geschwindigkeitsoptimiert werden kann. Der *compiled-retained mode* ist von der Struktur her wie der retained mode, allerdings wird er vor der Laufzeit in ein starres internes Format gebracht. Dieses Format ist noch einmal geschwindigkeitsoptimiert. In dem compiled-retained mode muß vorher angegeben werden, welche Aktionen mit den einzelnen Variablen möglich sind. So muß angegeben werden, ob die *Translation-Variable* eines *Transform-Objektes* geändert werden darf. Die folgenden Ausführungen beziehen sich auf den compiled-retained mode.

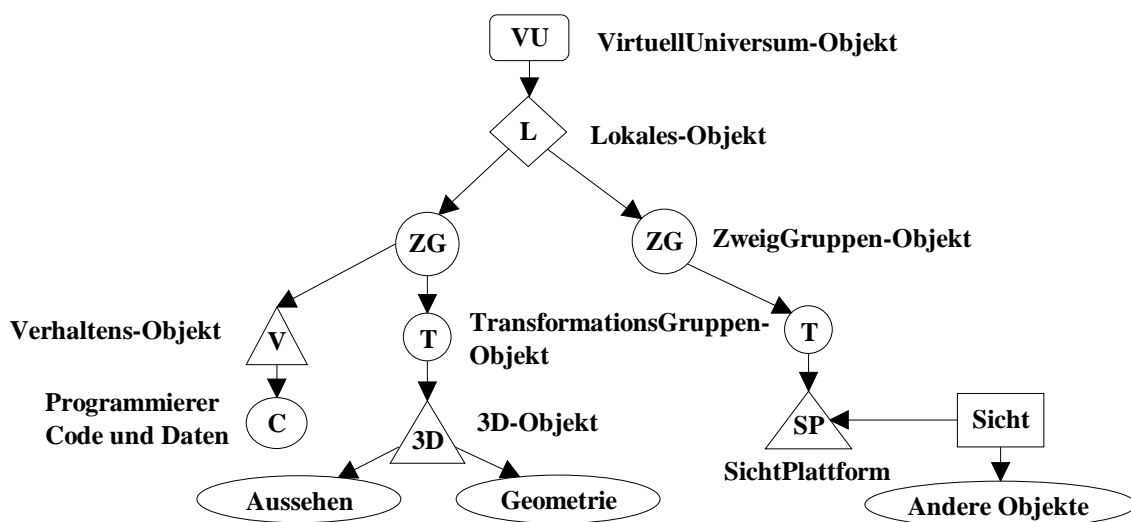


Abbildung 2: Die Baumstruktur des (compiled-) retained mode in Java3D  
(Übernommen aus Sun 98-1, Seite 7)

Es werden nun die verschiedenen Möglichkeiten, Animationen in Java3D zu erstellen, aufgezählt.

*Verhalten-Objekte* dienen dazu, beliebige Aktionen auszuführen und werden in der Regel dazu genutzt, Veränderungen an den Daten der 3D-Welt vorzunehmen. Es sind innerhalb solcher Objekte beliebige Berechnungen / Abfragen möglich, soweit sie Java zuläßt. Der Zusatz ist, daß bestimmte Variablen eines Transformation-Objektes verändert werden können. So kann dort bestimmt werden, daß die Translation-Variable einen bestimmten Wert zugewiesen bekommt. Die Entwickler von Java3D haben sich für ein *Aufwecksystem* entschieden, um die Verhalten-Objekte aufzurufen. Wenn eines der vordefinierten Ereignisse (siehe Liste) eintritt, dann teilt der *Scheduler* dieses den Verhalten-Objekten mit, die in seiner entsprechenden Datenstruktur stehen, also denjenigen, die auf das entsprechende Ereignis warten. Die verschiedenen Ereignisse lassen sich beliebig mit AND und OR zu neuen Arten von Ereignissen verknüpfen.

Die Liste der Ereignisse, die Java3D kennt:

- Betreten / Verlassen einer vordefinierten Region durch den Betrachter
- Verhalten-Objekt wird de-/aktiviert, Scheduleregion wird betreten / verlassen
- Transformationswertänderung eines Transformation-Objektes
- Kollision von geometrischen Objekten findet statt / eines der Objekte bewegt sich während eine Kollision / Kollision findet nicht mehr statt
- Verhalten-Objekt sendet Ereignis
- AWT (Abstract Windowing Toolkit)-Ereignis, z.B. Menüpunkt wurde ausgewählt
- Zeit-Ereignis
- Anzahl der dargestellten *Frames* ist erreicht
- *Eingabegerät* 'betritt' / 'verläßt' vordefinierte Region

Weitere Schritte in Sachen Dynamik können dann mit den Interpolatoren gemacht werden. Wenn ein Objekt sich von alleine um sich selber drehen soll, wird der Anfangs- und der Endpunkt angegeben sowie eine Aktivierungsfunktion. Die Aktivierungsfunktion gibt an, zu welchem relativen Zeitpunkt auf welche Art zwischen diesen Werten interpoliert wird und den Rest macht die API. Die ist wiederum darauf optimiert, nur den Teil der Szene neu zu berechnen, der durch die Veränderung beeinflußt wird. Es wird für diese Zwecke nicht nur ein Interpolator und eine Aktivierungsfunktion, sondern auch ein Zeitgeber benötigt. Der Zeitgeber bei Java3D ist im Scheduler mit integriert.

Es gibt die folgenden Interpolatoren:

- Rotations-Interpolator
- Skalierungs-Interpolator
- Positions-Interpolator
- Farben-Interpolator
- Transparenz-Interpolator
- Switchnode-Interpolator
- einige Weg-Interpolator

Die Weg-Interpolatoren dienen dazu, Objekte anhand eines 'Weges' durch die 3D-Welten zu erstellen. Im Gegensatz zum Positions-Interpolator können hier mehrere Stützpunkte angegeben werden.

Es gibt die folgenden Weg-Interpolatoren:



- Rotations-Weg-Interpolator
- Positions-Weg-Interpolator
- Positions-Rotations-Weg-Interpolator
- Positions-Rotations-Skalierungs-Weg-Interpolator

Mit einigen von den Weg-Interpolatoren ist es somit möglich, auch die Orientierung oder gar die Größe des Objektes zu beeinflussen. Die Interpolatoren sind Bestandteil der Transformations-Objekte. Trotzdem muß bei der Verwendung von Interpolatoren im compiled-retained mode die Erlaubnis, daß zu Laufzeit der Variablenwert geändert werden darf, explizit gegeben werden (siehe Programmbeispiel 1).

```
TransformGroup objTrans = new TransformGroup();
// Die Erlaubnis geben, daß die Transformationsmatrix
// während der Laufzeit geändert werden darf
objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
// Das geo.Objekt erzeugen und an das Transformation-Objekt anhängen
objTrans.addChild(new ColorCube(0.4));
// Die Transformationsmatrix erzeugen (default: Identitätsmatrix)
Transform3D yAxis = new Transform3D();
// Die Aktivierungsfunktion erstellen
Alpha rotationAlpha = new Alpha(-1, Alpha.INCREASING_ENABLE,
    0, 0, 4000, 0, 0, 0, 0, 0);
// Den Interpolator erzeugen
RotationInterpolator rotator = new RotationInterpolator
    (rotationAlpha, objTrans, yAxis, 0.0f, (float) Math.PI*2.0f);
// Den Interpolator an das selbe Transformation-Objekt anhängen
objTrans.addChild(rotator);
Programmbeispiel 1: Der Rotations-Interpolator in Java3D
```

Außerdem existiert eine *Billboard-Klasse*, die es ermöglicht, z.B. eine 2D-Textur von einer Baumkrone auf eine Kugel zu legen, so daß die Textur immer vom Betrachter zu sehen ist. Es wird so auf einfachste Weise eine '3D-Baumkrone' visualisiert.

Die *level-of-detail-Klasse* (LoD) dient dazu, entsprechend der Entfernung des Betrachters zu ihm, die Detailstufe eines geometrischen Objektes zu regulieren. Es wird also eines der *Kind-Objekte* des LoD-Objektes bestimmt, das dann als einziges dargestellt wird. Wenn z.B. ein komplexes geometrisches Objekt noch so weit entfernt ist, daß nur ein Umriß von ihm zu erkennen ist, dann ist es sinnvoll auch nur den Umriß beim Rendern zu beachten und nicht das komplette Objekt. Was dabei der Umriß ist, liegt dann im Ermessen des Programmierers und nicht der 3D-Graphik-Schnittstelle. Es existieren also so viele 'Kind-Objekte', wie es Detailstufen gibt.

Weiterhin gibt es eine Klassenfamilie, die sich mit dem *Anfassen* von Objekten in der 3D-Welt beschäftigt. Diese Klassenfamilie ist abstrakt gehalten. Es wird ein beliebiges Eingabegerät als *Anfasser* benutzt und auch die Aktion, die durch das Anfassen geschehen soll, ist nicht vorher definiert, bei VRML ist sie z.B. an das Drehen von Objekten gebunden. Die Utils-Klassensammlung, die bei Java3D mitgeliefert wird, hat ein paar Anwendungsfälle schon implementiert. Es wird zum Beispiel auch hier das Drehen von Objekten durch das *Anfassen* mit der Maus ermöglicht. Dabei handelt sich es in der Regel um ein paar wenige Programmaufrufe, die zu einer Klasse zusammengefaßt werden.

Eine schöner Zusatz ist die Morph-Klasse, sie benötigt ein Array von Arrays von geometrischen Daten. Sie erzeugt dann eine Interpolation zwischen den verschiedenen Array-Daten. So

kann z.B. durch 4 Arrays, die jeweils Daten eine Momentaufnahme einer winkenden Hand enthalten, eine Hand erzeugt werden, die scheinbar winkt.

Mit dem Vorsatz Java3D zu einem umfassenden Werkzeug werden zu lassen, wurde eine Headtrackerunterstützung integriert. So kann durch das Bewegen des Kopfes, wenn auf ihm ein Headtracker sitzt, die Betrachterposition / -orientierung in der 3D-Welt geändert werden [Sun 98-1].

## 2.3 VRML

In seiner neuesten Fassung (VRML 97) ist VRML eine 3D-Beschreibungssprache die dynamische 3D-Welten erzeugen kann. Bei VRML wird Interaktion / Dynamik durch das symmetrische Verändern von Variablen gesteuert, indem ein Verweis zwischen zwei Variablen erstellt wird. Das heißt, wenn die erste Variable sich ändert, ändert sich die zweite auf den selben Wert. Dynamik kann aber erst dann entstehen, wenn ein *Sensor* aktiviert wird. Es ist zu beachten, daß der Begriff Sensor in Java3D anders genutzt wird, der Begriff dient dort als Name für eine abstrakte Klasse, die Eingabegeräte-Werte einliest und weitergibt.

Es gibt folgende Sensoren:

- Sichtbarkeit-Sensor
- Zeit-Sensor
- Kollisions-Sensor, von einem Objekt mit dem Betrachter
- level-of-detail-Sensor
- Bereichs-Sensor
- Anklick-Sensor
- Anchor
- Zylinder-Sensor
- Sphären-Sensor
- Ebenen-Sensor

Die letzten drei Sensoren dienen dazu, ein Objekt durch die Maus zu drehen, entweder nur im 'Kreis' (Zylinder-Sensor) oder beliebig (Sphäre-Sensor), oder auf einer Ebene zu verschieben. Die anderen Sensoren erklären sich selbst, bzw. siehe Kapitel 2.2 (Seite 6). Die Unterschiede der Sensoren von VRML und der Ereignisse von Java3D wird im Kapitel 6.3 (Seite 6) behandelt. Es existieren auch in VRML Interpolatoren, die durch den Wert des Zeit-Sensors ihre Zwischenwerte berechnen.

Es gibt die folgenden Interpolatoren:

- Rotationsinterpolator
- Positionsinterpolator
- Skalierungsinterpolator
- Farbwertinterpolator
- Koordinateninterpolator
- Normaleninterpolator

Der Koordinateninterpolator wird dazu genutzt, einzelne Koordinaten eines Objekts zu interpolieren, also nicht seine Position, sondern einen einzelnen (Eck)-Punkt in dem Objekt, vergleiche Morph-Klasse von Java3D.

Bei VRML gibt es auch einen Billboard- und einen level-of-detail-Knoten (vergleiche Kapitel 2.2, Seite 6). Außerdem besteht die Möglichkeit, eine Videodatei als Textur anzugeben, womit sich z.B. sehr einfach ein Fernseher simulieren läßt.

Damit auch nicht vordefinierte Berechnungen möglich sind, existiert der Script-Knoten. Dieser bildet entweder eine Schnittstelle zu Java-Applets oder zu Script-Dateien, oder enthält einen Code, der in einer von dem Browser unterstützten Script-Sprache geschrieben ist, wie z.B. VRML-Script, eine Teilmenge von Javascript.

Eine Art von Animationsmöglichkeit, in dem Teilbereich Navigation, bietet das Festlegen von *Ansichtspunkten*. Durch die Definition eines Ansichtspunktes kann der Betrachter an einen vorbestimmten Ort gebracht werden, dies allerdings nur einmal und zwar, wenn die VRML-Datei geladen wird. Weitere Ansichtspunkte können definiert werden und von dem Betrachter in einer Liste ausgewählt werden. Von dort kann der Betrachter sich dann mit Hilfe des Browsers in der 3D-Welt bewegen. Wenn der Benutzer durch die 3D-Welt geführt werden soll, dann muß er mit einem Objekt, z.B. einem Auto, verknüpft werden, welches sich dann mit Hilfe von Interpolatoren durch die 3D-Welt bewegt [VRML 97-1].

### 2.4 Zusammenfassung

In diesem Kapitel wurde festgestellt, daß OpenGL nur Funktionen für die Erstellung von statischen Bildern bereitstellt und Animationen 'per Hand' erstellt werden müssen. Hingegen bestehen in Java3D und VRML Konzepte, um Animationen zu erstellen. Die Konzepte können in drei Bereiche aufgeteilt werden, dies sind:

- Sensor / Ereignis
- Interpolator
- Verhalten-Objekt / Script-Knoten

Diese Aufteilung ermöglicht Interaktionen zwischen Benutzer und 3D-Welt (Sensor /Ereignis), statische / vorgegebene Veränderung der 3D-Daten (Interpolator) und dynamische Veränderung der 3D-Daten (Verhalten-Objekt / Script-Knoten). Außerdem bieten die beiden Schnittstellen weitere Möglichkeiten zur Vereinfachung der Erstellung von Animation. Es sind dies unter anderem die LoD- und MovieTexture-Knoten, sowie die LoD- und Morph-Klassen.

### 3. Animation durch Programmiersprachen

Animationen können in OpenGL nur durch die Programmiersprache erzeugt werden, die OpenGL als Bibliothek enthält, also in der Regel die Sprache C (siehe auch Kapitel 2.1, Seite 6). Da das Prinzip für die drei 3D-Graphik-Schnittstellen ähnlich ist, wird dies an dieser Stelle für alle drei 3D-Graphik-Schnittstellen abgehandelt. Es ist Ziel dieses Kapitels, die verschiedenen Ansätze, Programmiersprachenelemente zu nutzen, zu verdeutlichen. Allgemein findet eine Veränderung der 3D-Welt durch eine Programmiersprache statt, indem die Parameter von den 3D-Graphik-Schnittstellen-Befehlen, anstatt Konstanten, Variablen sind. Dann werden diese Variablen durch beliebige Berechnungen verändert und so bei jedem Verarbeiten des 3D-Graphik-Schnittstellen-Befehls ein entsprechend modifizierter Parameter übergeben. Bei VRML ist dieses Prinzip ein wenig abgeändert (siehe 3.1).

Der Leistungsumfang der Möglichkeiten, um Animationen zu erstellen, hängt in diesem Bereich sehr stark von der Art der Schnittstelle zu den entsprechenden Programmiersprachen ab. Welche 3D-Graphik-Schnittstellen zu welcher Programmiersprachen in Verbindung steht, ist Tabelle 1 zu entnehmen.

| 3D-Graphik-Schnittstellen | Programmiersprache  |
|---------------------------|---|
| OpenGL                    | C, und fast jede andere Sprache   |
| Java3D                    | Java  |
| VRML                      | Java, eine Script-Sprache, die der WWW-Browser unterstützt, z.B. Javascript<br>EAI: Java und jede andere Sprache, die EAI unterstützt |

Tabelle 1: 3D-Graphik-Schnittstellen und Programmiersprachen

#### 3.1 Die VRML-Schnittstelle zu Programmiersprachen

Obwohl VRML einige Sprachen unterstützt, so ist doch die Schnittstelle zu der einzelnen Sprache bei OpenGL und Java3D natürlicher und effizienter. Das liegt vor allem daran, daß die beiden APIs Bestandteile der entsprechenden Sprache sind. Bei OpenGL ist es eine Bibliothek, bei Java3D sind es Pakete. VRML hingegen ist eigenständig und muß eine Client/Server-Verbindung zu anderen Applets/Scripten erstellen [Roehl, S. 62-63]. Aus der Abbildung 3 (Seite 12) ist zu ersehen, daß JavaScript-Scripte nur über den Script-Knoten in Verbindung zum VRML-Browser treten und sich dabei in der Rolle des Clients befinden. Hingegen kann ein Java-Objekt ein *Kindprozeß* vom VRML-Browser-Prozeß (Script-Knoten) oder ein unabhängiger Prozeß, mittels EAI, vom VRML-Browser sein. Das bedeutet: Wenn ein Applet über das EAI mit dem VRML-Browser kommuniziert, dann hat es alle Rechte, die ein Applet normalerweise hat. Wird aber ein Java-Objekt über ein Script-Knoten aufgerufen, dann ist das Java-Objekt an den VRML-Browser-Prozeß gebunden und kann nicht unabhängig von ihm handeln. Das bedeutet, daß ein Java-Class-File, das von einem Script-Knoten nachgeladen wird, nur eine Klasse in sich haben darf, die ein Nachkommen der Script-Klasse ist. Objekte dieser Klasse haben aber keine Möglichkeit, AWT-Methoden zu benutzen oder gar selbständig ein Ereignis zu erzeugen. Es kann zwar mittels eines Tricks, dem asynchronen Verhalten (siehe Kapitel 6.1, Seite 37) dieses Manko behoben werden, das EAI aber bietet einen direkteren Zugang zur Benutzung des AWTs.

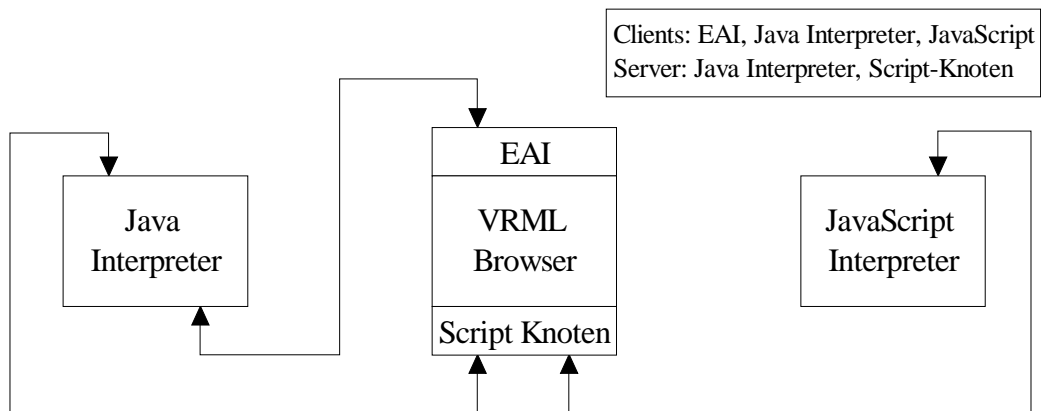


Abbildung 3: VRML-Interfaces-Übersicht  
(Abwandlung von [Marrin] bzw. [Roehl, Seite 63])

Statt Java kann auch eine beliebige andere Programmiersprache benutzt werden, wenn sie eine EAI-Bibliothek besitzt. Der Vorteil des EAI liegt darin, daß das mit der VRML-Welt verbundene Programm von dem VRML-Browser unabhängig agieren kann. So kann zum Beispiel nur über das EAI ein Client einer virtuellen Welt realisiert werden. Dabei ist die virtuelle Welt, eine Welt, die von mehreren Betrachtern gleichzeitig benutzbar ist. Mittels eines Servers können dann die verbundenen Benutzer miteinander in Interaktion treten, z.B. Nachrichten austauschen, hier ist das AWT nötig zum Einlesen der Nachricht.

Aber es soll hier mehr auf die Animationen auf dem Bildschirm eingegangen werden, als auf die Interaktionsmöglichkeiten der einzelnen 3D-Graphik-Schnittstellen. Der Script-Knoten bietet im Bereich der Animation genauso viel Möglichkeiten wie das EAI und daher wird nur auf den Script-Knoten näher eingegangen (4. Kapitel, Seite 17). Das nachfolgende Programmbeispiel soll den Umgang von VRML mit einer Programmiersprache verdeutlichen. Es wird in dem Beispiel VRML-Script-Code benutzt, der direkt in der VRML-ASCII-Datei steht.

```

// Definiere einen Ball (default: translation 0 0 0)
DEF Ball Transform { children [ Shape { geometry Sphere { } } ] }
// Definiere einen Zeitgeber
DEF Uhr TimeSensor { // Setze den Zeit-Sensor ... }
// Ein VRML-Script, das das Translationsfeld des Ball-Objektes
// neu berechnet
DEF Springer Script
{
    // 'Reinkommende'-Ereignisse/Variablen
    eventIn SFFloat set_Hoehe
    // 'Rausgehende'-Ereignisse/Variablen
    eventOut SFVec3f Koordinaten_changed
    // Das Script!
    url "vrmlscript: function set_Hoehe( bruchteil, zeitmarke )
    {
        // Ändere den y-Wert des Vektorfeldes
        Koordinaten_changed[0] = 0.0;
        Koordinaten_changed[1] = 4.0 * bruchteil * (1.0 - bruchteil);
        Koordinaten_changed[2] = 0.0;
    }"
}

```

```
// Verbinde die entsprechenden Felder
ROUTE Uhr.fraction_changed TO Springer.set_Hoehe
ROUTE Springer.Koordinaten_changed TO Ball.set_translation
```

Programmbeispiel 2: VRML und VRML-Script

Obwohl VRML eine Beschreibungssprache ist, wird durch den Script-Knoten die Möglichkeit gegeben Programmiersprachenelemente zu benutzen. Allerdings muß hier der Script-Knoten die Aufgabe übernehmen, Variablenwerte zwischen dem VRML-Browser und dem Programmierspracheninterpreter auszutauschen. Der Unterschied zu OpenGL könnte nicht größer sein. VRML ist nicht Bestandteil einer Programmiersprache, die Programmiersprache wird durch eine Schnittstelle zugänglich. OpenGL ist hingegen nichts anderes als eine Ansammlung von Funktionen. Die OpenGL-Funktionen werden wie jede andere Funktion in C aufgerufen. VRML schafft es, obwohl es eine Graphikbeschreibungssprache ist, Interaktionen zu verwirklichen. Der Trick ist der ROUTE-Befehl, der Variablenwertänderungen unter den verschiedenen Knoten zuläßt. Außerdem bietet der Aufbau des Script-Knotens, der nur Variablen und einen Verweis auf Programmcode enthält eine einfache Schnittstelle zu einer Programmiersprache. Denn es werden nur die Variablen zwischen den beiden Prozessen ausgetauscht. Der Script-Knoten wird durch eine Variablenänderung einer seiner eventIn-Variablen aufgerufen. So wird im Programmbeispiel 2 ein Zeitgeber definiert, der den Script-Knoten in regelmäßigen Abständen *aufruft*, indem der Wert seiner set\_Hoehe-Variable neu gesetzt wird. Dann wird in einem unabhängigen Variablenraum die Berechnung durchgeführt. Und zum Schluß der Wert des Vektors des Script-Knotens über den Wert des Translationsfeldes des Ball-Objektes *geschrieben*. Dabei ist das Prinzip dasselbe, ob jetzt der Script-Code direkt in der VRML-Datei steht, oder durch die URL-Angabe eine externe Script-Datei benutzt wird, bzw. eine Java-Klasse aufgerufen wird.

Nachdem gezeigt wurde, mit welchen Mechanismen VRML arbeitet, um mit Programmiersprachen zu kommunizieren, wird jetzt die Art und Weise, wie OpenGL mit C zusammenarbeitet, betrachtet.

### 3.2 Die OpenGL-Bilderzeugungsmechanismen

Da OpenGL durch Funktionsaufrufe gesteuert wird, besteht keine äußere Baumstruktur wie bei Java3D in den retained modi oder VRML. OpenGL kann an einer beliebigen Stelle in einem C-Programm benutzt werden. Die virtuelle OpenGL-Maschine merkt sich alle Aufrufe und erstellt erst dann ein Abbild der 3D-Szene, wenn dies durch den Aufruf der glFinish-Funktion explizit gefordert wird. Interessant ist weiterhin, daß keinerlei Optimierung von Animationen möglich ist, da die virtuelle OpenGL-Maschine bei jedem glFinish-Aufruf ihre Prozeß-Pipeline vom Anfang bis zum Ende durchläuft (siehe Abbildung 4, Seite 14) und danach keine Zwischenwerte mehr zur Verfügung stehen. Nach der Pipelinestruktur, wird die Projektionsmatrix erst in der Mitte der Berechnungen benötigt. Da keine Zwischenwerte von der 3D-Welt über das Ende des glFinish-Aufrufes hinweg gespeichert werden, ist es nicht möglich, erst an diesem Punkt der Pipeline, dem Projektions-Schritt, *einzusteigen*. Statt dessen müssen alle Werte bis zu diesem Punkt neu berechnet werden, auch wenn sich nur die Projektionsmatrix geändert hat, also der Blick auf die 3D-Welt und nicht die 3D-Welt selber. Eine Ausnahme bilden hier die Schritte nach dem Umrechnen der Homogenenkoordinaten (4D) in Leinwandkoordinaten, denn der Frame-Puffer wird über das Ende dieses Aufrufes hinweg gespeichert und kann zu einem beliebigen Zeitpunkt, bis zum nächsten glFinish-Aufruf oder dem Beenden des Programmes, durch entsprechende *Befehle* verändert werden.

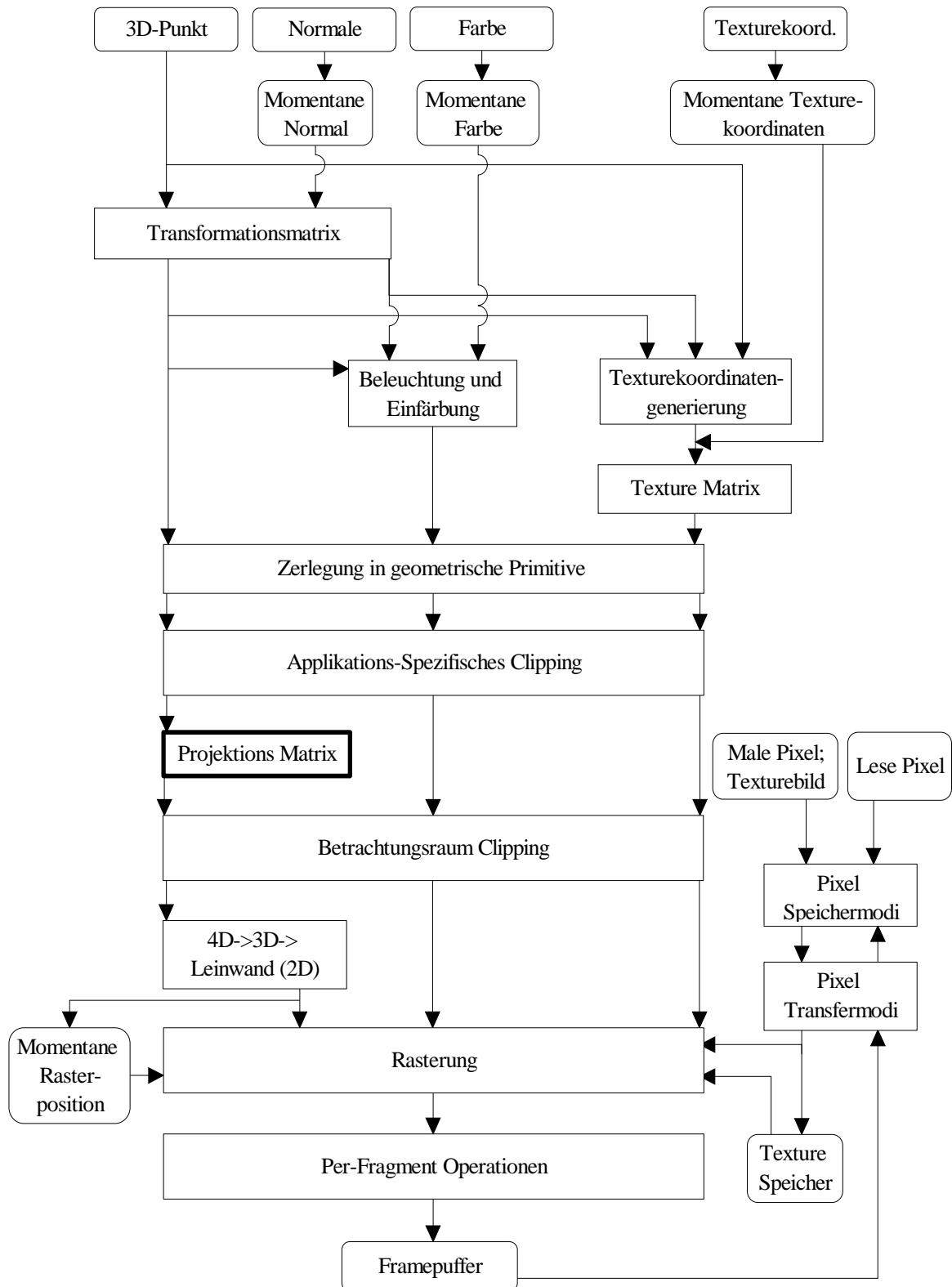


Abbildung 4: Die OpenGL-Pipeline  
(OpenGL 97-1, Seite 9)

Genau genommen ist durch die pixelweise / blockweise Veränderung des Framepuffers die Erstellung von Animation möglich, aber diese Änderungen beziehen sich auf eine 2D-Leinwand. Es ist also keine 3D-Animation möglich, wenn nicht unabhängig von OpenGL 3D-

Objekte in 2D-Objekte abgebildet werden und dann die entsprechenden Änderungen manuell vorgenommen werden. Dieses Vorgehen ist aber absurd, da es keinen Sinn macht, zwei 3D-Rendermaschinen zu benutzen. Hier ist gut ersichtlich, das OpenGL nur für einzelne Bilder konstruiert wurde, es wird keine Möglichkeit geboten, eine Reihe von Frames zu erzeugen. An dieser Stelle soll ein kurzer Programmcodauszug zur Verdeutlichung der Einbettung von OpenGL in die Programmiersprache C dienen.

```
// Die Zeichenfläche und die Puffer löschen ...
// Die folgenden 3D-Objekte um 'AngleZ'-Grad um die Z-Achse drehen
glRotatef(AngleZ, 0.0f, 0.0f, 1.0f);
// Es wird der Winkel bei jedem Durchgang erhöht
AngleZ += 5.0f;
// Einen Würfel definieren ...
// Erstelle Abbild ...
Programmbeispiel 3: OpenGL und C
```

Wichtig bei dem Programmbeispiel 3 ist nur die Zeile, in welcher der Winkel um 5 Grad erhöht wird. Bei OpenGL steht dies einfach mitten zwischen OpenGL-Befehlen, außerdem wird bei einem OpenGL-Befehl eine Variable übergeben. In VRML ist beides undenkbar, weil sie eine Beschreibungssprache ist. Der unterschiedliche Aufbau von OpenGL und VRML ist besonders bei dieser Technik zu beobachten.

Es sei noch bemerkt, daß es bei OpenGL dem Programmierer überlassen ist, mehrere Abbilder hintereinander zu erzeugen, bei Java3D hingegen übernimmt diese Arbeit die API und bietet somit mehr Komfort.

### 3.3 Java3D und Java

Java3D hat die Vorteile der beiden Konzepte, VRML und OpenGL, vereint. So ist es möglich, wie in OpenGL, keine Baumstruktur zu nutzen, sondern den immediate mode. Dadurch existieren in diesem Modus so viele Freiheiten, aber auch genausowenig Optimierungen wie bei OpenGL. Es ist jedoch genauso möglich, eine Baumstruktur zu nutzen, wie in VRML, wobei Java3D zusätzlich den Vorteil hat, eine direkte Einbettung in eine Programmiersprache zu haben, dies soll der folgende Programmcodauszug verdeutlichen.

```
class Test extends Behavior
{
    // Das Aufweckkriterium definieren ...
    // Setze das Aufweckkriterium für das erste Mal ...
    // Überschreibe die processStimulus-Methode, um selber zu Handeln
    public void processStimulus(Enumeration Aufrufkriterium)
    {
        // Die alte Translation lesen, neuen Wert berechnen und
        // neue Translation speichern
        Transform3D t = new Transform3D();
        TransformObj.getTransform(t);
        Vector3f neue_translation = new Vector3f();
        t.get(neue_translation);
        neue_translation.x += 10;
        t.setTranslation(neue_translation);
        TransformObj.setTransform(t);
        // Setze das Aufweckkriterium für das nächste Mal ...
    }
}
```



```

// Der Konstruktor wird mit dem zu verändernden Transfom-Objekt
// aufgerufen, der sich gemerkt wird ...
// Den Raum definieren, in dem der Betrachter sein muß,
// damit dieses Verhalten-Objekt beachtet wird ...
}

```

#### Programmbeispiel 4: Die Verhalten-Klasse von Java3D

Das Programmbeispiel 4 soll den Mechanismus einer Verhalten-Klasse demonstrieren. Es wird die Translation-Variable des zu verändernden Transformation-Objektes verändert, wenn eine Tastaturtaste gedrückt wird. Es wird wie in VRML, dort durch den Script-Knoten, ein separater Bereich geschaffen, um die Berechnung durchzuführen. Der Unterschied zu VRML ist, daß hier innerhalb der Sprache Java getrennt wird, um besser optimieren zu können, und nicht zwei verschiedene Programme, wie z.B. VRML-Browser und Script-Interpreter, miteinander kommunizieren müssen. Des weiteren ist nicht zu vergessen, daß in Java3D die Baumstruktur durch eine Hierarchie von Objekten aufgebaut wird, die wiederum Java-Sprachelemente sind. Es verflechten sich also wie bei OpenGL API und Programmiersprache. Es gibt allerdings auch Punkte in denen sich die Verflechtung von API und Sprache bei Java3D und OpenGL unterscheiden. So ist Java3D an Java gebunden, aber OpenGL kann in fast jeder Sprache benutzt werden.

Wie die Verhalten-Klasse im Detail arbeitet, wird im Kapitel 5.5 (Seite 33) erläutert.

### 3.4 Zusammenfassung

Es hat sich in diesem Kapitel gezeigt, daß die drei Schnittstellen Möglichkeiten bieten, eine Programmiersprache für Berechnungen zu nutzen. Bei OpenGL und Java3D ist dies einfacher als bei VRML, da OpenGL und Java3D Programmiersprachen-’Erweiterungen’ sind und somit direkten Kontakt zur Programmiersprache haben. VRML bietet eine Schnittstelle, die einfach gehalten ist und daher nicht viele Möglichkeiten bietet. Außerdem wurde, in dem die OpenGL-Renderpipeline betrachtet wurde, gezeigt, daß OpenGL kein Animationsmodell besitzt.

## 4. Das Animationsmodell von VRML

In diesem Kapitel soll das Animationsmodell von VRML näher beleuchtet werden. Es wird keine grundlegende Einführung in VRML dargestellt, bei Interesse sollte [Kloss] herangezogen werden. Des weiteren werden die einzelnen Bestandteile des Animationsmodells von VRML erörtert. Damit der Route-Mechanismus verstanden werden kann, wird erst einmal die Art von Variablen, die in VRML existieren, erklärt. Dieses Kapitel schließt mit einer Betrachtung der Optimierungen, die durch das Animationsmodell verfolgt werden.

Das Animationsmodell von VRML ist relativ einfach dazustellen, wie der folgenden Abbildung zu entnehmen ist:

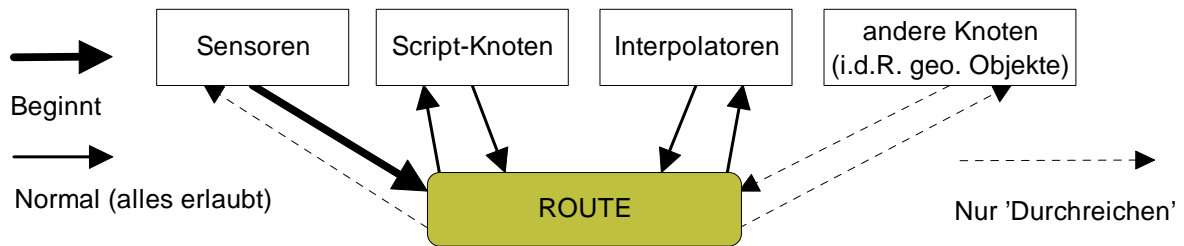


Abbildung 5: Das Animationsmodell von VRML

Es sind also insgesamt 4 Arten von Knoten, die miteinander mittels Route-Befehl verknüpft werden können, dabei kann jeder der 4 Knotenarten Nachrichten empfangen und senden. Es wird beim Senden und Empfangen von Nachrichten noch in Weitersenden, bzw. Durchreichen, nur Empfangen und Erzeugen von Nachrichten unterschieden. Das Erzeugen von Nachrichten ist von 'anderen Knoten' nicht, lediglich das Durchreichen von Nachrichten ist möglich. Außerdem gibt es keine Möglichkeit, daß Sensoren eine Nachricht nur empfangen (siehe Kapitel 4.3, Seite 21). Des weiteren wird eine Kette von Nachrichten in der Regel von einem Sensor-Knoten angestoßen. Dies ist kein Widerspruch dazu, daß alle vier Knotenarten Nachrichten senden können, denn es werden von den Script-Knoten, wie von den Interpolatoren, Nachrichten erzeugt, dies aber in der Regel nur durch das Bearbeiten einer empfangenen Nachricht.

### 4.1 Variablen, Parameter, Felder und Nachrichten

Es gibt zu den verschiedenen Knotentypen eine Reihe von Feldern. Diese Felder sind vorgegeben, eine Ausnahme bildet der Script-Knoten (siehe Kapitel 4.5, Seite 22). Die Felder können als Variablen angesehen werden, wenn sie veränderbar sind. Andererseits sind die Feldwerte im Grunde Parameter einer Funktion, wenn der Knoten dabei als Funktionsaufruf angesehen wird. Jedes Feld hat einen Feldtyp, einen Variablentyp, einen Parameternamen, bzw. Variablennamen und den Parameterwert, bzw. Variablenwert. Wenn Felder einen Defaultparameterwert (siehe [VRML 97-1, S.67-131]) haben, dann sind sie schon definiert und es darf nicht noch einmal ein Feld- oder Variablentyp angegeben werden, wenn ihr Wert neu gesetzt wird - genauso wie in den meisten Programmiersprachen. Zusätzlich zu dem Variablentyp wird ein Feldtyp benötigt, um Felder nach außen zu kennzeichnen. Angaben zu Außen und Innen beziehen sich auf die Sicht von dem aktuell betrachteten VRML-Knoten. Es gibt die Möglichkeit, daß der Feldinhalt von innen änderbar ('field'), von außen änderbar ('eventIn'), von innen änderbar und nach außen versendbar ('eventOut'), sowie von außen änderbar und nach außen versendbar ('exposedField') ist. Diese Angaben werden von dem Route-Mechanismus benötigt, um die Berechtigung der Zugriffe zu klären. Es geht hier wirklich nur um die Berechtigungen, denn alle Variablen sind global zugänglich, sobald der entsprechende Knoten mittels des Def-Befehles einen Namen zur Identifikation bekommen hat.

Das Animationsmodell von VRML ist, wie oben gesagt, auf dem Prinzip aufgebaut, Nachrichten zu versenden, dabei bestehen Nachrichten aus einem Variablenwert und einer Zeitmarke. Die Zeitmarke wird pro Nachrichtenkette konstant gehalten, wobei der interne Zeitgeber aber weiter läuft. Die Zeitmarke enthält die Angabe über die Sekunden die seit 1.1.1970 0:00 vergangen sind. Eine Nachrichtenkette ist eine Kette von Nachrichten, bei der die erste Nachricht von außen, z.B. Zeitgeber oder Benutzer, angestoßen wird und die restlichen Nachrichten nur Umformungen und/oder Vermehrungen der ersten sind. Für das Austauschen von Nachrichten zwischen den einzelnen Knoten ist der Route-Mechanismus zuständig.

## 4.2 Der Route-Mechanismus

Der Route-Befehl hat die folgende Syntax:

```
ROUTE QuellKnoten.QuellVariable TO ZielKnoten.ZielVariable
```

Bei dem Route-Befehl ist zu beachten, daß die Quell- und Zielvariablen denselben Variablentyp haben. Außerdem müssen den Quell- und Ziel-Knoten mit dem Def-Befehl Namen zugeordnet werden (siehe Programmbeispiel 2, Seite 13), damit sie eindeutig identifiziert werden können. Des weiteren müssen Quellvariablen von dem Feldtyp eventOut oder exposedField sein, und die Zielvariablen von dem Feldtyp eventIn oder exposedField. Als Namenskonvention sollten Namen von Zielvariablen mit einem 'set\_' beginnen und die Namen von Quellvariablen mit einem '\_changed' enden. Zum Beispiel sollte, wenn die Variable 'fraction' von dem Zeit-Sensor als Quellvariable benutzt wird, der Name 'fraction\_changed' benutzt werden.

Damit der Browser nicht in eine Endlosschleife bei einer Nachrichtenkettenabarbeitung geraten kann, muß sichergestellt werden, daß eine Nachrichtenkette auch endet. Es könnten theoretisch Endlosschleifen mit dem Route-Befehl erstellt werden, indem eine Zielvariable wieder auf die Quellvariable verweist. Aber die VRML-Browser sind angewiesen, pro Nachrichtenkette jede eventOut- / exposedField-Variable nur einmal ein Ereignis senden zu lassen. Dadurch sind in der Praxis keine Endlosschleifen möglich (siehe Erklärung zu Abbildung 8, Seite 20) [VRML 97-1, S. 47-51].

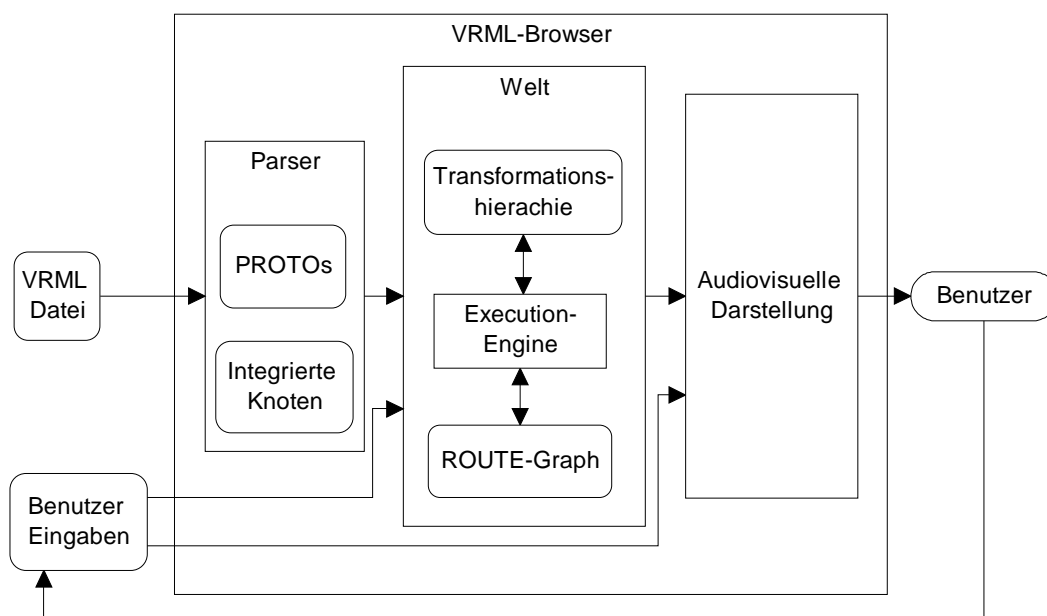


Abbildung 6: Die VRML-Umgebung  
(Übernommen aus VRML 97-1, Seite 23)

Wie aus der vorangehenden Abbildung 6 zu sehen ist, befindet sich die Eingabe für den VRML-Browser, die VRML-Datei und die Benutzer-Eingaben, auf der rechten Seite. Die VRML-Datei wird geparkt und in die interne Struktur umgewandelt, dabei werden die integrierten Knoten, wie 'Sphere', und die Prototypen zur Auflösung der VRML-Befehle herangezogen. Der Transformationsbaum und der Route-Graph sind die Ergebnisse des Parsers. Der Transformationsbaum wird herangezogen um die Audiovisuelle Darstellung zu berechnen. Diese wirkt auf den Benutzer und der kann dann durch Eingaben entweder in der 3D-Welt navigieren (der Pfeil zu der Audiovisuellen Darstellung) oder einen Sensorwert indirekt, z.B. durch das Anklicken eines Objektes, verändern (der Pfeil zur 'Welt'). Wenn ein Sensor in Aktion tritt, dann leitet die Execution-Engine die Nachrichten weiter, indem sie nach Verbindungen von der entsprechenden Sensor-Variablen zu anderen Variablen im Route-Graph sucht. Durch das Versenden von Nachrichten ändern sich Variablen in der Transformationshierarchie. Innerhalb von Script-Knoten ist auch die Veränderung des Route-Graphen möglich (siehe Kapitel 4.5, Seite 22).

Um den Route-Mechanismus genauer zu verstehen, ist ein Vergleich mit dem message passing-Mechanismus (=Botschaftentransport) von Betriebssystemen geeignet. Das message passing handelt nach dem Prinzip, Nachrichten zwischen Prozessen auszutauschen. Es wird dabei die Nachricht und das Ziel / die Quelle angegeben. Bei dem Route-Mechanismus ist das etwas anders, denn:

1. Prozesse in VRML sind keine Prozesse, wie sie bei Betriebssystemen auftauchen und
2. die Ziele und Quellen werden über einen Route-Graphen bestimmt, das heißt, der laufende Prozeß muß bei dem Versenden von Nachrichten nicht das Ziel angeben.

zu 1.: Die Prozesse unter VRML werden nicht von der VRML-Execution-Engine (siehe Abbildung 6, Seite 18) unterbrochen, sie werden in einen Bereitschaft-Modus gesetzt, wenn sie eine einkommende Nachricht bearbeitet haben. Es kann dabei jeder Script-, Sensor- und Interpolator-Knoten als ein Prozeß betrachtet werden. Prozesse werden bei dem in den Speicher Laden des entsprechenden Knotens erzeugt und beim Entfernen des entsprechenden Knotens beendet. Die Prozesse laufen also nacheinander und es herrscht kein Mehrprogramme-Betrieb. Es fällt daher der Schedule-Algorithmus relativ einfach aus, da immer der Prozeß in den Ausführungsmodus gesetzt wird, der als erster eine Nachricht bekommen hat.

Da die VRML-Execution-Engine ein abgeschlossenes System ist, kann immer entschieden werden, welcher Prozeß als erstes eine Nachricht bekommen hat. Es können nicht physisch gleichzeitig mehrere Nachricht bei einen Prozeß ankommen. Es kann aber logisch gleichzeitig eine Nachricht an mehrere Prozesse gesendet werden. Das Empfangen von Nachrichten ist nicht physisch zeitgleich mit dem Senden der selben. Auch könnte das logisch gleichzeitige Senden von Nachrichten nicht bewerkstelligt werden, wenn Nachrichten von der Execution-Engine nicht zwischengespeichert werden würden. Es wird von der Execution-Engine für jede Zielvariable im Route-Graph (siehe zu 2., Seite 20) eine Warteschlange gehalten, die sich die Nachrichten an diese Zielvariable merkt und wenn der Prozeß aktiv wird, dann wird diese Warteschlange in chronologischer Reihenfolge abgearbeitet. Dieser Mechanismus des Zwischenspeicherns von Ereignissen wird auch für den Fall benötigt, bei dem die Sensoren mit Werten von außen, z.B. durch ein Eingabegeräte oder den Zeitgeber, verändert werden. Denn der VRML-Browser ist in der Regel mit dem Darstellen der 3D-Welt oder die Execution-Engine mit dem Abarbeiten einer Nachrichtenkette beschäftigt, so daß die Nachrichten von außen nicht sofort bearbeitet werden können. Da über kein unsicheres Medium Interprozeß-Kommunikation betrieben wird, ist auch kein Bestätigungsmechanismus für das Ankommen von Nachrichten notwendig. Auch existiert keine Prozeßhierarchie, da keine Kindprozesse erzeugt werden können. Andererseits ist der Route-Mechanismus ein Monolithisches-System,

in dem der Kernel-Modus (=Nachrichten weiterleiten) durch den Benutzer-Modus (=die verschiedenen Prozesse) aufgerufen wird (siehe Abbildung 7).

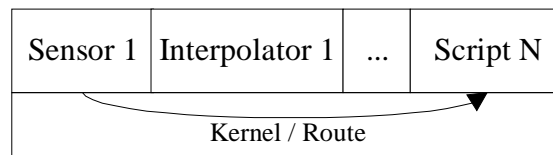


Abbildung 7: Der Route-Mechanismus als Kernel dargestellt  
(Abwandlung von [Tannenbaum, Seite 35])

zu 2.: Nach dem Parsen eines Route-Befehls wird in dem Route-Graph eine Verbindungen zwischen den entsprechenden Variablen erzeugt. Wenn eine oder beide der Variablen bisher noch nicht in dem Route-Graph vorkommt, dann wird sie als ein neuer Route-Graph-Knoten eingefügt. Dieser Graph ist nicht zwingend zusammenhängend oder zyklenfrei, aber gerichtet und ungewichtet (siehe Abbildung 8).

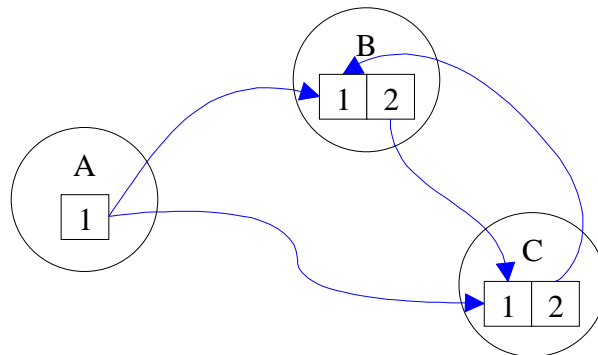


Abbildung 8: Ein Route-Graph-Beispiel

Den Route-Graph aus Abbildung 8 würden die folgenden Route-Befehle erzeugen:

```
ROUTE A.1 TO B.1
ROUTE B.2 TO C.1
ROUTE C.2 TO B.1
ROUTE A.1 TO C.1
```

Es wird von A.1 logisch gleichzeitig an B.1 und C.1 dieselbe Nachricht los geschickt. Intern bekommt B.1 vor C.1 die Nachricht, da der Route-Befehl für die A.1-B.1-Verbindung vor dem Route-Befehl für die A.1-C.1-Verbindung steht. Es wird dann von B eine Nachricht erzeugt, die von B.2 an C.1 gesendet wird. Dann wird von C eine Nachricht erzeugt, die von C.2 an B.1 geschickt wird. B verarbeitet die Nachricht, es wird aber keine Nachricht an C.1 weitergegeben. Jetzt erst wird die Nachricht von A.1 an C.1 weitergegeben, die zwar noch von C verarbeitet wird, aber auch hier wird keine Nachricht an B.1 weitergegeben. Danach haben alle Route-Graph-Knoten einmal eine Nachricht geschickt und somit ist die Nachrichtenkette beendet. Es entsteht also keine Endlosschleife zwischen B und C. Die Abarbeitungsstrategie ist leider VRML-Browser abhängig. Es kann sein, daß Nachrichten, die nicht zur Weiterberechnung benötigt werden, erst erstellt werden, wenn alle Nachrichten abgearbeitet worden sind. Oder aber die Nachrichten werden sofort erzeugt, oder nachdem die Funktion ausgeführt wurde. Das Problem, daß es in der Praxis noch kein einheitliches Konzept gibt, kann sich bei komplexen VRML-Welten in der Weise auswirken, daß verschiedene VRML-Browser verschiedene Nachrichtenketten erzeugen und somit verschiedene Nachrichten gesendet werden können.

Um diese Manko zu beheben, hat sich eine offizielle VRML-Arbeitsgruppe gebildet, es ist dies die 'Conformance Working Group' [VRML 98-1].

Nachdem jetzt geklärt ist, wie Nachrichten ausgetauscht werden, ist es sinnvoll zu sehen, wie eine Nachrichtenkette erzeugt wird.

### 4.3 Sensoren

Die Hauptaufgabe der Sensoren besteht darin, eine Nachrichtenkette einzuleiten. Wenn also eine vorgegebene Zeit verstrichen ist (Zeit-Sensor), oder der Benutzer eine abfragbare Eingabe (siehe unten) macht, dann ändert der entsprechende Sensor-Knoten seine Variablenwerte, sobald die Execution-Engine von VRML wieder aktiviert wird. Wenn die Sensor-Variable mittels des Route-Befehls mit anderen Variablen von anderen Knoten verknüpft ist, dann wird der selbe Wert, der jetzt in der entsprechenden Sensor-Variable steht, in die entsprechende Ziel-Knoten-Variable geschrieben. Um den Aufbau der Sensor-Knoten zu verdeutlichen, wird der Anklick-Sensor genauer betrachtet.

#### TouchSensor

```
{
  exposedField SFBool   enabled
  eventOut      SFBool   isActive
  eventOut      SFBool   isOver
  eventOut      SFVec3f   hitNormal_changed
  eventOut      SFVec3f   hitPoint_changed
  eventOut      SFVec2f   hitTexCoord_changed
  eventOut      SFTIME    touchTime
}
```

Der Defaultwert für die enabled-Variable ist 'TRUE'. Daß heißt, wenn ein Anklick-Sensor erstellt wird, dann ist er automatisch an. Wenn das zu dem Sensor gehörige Objekt angeklickt wird, dann wird die isActive-Variable 'TRUE'. Ähnliches bewirkt die isOver-Variable, denn sie gibt an, ob der Mauszeiger gerade auf ein zu dem Sensor gehöriges 3D-Objekt zeigt. Die nächsten drei Variablen geben die Vektoren an, die sich aus Mauszeigerposition und 3D-Objekt-Oberfläche ergeben und sind für den etwas komplexeren Einsatz dieses Sensors geeignet. Interessant ist meist noch der Zeitpunkt, wann das 3D-Objekt angeklickt wird, dafür ist die touchTime-Variable zuständig. Wichtig sind hier auch die Variablentypen, denn wenn zur Weiterverarbeitung eine Zeit-Variable gebraucht wird, dann muß mittels Route-Befehl die Quell- und Ziel-Zeit-Variable verknüpft werden. An den Feldtypen ist zu erkennen, daß es hauptsächlich eventOut- und einige exposedField-Variablen gibt, es gibt aber keine eventIn-Variablen. Es kann also die Einstellung eines Sensor nur eingeschränkt mittels des Route-Mechanismus verändert werden. Wie generell mittels des Use-Befehles Variablen anderer Knoten verändert werden können, ist im Kapitel 4.5 (Seite 22) erklärt.

Abgesehen von dem Zeit-Sensor sind alle Sensoren abhängig von Benutzereingaben und wie diese Eingaben im Zusammenhang mit der VRML-Welt steht, ist in Abbildung 6 (Seite 18) zu sehen. Welche Benutzereingaben es gibt, ist dem Kapitel 2.3 (Seite 9) zu entnehmen.

Manchmal ist es auch sehr interessant, daß sich Objekte drehen, bewegen und ähnliches, nach vorgegebenem linearen Muster. Für solche Zwecke ist eine Kombination von einem Zeit-Sensor und einem Interpolator geeignet.

### 4.4 Interpolatoren

Da der Zeit-Sensor nur Zeitwerte liefert, aber in der Regel Rotations-, Skalierungs- oder Translationswerte benötigt werden, müssen diese noch umgerechnet werden, dafür sind die Interpo-

latores da. Welche Interpolatoren existieren, ist dem Kapitel 2.3 (Seite 9) zu entnehmen. Ein Interpolator rechnet also die Zeitwerte in andere Werte um. Die Art der neu berechneten Werte hängt von der Art des Interpolator ab, so wandelt ein Rotations-Interpolator Zeitwerte in Rotationswerte um. Dafür wird eine lineare Interpolation benutzt. Das heißt, es werden die Anfangs- und Endwerte der neu zu berechnenden Werte angegeben und dann linear interpoliert in Abhängigkeit von dem `fraction_changed`-Variablenwert des Zeit-Sensors. Dabei wird der `fraction_changed`-Variablenwert immer neu gesetzt, nachdem ein Abbild der 3D-Welt erstellt wurde. Auch hier soll, wie bei den Sensoren, ein Interpolator etwas genauer betrachtet werden, es ist dies der Koordinaten-Interpolator.

#### CoordinateInterpolator

```
{
  eventIn      SFFloat  set_fraction
  exposedField MFFloat  key
  exposedField MFVec3f  keyValue
  eventOut     MFVec3f  value_changed
}
```

Die erste Variable trägt den Namen `set_fraction` und ist das Gegenstück zu der `fraction_changed`-Variable des Zeit-Sensors. Die `key`- und `keyValue`-Variablen geben die Bruchteil- und die zu interpolierenden Wert-Intervalle an. Die Variablenwerte `key = [0, 0.5, 1]` und `keyValue = [5 0 0, 8 0 0, 9 0 0]` würden angeben, daß die Werte in dem Bruchteil-Intervall `[0, 0.5]` in ein Wert-Intervall von `[5 0 0, 8 0 0]` linear abgebildet werden würden, und von `[0.5, 1]` auf `[8 0 0, 9 0 0]`. Also ist in der ersten Zeithälfte die Steigung stärker als in der zweiten. Die letzte Variable (`'value_changed'`) gibt dann die aus dem Bruchteil berechnete Koordinate an. Wenn eine lineare Interpolation nicht mehr ausreicht, müssen komplexere Mittel her, dafür existiert der Script-Knoten.

### 4.5 Der Script-Knoten

Der Script-Knoten ist sehr umfangreich in seinen Möglichkeiten, es soll daher erst einmal auf seine äußere Struktur eingegangen werden. Das Gerüst eines Script-Knotens besteht aus einem Interface-Teil, in dem Variablen, und einem Script-Teil, in dem der Programmcode steht. Die Variablen, die im Interface-Teil stehen, sind dem Programmcode zugänglich und können von ihm geändert werden. Es gibt jetzt zwei Möglichkeiten, wie Variablen, die nicht im eigenen Script-Knoten sind, erreichbar sind: Die erste ist die schon Bekannte mittels des Route-Befehls. Die zweite ist im Grunde auch eine VRML-Standard-Methode und zwar kann mittels `'USE'` ein Knoten als Variable angegeben werden und dann durch diese Variable auf die Knoten-Variablen des anderen Knotens zugegriffen werden. Als Beispiel wäre im Variablenteil die folgende Zeile nötig:

```
field SFNode baum USE Baum
```

Und im Script-Teil dann ein folgender Zuweisung gültig:

```
baum.translation[0]=baum.translation[0]+5;
```

Es würde also in dem Transformation-Knoten, der den Namen `Baum` zugewiesen bekommen hat, das Translationsfeld an der X-Position um 5 erhöht. Es kann durch diese Methode eine Knoten-Variable von zwei unterschiedlichen Script-Knoten innerhalb einer Nachrichtenkette verändert werden, was sonst der Route-Mechanismus verbieten würde, damit keine Endlosschleife in der Nachrichtenkette entsteht. Dies liegt daran, daß der Route-Mechanismus mittels

des Use-Befehls umgangen wird, und somit keine Nachricht gesendet, sondern nur die Zielvariable geändert wird.

Um die Variablen des Script-Knotens genauer zu überschauen, hier eine Auflistung:

```
Script
{
  exposedField    MFString    url []
  field           SFBool      directOutput FALSE
  field           SFBool      mustEvaluate FALSE
  # Eine beliebige Anzahl von:
  eventIn         eventType    eventName
  field           fieldType    fieldName initialValue
  eventOut        eventType    eventName
}
```

Die erste Variable ('url') ist schon in früheren Kapiteln aufgetaucht und enthält entweder eine Internetadresse, an der eine Java-Klassen- oder Script-Datei liegt, oder haben einen String, der das Script enthält. Die Variablen directOutput und mustEvaluate verändern die Art wie der Script-Knoten mit dem Route-Mechanismus interagieren soll. Dabei bewirkt die directOutput-Variable, daß der Abkapselung-Mechanismus des Script-Knotens zu der VRML-Browser aufgehoben wird. Die Zugriffsmöglichkeiten beschränken sich dann nicht mehr nur auf den Interface-Teil. Es können so auch neue Route-Verbindungen erstellt werden (siehe unten). Außerdem muß diese Variable auf 'TRUE' gesetzt werden, wenn mittels des Use-Mechanismus Variablen anderer Knoten verändert werden sollen. Es wird der Route-Mechanismus erst umgangen, wenn diese Variable auf TRUE gesetzt ist. Wenn die mustEvaluate-Variable gesetzt ist, bedeutet dies, daß alle ankommenden Nachrichten sofort zur Ausführung der entsprechenden Funktion führen. Dieses Vorgehen ist bei den Scripten sinnvoll, die Berechnungen durchführen, die notwendig sind, damit die 3D-Welt ihre Konsistenz behält. Der Kollisions-Sensor in einer 3D-Welt ist z.B. sinnlos, wenn nicht sofort bei einer Kollision des Betrachters mit einem tödlichen Objekt gehandelt wird. Diese Variable sollte nicht gesetzt sein, wenn es sich um eine komplexere 3D-Welt handelt, da dort der Browser schon vom Abbild Erstellen ausgelastet ist. Da im vorhinein nicht bekannt ist, auf welchen Rechner die VRML-Welt dargestellt wird, sollte der Browser entscheiden dürfen, ob er nicht einige der eintreffenden Nachrichten erst einmal sammeln darf und später ausführt, wenn wichtigere Sachen erledigt sind.

Sonst gibt es keine vordefinierten Variablen, es können beliebig viele Variablen der Typen eventIn, eventOut oder field erstellt werden. Es dürfen aber keine exposedField-Variablen erstellt werden, da die Funktionen keine Rückgabewerte haben dürfen. Diese Tatsache leuchtet ein, wenn berücksichtigt wird, daß Variablenänderungen eines Script-Knotens, den Aufruf der gleichnamigen Funktion des selben Script-Knotens bewirkt.

Der Script-Teil des Script-Knotens kann auf verschiedene Sprachen zurückgreifen (s.o.). Da es nicht Sinn dieser Arbeit ist, den Unterschied der verschiedenen Sprachen und verschiedenen Browser aufzuzeigen, wird in der Regel das Verhalten des Cosmoplayer 2.1 und Javascript beschrieben. Es ist anzumerken, daß es kleine Unterschiede in der Namensgebung der Funktionen gibt und das die Nachrichtenweiterleitung von der beschriebenen im Detail abweichen kann, das Konzept aber das gleiche ist (siehe Anmerkung zu Abbildung 8, Seite 20).

Der Script-Teil des Script-Knotens besteht aus einer Reihe von Funktionen, für jede eventIn-Variable eine Funktion, die genau den selben Namen wie die Variable hat. Dabei haben diese Funktionen immer zwei Parameter, den Nachrichtwert, der denselben Variablentyp hat wie die eventIn-Variable, und die Zeitmarke. Außerdem können noch Funktionen angegeben werden, die aufgerufen werden, wenn der Script-Knoten in den Speicher geladen wird, der



Script-Knoten aus dem Speicher gelöscht wird und wenn eine Reihe von (zwischengespeicherten) Nachrichten an den selben Script-Knoten abgearbeitet worden ist (siehe Kapitel 4.2, zu 1., Seite 19). Die letztere Funktion kann genutzt werden, um eine bestimmte Nachricht besonders zu berücksichtigen, oder damit der Löwenanteil einer Rechnung nur einmal gemacht werden muß, wenn nur ein Nachrichtenwert berücksichtigt werden muß [Ames, S. 571-585]. Diese Funktion ist das Gegenstück zur `mustEvaluate-Variable`, die bei jeder Variablenänderung den Aufruf der Script-Funktion erzwingt. Sinnvoll ist die Funktion bei einem selbst geschriebenen Interpolator einzusetzen. Anstatt alle Zwischenwerte zu berechnen und dadurch eine langsam ablaufende Interpolation zu erzeugen, werden einfach ein paar Interpolationsschritte übersprungen (Näheres bei VRML 97-1, Kapitel 4.10.3).

Der Script-Teil bietet vor allem die Möglichkeit, Variablen anderer Knoten durch komplexe Eingriffe zu verändern. So kann durch den `addChilden`-Befehl sogar die mehrdimensionale `children-Variable` um einen weiteren Knoten ergänzt werden oder ein bestimmter Knoten in einem Switch-Knoten aktiviert werden. Die Script-Sprache kann in dem Script-Teil zwar genutzt werden, es ist allerdings nur der Sprachkern vorhanden; also Variablenbehandlung, Schleifen, Bedingungen usw. Aber es sind die meisten (vertrauten) Objekte, wie das Document-Objekt, nicht vorhanden. Stattdessen werden sogenannte VRML-Objekte bereitgestellt, welche die Variablentypen von VRML und die sprachspezifischen Variablentypen ineinander umwandeln. Javascript wurde weiterhin um das Browser-Objekt (siehe Tabelle 2) bereichert, das Methoden bereitstellt, die Script-Knoten-Interface übergreifend sind. Es kann mit diesem Objekt dann sogar eine ganz neue VRML-Welt geladen werden. Weiterhin gibt es auch ein Objekt, das die Matrizen- und Vektorrechnungen vereinfacht [VRML 97-1, S. 185-205]. Es folgt eine Übersicht über die Funktionen des wichtigen Browser-Objektes.

| Rückgabetyp | Funktion   | Beschreibung  |
|-------------|--|---|
| SFString    | <code>getName()</code>                                 | des VRML-Browsers   |
| SFString    | <code>getVersion()</code>                              | des VRML-Browsers   |
| SFFloat     | <code>getCurrentSpeed()</code>                         | des Betrachters (m/sec)                                       |
| SFFloat     | <code>getCurrentFrameRate()</code>                     | Bilder/sec  |
| SFString    | <code>getWorldURL()</code>                             | Wo ist diese Datei her  |
| void        | <code>replaceWorld(MFNode Knoten)</code>               | Ersetze momentane VRML-Welt durch angegebene                  |
| void        | <code>loadURL(MFString url, MFString Parameter)</code> | Lade VRML-Welt mit Hypertextreferenz-Parametern (z.B. target) |
| void        | <code>setDescription(SFString Beschreibung)</code>     | Der VRML-Welt einen Namen geben                               |
| MFNode      | <code>createVrmlFromString(SFString vrmlSyntax)</code> | Erstelle einen Knoten aus einem String                        |

Tabelle 2: Methoden der Browser-Klasse von Javascript

| Rückgabetyp | Funktion   | Beschreibung  |
|-------------|--|---|
| void        | createVrmlFromURL(MFString url, SFNode Knoten, SFString event)                           | Lädt eine VRML-Welt und gibt sie an das Event-Feld des angegebenen Knotens weiter |
| void        | addRoute(SFNode vonKnoten, SFString vonEventOut, SFNode zuKnoten, SFString zuEventIn)    | Wie der Route-Befehl  |
| void        | deleteRoute(SFNode vonKnoten, SFString vonEventOut, SFNode zuKnoten, SFString zuEventIn) | Lösche Route-Verbindungen   |

Tabelle 2: Methoden der Browser-Klasse von Javascript

#### 4.6 Optimierungen durch das Animationsmodell

Um die Frage nach den Optimierungen bei VRML zu beantworten, muß die Frage erst einmal in drei Bereiche aufgeteilt werden. Der erste Bereich enthält die Optimierungen bei den Berechnungen der Abbilder, der zweite Bereich das Vorhandensein dynamischer Systeme, bei der Erstellung eine Reihe von Bilder und der dritte Bereich befaßt sich mit der Übersetzung der Sprache in ein 'maschinen-gerechteres'-Format.

Der erste Bereich ist leider mit einem 'Jein' zu beantworten. Das liegt daran, daß einige VRML-Browser in OpenGL geschrieben sind, z.B. der Cosmoplayer 2.0 von SGI für Win32-Systeme [Release Anmerkungen im Programm]. OpenGL hat aber keine Optimierungen in diesem Bereich. Es existieren aber trotzdem Optimierungen in diesem Bereich, sie werden durch Begrenzungsboxen realisiert, die das Culling (=aussortieren) von nicht sichtbaren Objekten übernehmen. Diese Begrenzungsboxen müssen allerdings vom VRML-Welt-Ersteller angegeben werden [VRML 97-1, Kapitel 4.6.4]. Das heißt, bei einer Abbilderstellung, wird der Berechnungsaufwand hierfür verkleinert. Eine andere Optimierung in diesem Bereich ist der LoD-Knoten. Die Optimierung ist sehr rechenerleichternd und einfach zu bewerkstelligen. Es werden bei weiten Entfernungen erst gar nicht die komplexeren Objekte zur Abbildberechnung verwendet, sondern vereinfachte Objekte, die der Programmierer angeben muß.

Der zweite Bereich beinhaltet bei VRML, daß der Browser entscheidet, wann er 'Zeit' hat, um die Script-Knoten zu verarbeiten (siehe 'mustEvaluate', Seite 23). Es wird an dieser Stelle mehr Wert darauf gelegt, eine höhere Abbildrate zu haben, als die Benutzeraktivitäten sofort zu berücksichtigen. Eine andere Optimierung ist die klare Abtrennung von statischen, also geometrische Daten, und dynamischen Bereichen, also Sensoren, Interpolatoren und Script-Knoten (siehe 'directOutput', Seite 23 & Abbildung 6, Seite 18), so daß die dynamischen die statischen Bereiche nur in minimaler Weise beeinflussen.

Es wird in dem dritten Bereich, der Optimierung des Laufzeitverhaltens der VRML-Sprache selber, durch eine offizielle Arbeitsgruppe von VRML geforscht. Es ist dies die Arbeitsgruppe, die ein komprimiertes binäres Dateiformat erzeugen möchte [VRML 98-1]. Wenn dieses Format existiert, ist der Effekt, daß das Laden von VRML-Dateien, wie die Umsetzung in eine interne Struktur, schneller ist. Der Effekt breitet sich somit auch zur Laufzeit aus. Denn bei größeren VRML-Welten werden, in der Regel während des Betrachtens der 3D-Welt, weitere Teile der VRML-Welt nachgeladen, wenn sie benötigt werden. So ist z.B. bei einer 3D-Welt die aus zwei Räumen besteht, der zweite erst sichtbar, wenn der erste verlassen wird.

#### 4.7 Zusammenfassung

Nachdem das Animationsmodell von VRML als ganzes dargestellt wurde und die Teilbereiche: Route-Mechanismus, Sensoren, Interpolatoren und Script-Knoten im Detail besprochen wurden, ist ersichtlich, daß eine gute Trennung zwischen statischen und dynamischen Bereichen der 3D-Welt geschaffen wurde, die zu (angesprochenen) Optimierungen führt. Dabei ist das Verfahren eine Animation zu erzeugen, grob in das folgende Schema zu fassen.

Sensoren ändern ihre Variablenwerte, diese Änderung wird synchron bei anderen Variablen vorgenommen. Jetzt können durch vorgegebene Interpolatoren und / oder selbstgeschriebene Script-Knoten die Variablenwerte in andere umgewandelt werden. Es werden dann die Variablen des Transformationshierarchiebaumes mit den berechneten Werten synchronisiert. Dadurch entstehen dann die Animationen.

Weiterhin hat sich gezeigt, daß sich Gedanken zur Renderberechnungserleichterung gemacht wurden und ihre Umsetzung Performance-Vorteile bringt.

## 5. Das Animationsmodell von Java3D

Dieses Kapitel widmet sich der Aufgabe, das Animationsmodell von Java3D im Detail zu untersuchen. Es wird unter anderem auf die verschiedenen Rendermodi eingegangen und das Animationsmodell als Ganzes dargestellt. Des weiteren werden die Bestandteile des Modells genauer besprochen. Den Abschluß bildet, wie im entsprechenden Unterkapitel über das VRML-Animationsmodell, eine Betrachtung der Optimierungen, die mit dem Modell geschaffen werden.

Für eine Einführung in die Programmierung von Java3D existieren im WWW verschiedenste Tutorial (siehe [Java3D-Repository]), die aber alle kein Lehrbuch ersetzen. Durch die Neuheit des Produktes existiert auch noch kein gutes Lehrbuch.

### 5.1 Rendermodi

Auf abstrakter Ebene ist das VRML-Animationsmodell einfach, aber dafür hat es im Detail kompliziertere Regelungen. Bei dem Java3D-Animationsmodell ist es anders, das Modell ist etwas komplizierter, hält dafür aber seine Struktur konsequent durch. Daß das Java3D-Animationsmodell komplexer ist, zeigt schon die Tatsache, daß Java3D drei Rendermodi hat, den immediate, den retained und den compiled-retained mode (vgl. Kapitel 2.2, Seite 6). Der immediate mode kann dabei noch mit den beiden anderen Modi kombiniert werden. Wie das im Einzelnen aussieht, ist an dieser Stelle nicht von Bedeutung und kann im Kapitel 13.1.2 der Java3D-Spezifikation [Sun 98-1] nachgelesen werden. Der immediate mode muß also wegen der Kombinierbarkeit eine minimale Struktur haben. Die Strukturvorgabe besteht aus dem Hierarchiebaumteil des retained mode, an dem das Sicht-Objekt hängt (vgl. Abbildung 2, Seite 6). Es existiert somit auch im immediate mode ein VirtuallUniversum-Objekt, an dem ein Lokales-Objekt hängt usw., bis hin zum Sicht-Objekt und dort hängen dann das Leinwand-Objekt und das Bildschirm-Objekt dran. Da im immediate mode auch geometrische Objekte in Programmiersprachen Objekten angegeben werden müssen und Transformationen durch das Überordnen eines Transformation-Objektes über die geometrischen Objekte erfolgen, ist der Unterschied der drei Modi nicht mehr so groß. In den retained modi herrscht eine strengere Hierarchie. Der Vorteil dabei ist, daß Änderungen, die in einer hierarchisch höheren Position gemacht werden, eindeutige Auswirkungen auf hierarchisch niedrigere Positionen haben, wenn diese im selben Teilbaum der Hierarchie sind. Der Nachteil ist, daß die Hierarchie und deren Regeln eingehalten werden müssen. In den retained modi ist es trotz Hierarchie möglich, daß ein Objekt mehrere direkt übergeordnete Objekte hat. Dies geschieht, indem der Share-Mechanismus genutzt wird (siehe Sun 98-1, Kapitel 6.1). Der Unterschied zwischen den beiden retained modi ist im wesentlichen, daß beim compiled-retained mode ein noch starreres System herrscht. Es muß in diesem Modus sogar die Erlaubnis für Änderungen zur Laufzeit explizit gegeben werden. So gibt es für jede Java3D-Klasse Konstanten (siehe Sun 98-1, Kapitel 4-5), die der setCapability-Methode (=Fähigkeiten setzen) des eigenen Objektes übergeben werden können, so daß dann das Lesen oder Schreiben von einer Variablen zur Laufzeit erlaubt ist. Das heißt, wenn zum Beispiel die Fähigkeit, daß ein Wert des Objektes gelesen werden darf, nicht gesetzt wird, dann bewirkt schon das Lesen einer Variable dieses Objektes, daß eine Exception (=Ausnahme) auftritt.

Obwohl es diese drei Rendermodi gibt, ist es den Java3D-Spezifizierenden gelungen, ein einheitliches Animationsmodell zu schaffen.

Wenn bei Java3D Animation ins Spiel kommt, die nicht von der Programmiersprache Java selbst kommt, dann wird sie entweder durch Interpolatoren oder Verhalten-Objekte umgesetzt. Diese zwei Arten werden ihrerseits von dem Schedule-Mechanismus angestoßen. Den Zusammenhang zwischen allen Animationsmechanismen von Java3D stellt die Abbildung 9 da.

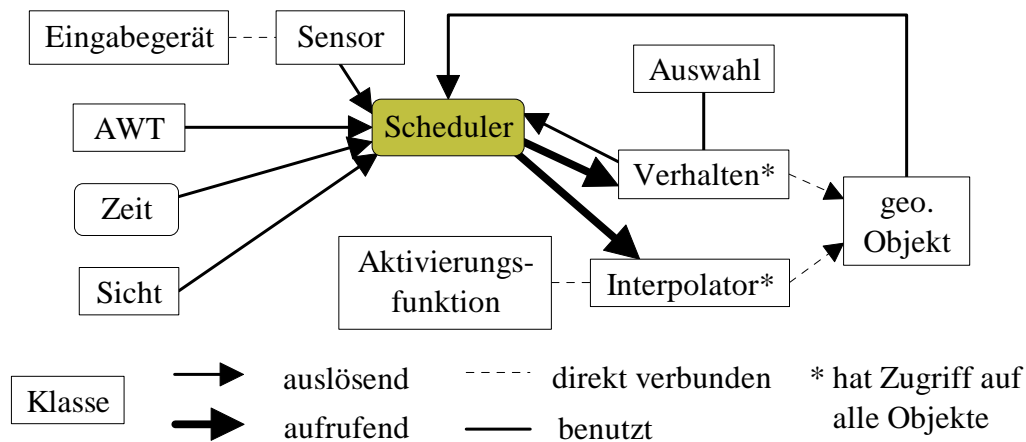


Abbildung 9: Das Animationsmodell von Java3D

Um die Zusammenhänge besser begreifen zu können, werden nun die einzelnen Komponenten besprochen.

## 5.2 Eingabegeräte und Sensoren

Da Java schon Tastatur und Maus als Eingabegeräte unterstützt, ist die Eingabegerät-Klasse von Java3D hauptsächlich für Geräte wie den Joystick oder ein Sechs-Freiheitsgrade-Tracker zuständig. Ein Sechs-Freiheitsgrade-Tracker ist ein Gerät, das im Gegensatz zum Joystick, der 2 Dimensionen hat, 3 Dimensionen hat, dies ist z.B. ein 3D-Handschuh. Da diese Geräte nicht wichtig sind, um das Animationsmodell von Java3D zu verstehen, wird diese Klasse nicht weiter behandelt. Bei Interesse sollte in der Java3D-Spezifikation (Sun 98-1, Kapitel 10.1) nachgeschaut werden.

Die Sensor-Klasse wird dazu genutzt, die Werte, welche die Eingabegerät-Objekte von ihren Eingabegeräten bekommen, mit einer Zeitmarke zu versehen. Die Sensor-Klasse speichert auch den Status der Knöpfe und Schalter der Geräte zwischen. Die Zwischenspeicherung der Dimensionswerte des Eingabegerätes ist so gestaltet, daß die Werte, welche das Eingabegerät-Objekt liefert, in Position und Orientierung umgerechnet werden. Die Position- und die Orientierungsangaben beziehen sich auf den Sensor innerhalb des Koordinatensystems der virtuellen Welt. Für diese Umrechnung kann ein Anfangspunkt für den Sensor angegeben werden, ansonsten wird der Koordinatenursprung als Anfangspunkt angenommen. Das Format, welches das Sensor-Objekt liefert, ist unabhängig von dem Eingabegerät, das hinter diesem Sensor steckt. Die Sensoren haben bei Java3D eine etwas andere Bedeutung als bei VRML, denn hier sind sie einzig und allein für Eingabegeräte und nicht auch noch für Kollisionen oder Zeit zuständig. Ein Sensor übernimmt fast dieselbe Aufgabe wie das AWT, wenn dieses die Maus beobachtet, sich deren Knopf-Status und Position merkt und dementsprechend Ereignisse erstellt. Der Sensor macht dies aber im 3D-Raum und nicht im 2D-Raum wie das AWT. Die Abgrenzung zwischen der Sensor-Klasse und der Eingabegerät-Klasse ist die der Gerätetreiber-Schnittstelle zur API-Schnittstelle. Das heißt, wenn normalerweise auf ein Hardware-Gerät zugegriffen wird, existiert ein genereller Treiber für alle Prozesse eines Betriebssystems und eine Zugriffsmethode einer Programmiersprache, die es ermöglicht, die Werte des Treibers zu erhalten. Diese Zugriffsmethode wird dann von einem einzelnen Prozeß benutzt. Es besteht eine Client/Server-Struktur, die es mehreren Prozessen ermöglicht, die Werte eines Eingabegerätes abzulesen. Bei Java3D ist dieses Konzept eine Ebene höher gerutscht, und zwar von der Betriebssystemebene in die Programmiersprachenebene, so daß zusätzlich zu dem Betriebssystemtreiber noch ein

Java3D-’Treiber’ hinzukommt, auf welchen die Sensor-Zugriffsmethode zugreift. Daraus wird auch klar, warum es sich nicht immer um ein eins-zu-eins-Verhältnis zwischen Sensor- und Eingabegerät-Objekt handeln muß. Es braucht ein Eingabegerät-Objekt nur an ein weiteres Sensor-Objekt ’angehängt’ werden. Dies macht Sinn, sobald die Werte des Eingabegerätes an zwei verschiedene Interessenten, also zwei verschiedene Objekte, weitergegeben werden sollen, welche die Werte dann nach ihren eigenen Bedürfnissen nutzen.

Als Demonstrationsprogramm soll ein unverändertes Beispiel von Sun dienen. Es wird in diesem Programm ein virtuelles Eingabegerät durch Regler in einem Fenster erzeugt. Das heißt, es wird mit der Maus an verschiedenen Reglern die Position des Betrachters verändert. Die Regler unterteilen sich in:

- Positionsbewegungsregler: Hoch/Runter, Links/Rechts und Vorne/Hinten
- Sichtregler: Winkel, Hoch/Runter und Links/Rechts

```
...
InputDevice device =
    new VirtualInputDevice( positionControls, wheelControls );
...
device.initialize();
u.getViewer().getPhysicalEnvironment().addInputDevice( device );
...
SensorBehavior s =
    new SensorBehavior( viewTrans, device.getSensor(0) );
...

... Behavior {...
private WakeupOnElapsedFrames conditions =
    new WakeupOnElapsedFrames(0);
...
sensor.getRead( transform );
transformGroup.setTransform( transform ); ...}

... InputDevice {...
public void pollAndProcessInput() {
    sensorRead.setTime( System.currentTimeMillis() );
    ...
    sensorRead.set( newTransform );
    sensors[0].setNextSensorRead( sensorRead ); } ... }

```

Programmbeispiel 5: Sensor und Eingabegerät mit Java3D

Es wird im Programmbeispiel 5 erst einmal ein Eingabegerät-Objekt von einer selbst geschriebenen Klasse erstellt. Dann wird das Eingabegerät initialisiert und an das aktuelle Sicht-Objekt gehängt. Dann wird noch ein Verhalten-Objekt erzeugt, das immer direkt vor dem Erstellten eines Abbildes aufgerufen wird und nichts anderes tut, als die aktuellen Sensorwerte in das Transformation-Objekt des Sicht-Objektes zu übertragen. Die Eingabegerät-Klasse muß so geschrieben werden, daß eine Methode existiert, welche die Werte einliest und in den Sensor schreibt, es wird dabei noch zusätzlich die Zeit in Millisekunden mit angegeben. Das Beispiel ist für den Nachfrage-Betrieb und nicht für den nicht-/Block-Betrieb geschrieben, das heißt es wird auf Anfrage das physikalische Eingabegeräte ’befragt’ und nicht permanent der Zustand des Eingabegerätes abgelesen. Letzteres ist ressourcenaufwendiger, da sehr viele Daten umsonst abgelesen und weitergegeben werden. Wenn der Zustand des Eingabegeräts kontinuierlich benötigt wird, sollte der nicht-/Block-Betrieb statt dem Nachfrage-Betrieb gewählt wer-

den. Bei dem nicht-/Block-Betrieb wird einmal das Interesse an den Werten des Eingabegeräts bekundet und dann wird regelmäßig die selbst geschriebene `processStreamInput`-Methode aufgerufen und abgearbeitet. Diese Methode macht nichts anderes, als die Werte von dem Treiber einzulesen und an den Sensor weiter zugeben. Der Unterschied zwischen dem Block- und dem nichtBlock-Betrieb besteht darin, daß bei dem Block-Betrieb der Java3D-Treiber-Thread 'angehalten' / blockiert wird, bis Daten vom Hardware-Treiber vorhanden sind. Beim nicht-Block-Betrieb wird der Thread nicht blockiert und falls keine Daten vorhanden sind, werden auch keine an den Sensor weitergegeben.

Eingabegeräte dienen auch zur Erzeugung von Ereignissen, sie lassen also den Scheduler in Aktion treten. Was für Aufgaben der Scheduler hat, ist Bestandteil des nächsten Unterkapitels.

### 5.3 Der Scheduler

Der Aufgabenbereich des Scheduler-Mechanismus ist nicht nur, eine Tabelle bereit zu stellen, anhand welcher zu sehen ist, was wann getan werden muß, sondern er stößt die Tätigkeiten an, indem die Objekte benachrichtigt werden, die in der entsprechenden 'Zeile' des Ereignisses der Tabelle stehen. Die Zuweisung einer Aufgabe zu einem Ereignis wird durch die `wakeupOn`-Methode der aufzurufenden Objekte erledigt. Als Parameter hat diese Methode einen Wert, der angibt, auf welches Ereignis oder welche Ereignisse reagiert werden soll (Ereignisse-Übersicht im Kapitel 2.2, Seite 7). Wenn auf mehrere Ereignisse reagiert werden soll, dann werden die einzelnen Ereignisse mittels 'and' und 'or' zu dem gewünschten Ereignis verknüpft. Programmiertechnisch ist zu beachten, daß eine Region angegeben werden muß, in welcher der Betrachter / der Sensor sein muß, damit das Auftreten des Ereignisses dem Objekt signalisiert wird. Dabei ist die Region an ein Transformation-Objekt gebunden, so daß bei der 'Positionsänderung' des Transformation-Objektes sich auch die Position der Region ändert. Der Schedulerregion-Mechanismus dient dazu, den Scheduler / den Prozessor von Aufgaben zu befreien, die nicht nötig sind. Es ist z.B. nicht notwendig, daß ein Bleistift sich um die eigene Achse dreht, wenn der Betrachter so weit von dem Bleistift entfernt ist, daß er ihn nur noch als Strich sieht. Dieser Mechanismus ist somit eine Optimierungsmaßnahme, die sehr wichtig ist, um die Rechneransprüche für die Erzeugung von Animationen von vornherein auf ein Minimum zu bringen.

Intern werden erst alle Schedulerregionen mit der Betrachterposition geschnitten und dann wird überprüft, ob die Ereignisse der 'aktiven' Schedulerregionen mit dem aufgetretenen Ereignis übereinstimmen. Dann werden nur die Objekte aufgerufen, bei denen beide Überprüfungen positiv sind. Damit die Berechnung der aktiven Schedulerregionen beschleunigt / vereinfacht wird, werden die Schedulerregionen in einem hierarchischen Baum aufgenommen. Die Baumstruktur ist dabei so angelegt, daß sie einem Octree gleicht. Der Unterschied besteht nur darin, daß diese Struktur sich ständig ändern kann, da es ja Schedulerregionen gibt, die ihre Position ändern. Auch die Ereignisse werden zur schnelleren Auffindung in einer Baumstruktur verwaltet. Es ist dann leicht, trotz der Verknüpfung von mehreren atomaren Ereignissen, zu überprüfen, ob das entsprechende Objekt, in der Regel ein Verhalten-Objekt, aufgerufen werden soll. Die Struktur des Baumes ist dabei so aufgebaut, daß anhand der 'or'-Verbindungen die Ereignisse / die Äste auf der selben Ebene getrennt werden und daß somit bei den 'and'-Verbindungen die Ereignisse ebenenweise getrennt werden. An jeden Knoten wird dann natürlich ein Verweis auf die entsprechende Zeile in der Schedulertabelle gehängt [Sun 98-1, Kapitel 9.4].

Der Benachrichtigungsvorgang läuft also in der Weise ab, daß von dem Objekt der Wunsch, bei einem bestimmten Ereignis benachrichtigt zu werden, in der Form geäußert wird, daß es die eigene `wakeupOn`-Methode aufruft und dabei die entsprechende Ereignisverknüpfung übergibt. Es wird dann die entsprechende Zeile in der Schedulertabelle erstellt und die beiden Baumstrukturen, Schedulerregion und Ereignisverknüpfung, werden aktualisiert. Wenn jetzt ein Ereignis

erfolgt, dann wird anhand der Scheduleregionbaumstruktur entschieden, welche Objekte überhaupt räumlich infrage kommen und dann wird anhand der Ereignisverknüpfungsbaumstruktur überprüft, ob die räumlich infrage kommenden Objekte auch auf das Ereignis gewartet haben. Wenn dem so ist, wird von jedem zutreffenden Objekte die `processStimulus`-Methode aufgerufen und das Aufrufkriterium von diesen Objekten aus der Ereignisverknüpfungs-Baumstruktur gelöscht. Die letzte Maßnahme dient dazu, daß ein Objekt sein aufrufendes Ereignis während der Laufzeit ändern kann. Es ist aber somit auch notwendig, daß jedesmal vor dem Verlassen der `processStimulus`-Methode ein Aufruf der `wakeupOn`-Methode geschieht, sonst würde das Objekt nicht noch einmal vom Scheduler aufgerufen werden.

Zusätzlich zu der `wakeupOn`-Methode gibt es noch eine `postId`-Methode, die dazu dient, anderen Verhalten-Objekten ein Ereignis zu schicken. Das heißt, es wird beim Aufruf dieser Methode ein Ereignis generiert und an den Scheduler weitergegeben. Dieser geht dann ganz normal mit diesem Ereignis um. Verhalten-Objekte, die Interesse an einem Post-Ereignis angemeldet haben, müssen allerdings die gleiche Post-ID angegeben haben, wie die Post-ID des generierten Ereignisses, damit sie benachrichtigt werden [Sun 98-1, Kapitel 9.2].

In kritischen Situationen, in denen es wünschenswert ist, daß der Scheduler nicht in Aktion tritt, und dabei den momentanen Vorgang unterbrechen würde, ist es möglich, den Scheduler zu deaktivieren. Nach der kritischen Situation ist es dann auch möglich, den Scheduler wieder zu aktivieren [Sun 98-1, Kapitel 8.7.5]. Eine kritische Situation ist z.B. das Rendern, da während des Rendern sich die 3D-Welt-Daten nicht verändern sollten. Es könnte dieses Vorgehen mit dem Verhindern einer Interrupt-Unterbrechung gleichgesetzt werden. Das ist allerdings nicht ganz richtig, da es anderen Prozessen, wie auch dem Scheduler, weiterhin erlaubt ist, den Renderer zu unterbrechen. Es wird nur verhindert, daß die Daten, auf die der Renderer zugreift, während seiner Arbeit verändert werden. Das heißt, es wird, wenn der Renderer anfangen will ein Frame zu erstellen, ein Signal an den Scheduler gesendet. Dieses Signal bewirkt, daß der Scheduler keine Datenänderung mehr anstößt, also keine Verhalten-Objekte mehr benachrichtigt, dafür aber die Ereignisse zwischenspeichert. Wenn der Renderer mit der Erstellung des Frames fertig ist, wird dem Scheduler signalisiert, daß er wieder die Verhalten-Objekte benachrichtigen darf. Der Scheduler überprüft nun, ob er zwischengespeicherte Ereignisse hat und verarbeitet sie regulär. Das De- und Aktivieren erledigt in den `retained` modi die API selber.

Es ist auch der Scheduler-Mechanismus, wie der Route-Mechanismus, mit dem `message passing`-Mechanismus zu vergleichen (siehe Kapitel 4.2, Seite 18). Diesmal ist es nur umgekehrt, es wird nicht angegeben, wohin die Botschaft weiter gesendet wird, sondern die Ziele geben an, auf welche Ereignisse sie warten. Außerdem gibt es nicht wie in VRML direkte Verbindungen, sondern Ereignisse. Auf Ereignisse kann jedes Verhalten-Objekt warten ohne den Ursprung direkt zu kennen. Der Ursprung eines Ereignisses wird nur indirekt, durch die Art des Ereignisses angegeben. Ganz wichtig ist, daß im Gegensatz zum Animationsmodell von VRML bei Java3D keine Ereigniskaskaden existieren. Eine Ereigniskaskade kann nicht entstehen, weil die Ereignisse nicht an Zeitpunkte gebunden werden, das heißt aber nicht, daß Ereignisse keine Zeit kennen, die Zeitwerte müssen allerdings gezielt abgefragt werden. Dadurch existiert bei dem Java3D Animationsmodell keine Struktur, die vorgibt wie eine Ereigniskaskade abgearbeitet werden muß. Die Struktur ein Ereignis zu bearbeiten wurde oben schon ausführlich beschrieben und ist um einiges 'sauberer', als die Struktur bei VRML, da hier keine Ausnahmen existieren.

Zu dem Schedule-Mechanismus gehört auch die Verwaltung des Zeitgebers, so daß er dafür sorgt, daß die Aktivierungsfunktionen der Interpolatoren mit Werten versorgt werden (siehe nächstes Unterkapitel).



## 5.4 Interpolatoren und ihre Aktivierungsfunktion

Wie aus Abbildung 9 (Seite 28) zu ersehen ist, spielt die Aktivierungsfunktion eine wichtige Rolle, wenn Interpolatoren genutzt werden. Die Aktivierungsfunktion kann dabei recht komplex werden. So ist es möglich anzugeben, wie lange der Ausgabewert Null ist, wie lange er eins ist und wie die Steigung der Übergänge zwischen Null und eins sind und wie lang sie dauern sollen. Die Ausgabewerte der Aktivierungsfunktion haben pro Zyklus eine feste Reihenfolge, sie ist wie folgt: aufsteigend, eins, absteigend, Null. Daher reicht es aus, nur die Länge dieser Zykusteile anzugeben. Wichtig ist dabei noch zu wissen, daß der Zuwachs nicht konstant sein muß. Der Wert, der das Steigungsverhalten angibt, wird getrennt nach aufsteigend und absteigend angegeben. Dieser Wert legt fest, ob es sich um eine konstante Steigung (=0) oder um eine stärkeren Steigung in der Mitte (>0) handelt. Das heißt, es ist möglich, eine nicht-lineare Aktivierungsfunktion anzugeben (siehe Abbildung 10) [Sun 98-1, Kapitel 9.6.1-9.6.3].

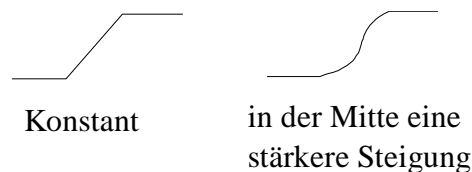


Abbildung 10: Beispiele von Aktivierungsfunktionsteigungen  
(Übernommen aus Sun 98-1, Seite 246)

Der Wert, der das Steigungsverhalten angibt, ist nichts anderes als eine Millisekundenangabe. Es wird dabei in drei Fälle unterschieden:

1. Kleiner oder gleich Null: Die Steigung ist immer konstant
2. Kleiner als die Hälfte der Zeit des Auf/Abstieges der Aktivierungsfunktion: Die Steigung fängt bei Null an und steigt die angegebene Zeit linear an. Die restliche Zeit bis zur Hälfte des Auf/Abstieges ist sie konstant und dann ist das Verhalten genau entgegengesetzt. Das heißt erst konstante Steigung und dann lineare Abfall bis Null.
3. Größer oder gleich der Hälfte der Zeit des Auf/Abstieges der Aktivierungsfunktion: Die Steigung fängt bei Null an und steigt bis zur Hälfte der Zeit des Auf/Abstieges linear an. Ab dort fällt die Steigung linear ab bis Null. Das heißt die Steigungsfunktion hat in der Mitte der Zeit eine unstetige Stelle. Da die Steigung der Aktivierungsfunktion immer symmetrisch ist, wird eine Angabe, die größer als die Hälfte der Zeit des Auf/Abstieges der Aktivierungsfunktion ist, einfach abgeschnitten, so daß sie genau die Hälfte der Zeit des Auf/Abstieges der Aktivierungsfunktion beträgt.

Als Beispiel soll hier eine Aktivierungsfunktion angegeben werden, die alle Möglichkeiten ausschöpft:

```
Alpha bspAlpha =
    new Alpha(100, Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE,
        50000, 500, 1001, 200, 1002, 1003, 501, 1004);
```

Es wird mit diesem Konstruktor ein Aktivierungsfunktion-Objekt erstellt, daß 100 Zyklen durchläuft. Es ist sowohl ein stetiger Auf- als auch ein stetiger Abstieg der Werte vorgesehen. Die Aktivierungsfunktion wird das erste Mal 50000 msec nachdem das Java-Programm gestartet wurde, aktiviert. Dann erfolgt eine Abstimmphase mit anderen Interpolatoren, die 500 msec lang ist und nur vor dem ersten Zyklus durchlaufen wird. Der Wert der Aktivierungsfunktion ist in dieser Abstimmphase 0 oder 1 je nachdem ob ein Aufstieg vorgesehen ist oder nicht. Der

Aufstieg dauert 1001 msec und hat in seiner Mitte eine leicht stärkere Steigung, das heißt die Steigung steigt bis 200 msec an, dann ist die Steigung konstant bis 801 und von da fällt die Steigung wieder. Der Wert 1 wird 1002 msec weitergegeben. Der Abstieg dauert 1003 msec und hat in seiner Mitte eine stärkere Steigung als am Anfang und am Ende, das heißt, die Steigung steigt bis 501 msec an und von da fällt die Steigung wieder. Der Wert 0 wird 1004 msec weitergegeben. Danach wird mit dem Aufstieg wieder begonnen, wenn nicht schon 100 Zyklen durchlaufen sind. Wenn '-1'-Zyklen angegeben werden, dann werden beliebig viele Zyklen durchlaufen.

Diese Werte aus dem [0,1]-Intervall werden, ähnlich wie in VRML, durch einen Interpolator z.B. in Transformationswerte oder Rotationswerte umgewandelt. Beim Interpolator werden jetzt nur noch Start- und Endwerte angegeben und den Rest erledigt dann die API. Es können allerdings keine Zwischenwerte angegeben werden, dafür ist bei Java3D die Aktivierungsfunktion zuständig.

Die Interpolator-Klasse ist eine Kindklasse von der Verhalten-Klasse, die immer bei Zeitergebnissen aufgerufen wird. Da die Funktionalität der Interpolatoren klar ist, werden nun noch die Details durch ein Beispiel erklärt:

```
RotationInterpolator bspRotator = new RotationInterpolator(bspAlpha,
    objTrans, Axis, 0.0f, (float) Math.PI*2.0f);
```

Es wird mit diesem Konstruktor ein Rotation-Interpolator erstellt, der die bspAlpha-Aktivierungsfunktion benutzt. Es soll die Rotation auf das objTrans-Transformation-Objekt angewandt werden. Es soll um die in dem Transform3D-Objekt 'Axis' angegebenen Achsen rotiert werden. Der Anfangswinkel ist 0 und der Endwinkel ist  $2\pi$ .

Durch die Angabe in msec und den zentralen Scheduler ist es möglich, sogar auf verschiedenen Rechnern zwei Interpolatoren laufen zu lassen, und doch im Abbild der 3D-Welt ein synchrones Verhalten der Interpolatoren zu sehen. Vorrausgesetzt die Uhren der beiden Rechner laufen gleich.

Da die Beschränktheit der Interpolatoren auffällt, soll nun auch bei Java3D, wie vorher bei VRML, das umfassendere Werkzeug besprochen werden, die Verhalten-Klasse.

## 5.5 Die Verhalten- und die Auswahl-Klasse

Die Objekte der Verhalten-Klasse werden grundsätzlich durch das Aufwecksystem des Schedulers aufgerufen. Die Methode, die der Scheduler von dem Verhalten-Objekt aufruft, ist die processStimulus-Methode. Da es sich aber um ganz normale Objekte handelt, können die Methoden des Objektes jederzeit von anderen Objekten aus aufgerufen werden. Java-Programme bestehen nur aus Objekten. Die Verhalten-Klasse hat natürlich einen Konstruktor und eine Initialisierungsroutine. Der Konstruktor wird wie gewohnt bei der Definition eines Objektes aufgerufen und die Initialisierungsroutine wird aufgerufen, wenn die Scheduleregion betreten wird. Die Scheduleregion ist die Region, in der sich der Betrachter befinden muß, damit der Scheduler das Ereignis an das Verhalten-Objekt weitergibt. Es gibt zwingendermaßen Methoden, welche die Scheduleregion festlegen, sowie die Methode, die nötig ist, um dem Scheduler mitzuteilen, auf welches Ereignis das eigene Objekt wartet. Diese Methode wird nicht innerhalb der Verhalten-Klassendefinition benutzt, sondern bei der Erstellung des Verhalten-Objektes. Die postIt-Methode der Verhalten-Klasse ist ein Mechanismus, ein Ereignis zu kreieren. So kann ein Objekt der Verhalten-Klasse ein postIt-Aufruf dazu verwenden, ein Signal zu senden, das andere Objekte über den Scheduler empfangen können und so z.B. nur aufgerufen werden, wenn das erste Verhalten-Objekt es wünscht. Dieses Signal ist nichts anderes als ein künstliches Ereignis, das ganz normal vom Scheduler verarbeitet wird, indem er die Objekte aufruft, die Interesse an diesem Signal angemeldet haben. Eine weitere Standardme-

thode ist die getView-Methode, die das primäre Sicht-Objekt zurückgibt. Primäres Sicht-Objekt ist immer das, welches als erstes an das aktive SichtPlattform-Objekt angehängt wurde. Es gibt immer nur ein SichtPlattform-Objekt. Es kann so die Entfernung oder die Richtung zu dem Betrachter in eine Berechnung mit einfließen, die in einem Verhalten-Objekt vorgenommen werden soll, z.B. muß bei einem Billboard-Objekt die Blickrichtung und beim LoD-Objekt die Entfernung erfragt werden [Sun 98-1, Kapitel 9.5.1].

```
public class bspBehavior extends Behavior
{
    // Variablendeklaration
    public bspBehavior(Parameter p) { // Variablen initialisieren }
    public void initialize() { wakeupOn(wakeupKriterium); }
    public void processStimulus(Enumeration criteria)
    {
        // Die Handlung
        wakeupOn(wakeupKriterium);
    }
}
```

Das Beispiel stellt schematisch eine Verhalten-Klasse dar. Objekte dieser Klasse werden dann ganz normal mittels Konstruktor erstellt. Durch den Aufruf des Konstruktor wird das entstandene Objekt automatisch in den Scheduler eingefügt. Allerdings darf nicht vergessen werden, die Scheduleregion anzugeben, indem von dem Objekt eine der Scheduleregionen-Setzer-Methoden aufgerufen wird. Zu beachten sind die zwei wakeupOn-Aufrufe, diese müssen an dieser Stelle stehen, da sonst das Aufrufkriterium für das Objekt nicht bekannt ist. Daß das Aufrufkriterium jedesmal wieder nach einem Aufruf gesetzt werden muß, hat zwar den Vorteil, daß jedesmal ein neues Kriterium benutzt werden kann, aber auch den Nachteil, daß vergessen werden kann, es zu setzen.

Sehr nützlich für die Verhalten-Klassen erweisen sich die Auswahl-Methoden, der Lokales-Objekt-Klasse oder der ZweigGruppen-Klasse. Diese Methoden können mittels Auswahl-Objekten eine Eingrenzung auf entweder einen *Koordinatenpunkt*, einen *Strahl* oder ein *Segment* vornehmen. Ein Segment ist eine Linie, die durch Anfangs- und End-Punkt angegeben wird. Die Methoden liefern dann die Objekte zurück, die mit diesen Auswahl-Objekten einen Schnittpunkt haben. Standardmäßig sind in allen Rendermodi alle Objekte befähigt, von diesen Methoden referenziert zu werden, wenn Objekte aber prinzipiell nicht 'angefast' werden sollen, dann kann dies auch mittels einem setCapability-Aufruf verboten werden. Als Auswahl-Methoden gibt es die Möglichkeiten:

- Alle Objekte, auf die das Geforderte zutrifft, zurückgeben.
- Alle Objekte, auf die das Geforderte zutrifft, sortiert nach Abstand zum Anfangspunkt zurückgeben.
- Das Objekt, was die geringste Entfernung vom Betrachter hat und auf welches das Geforderte zutrifft, zurückgeben.
- Das erste Objekt, das den Anforderungen genügt, zurückgeben, egal wie weit es vom Betrachter entfernt ist.

Dabei werden nicht einfach nur Referenzen von den entsprechenden Objekten zurückgegeben, sondern auch die Referenzen auf die Objekte, die hierarchisch über dem Objekt stehen, also der 'Weg' von dem Objekt, das den Schnittpunkt hat, bis zu seinem ZweigGruppen-Objekt oder seinem Lokales-Objekt.

Hinter den Auswahl-Methoden steht ein Algorithmus, der bei der Strahlmethode alle Objekt-Strahl-Schnitte durchführt. Die Punktmethode ist trivial, da hier nur Koordinaten verglichen werden müssen. Die Segmentmethode ist der Strahlmethode sehr ähnlich, nur wird hier noch überprüft, ob der Schnittpunkt, falls vorhanden, innerhalb des Segmentes ist. Es werden dabei nur Objekte berücksichtigt, die Mitglied des eigenen Objektes sind und das Recht haben, ausgewählt zu werden. Wenn ein Objekt den Voraussetzungen entspricht, dann wird es entweder sofort zurück gegeben, oder es wird temporär in einer Liste gespeichert. Wenn alle Objekte untersucht wurden, dann wird die Liste evtl. noch sortiert und dann evtl. nur das oberste Element zurückgegeben [Sun 98-1, Kapitel 10.3].

Es ist zu beachten, daß es Auswahl-Methoden und Auswahl-Klassen gibt. Die Klassen stellen für die Methoden die Objekte, nach denen ausgewählt wird. Und die Methoden liefern die Objekte, auf welche die Auswahl zutrifft.

Zum Abschluß noch ein Beispiel, bei dem erst ein Auswahl-Strahl erzeugt wird und dann alle zutreffenden Objekte vom ZweigGruppen-Objekt in einem Vektor gespeichert werden, dessen Elemente schon nach der Entfernung zum mousePos-Punkt sortiert sind.

```
PickRay pickRay = new PickRay(mousePos, mouseVec);
SceneGraphPath sceneGraphPath[] = branchGroup.pickAllSorted(pickRay);
```

## 5.6 Optimierungen durch das Animationsmodell

Nachdem nun alle Teilbereiche des Animationsmodelles von Java3D besprochen wurden, soll wie im entsprechenden Unterkapitel bei VRML (Kapitel 4.6, Seite 25) die Optimierungen betrachtet werden, die durch das Animationsmodell realisiert werden, dabei werden die Optimierungen in drei Bereiche unterteilt. Der erste Bereich dreht sich um die Verfeinerung der Algorithmen zur Abbildberechnung, der zweite Bereich beschreibt die Ereignisverwaltung (=Schedulealgorithmen) und der dritte Bereich enthält die Laufzeitübersetzung der API-Sprache in eine 'maschinen-gerechte'- Sprache.

Im ersten Bereich ist, wie bei VRML, zu bemerken, daß Java3D in der Regel auf OpenGL aufbaut und daher (fast) keine Optimierungen möglich sind. Es gibt zwar auch eine DirectX-Version, aber da sie nur eine zweitrangige Rolle spielt, hat sich kein grundlegender Mechanismus bei dieser Version geändert. Wie in VRML gibt es auch in Java3D ein LoD-Objekt und dieses führt zu dem Ergebnis, daß komplexere Objekte, die noch weit entfernt sind, als ein einfaches Objekt gesehen werden und somit die Berechnungen weniger sind. Den gleichen Mechanismus verwendet das Sicht-Objekt mit seinem Clip-Parameter; dieser Parameter gibt eine Entfernung an, in der sich ein Objekt vom Betrachter aus befinden muß, damit es bei der Abbil-derstellung berücksichtigt wird. Prinzipiell hätte auch ein LoD-Objekt diese Aufgabe übernehmen können, aber es ist so eine Verfeinerung des reinen Clippings möglich, die wiederum eine Performancesteigerung mit sich bringt.

Im zweiten Bereich kommt insbesondere der Scheduling-Mechanismus zum Tragen. Der vom Prinzip wieder mit dem Clipping-Mechanismus verwandt ist. Es wird so verhindert, daß zu viel Rechenleistung in unnötige, weil nichts (sichtlich) verändernde, Verhalten-Objekt-Aufrufe verbraucht wird.

Der dritte Bereich ist mit den Bemühungen um ein Binäres-VRML-Dateiformat ein wenig verwandt. Java3D unterstützt, wie schon erwähnt drei Rendermodi. Diese drei Rendermodi enthalten die verschiedenen Bestrebungen der anderen beiden 3D-Graphik-Schnittstellen. Um die verschiedenen Ansätze und Modi einmal gegenüberzustellen, wird ein schemenhafter Vergleich gemacht, der sicherlich nicht ein 1:1-Vergleich ist:

1. Der immediate mode ist ganz gut mit der Struktur von OpenGL-Programmen zu vergleichen, da hier kein Hierarchie-Baum aufgezwungen wird

2. Der retained mode ist gut mit dem (ASCII-)VRML-Dateiformat zu vergleichen, da beide eine Baumstruktur der geometrischen Objekte besitzen und es Bemühungen gibt, Rechenleistung durch verschiedene Culling-Mechanismen einzusparen.
3. Der compiled-retained mode ist hingegen nur schwach mit dem zukünftigen Binär-VRML-Dateiformat zu vergleichen. In beiden fruchtet die Bemühungen, die Umwandlung der Programmiersprache in ein 'maschinen-gerechtes' Format vor der 'Laufzeit' (im Sinne der visuellen Präsentation) in ein Zwischenformat zu bringen, was die Übersetzungsarbeit zur 'Laufzeit' verringert. Dabei ist allerdings zu beachten, daß Java3D im compiled-retained mode einen großen Teil der Optimierung durch die Unveränderbarkeit von Variablen erzielt und bei VRML immer alle Variablen veränderbar sind.

Es sei noch erwähnt, daß auch hier das Prinzip der Trennung von statischen und dynamischen Bereichen der 3D-Welt zur Optimierung eingesetzt wurde, indem die Verhalten-Klasse geschaffen wurde.

### 5.7 Zusammenfassung

Es wurde in diesem Kapitel gezeigt, daß das Animationsmodell von Java3D als Ganzes übersichtlich und gut strukturiert ist. Außerdem zeigt sich, daß die einzelnen Komponenten des Animationsmodells sehr sauber, nach OO-Manier, entworfen sind und es keine Ausnahmen gibt. Im Zentrum des Animationsmodells steht der Scheduler, der alle Ereignisse überwacht und gegebenenfalls Verhalten-Objekte, bzw. Interpolator-Objekte aufruft. Diese Objekte wiederum verändern die Daten der 3D-Welt. Als Innovation kann auch die Aktivierungsfunktions-Klasse gesehen werden, welche die Aktivierung des Interpolator steuert.

Die Prinzipien, die es in VRML schon in ähnlicher Form gegeben hat, wurden in Java3D noch verfeinert und eine richtige 3D-Graphik-API geschaffen. Es sind keine grundlegend neuen Algorithmen geschaffen worden, welche die Performance stark steigern würden, aber Java3D ist eine umfassende und sehr sauber programmierte API, die sicher ihren Weg zum Internet-standard machen wird. Die Hoffnung auf eine schnellere 3D-Sprache ist zwar geschmälert, es bleibt aber die Hoffnung, daß in Zukunft ein grundlegend anderes 3D-Animationsmodell geschaffen wird, was die Performance deutlich steigern wird.

## 6. Vergleich der Animationsmöglichkeiten der 3D-Graphik-Schnittstellen

Nachdem die 3D-Graphik-Schnittstellen auf ihre Möglichkeiten hin betrachtet wurden, Animationen zu erstellen, soll nun ein Vergleich der Schnittstellen selbst stattfinden. Allgemein kann festgestellt werden, daß OpenGL kein Animationsmodell hat und Animation nur durch eigene Routinen erzeugt werden kann. Eine nähere Betrachtung der Animationsmodellunterschiede ist daher nur bei den beiden 3D-Graphik-Schnittstellen VRML und Java3D notwendig. Es wird aber für die VRML- und Java3D-Beispiel in der Regel auch eine Umsetzung in OpenGL gezeigt. Die Tatsache, daß immer eine OpenGL-Umsetzung existiert, läßt sich dadurch erklären, daß die Rendermaschinen der beiden anderen 3D-Graphik-Schnittstellen normalerweise in OpenGL geschrieben sind (siehe Kapitel 4.6 und 5.6). In den folgenden Unterkapiteln wird der Versuch unternommen, alle Beispiele 1:1 in die anderen Schnittstellen umzusetzen. Wenn der Aufwand zu groß dafür wird, dann wird die entsprechende Umsetzung nur schemenhaft vollzogen.

Die einzelnen Unterkapitel werden sich mit jeweils einem speziellen Unterschied vom VRML- und Java3D-Animationsmodell beschäftigen und aufzeigen, welche Konsequenzen sich daraus für den Programmierer ergeben. Es werden die sich entsprechenden Teile der Animationsmodelle von VRML und Java3D verglichen. So wird z.B. die Realisierung der Interpolatoren oder die möglichen Ereignisse der beiden Schnittstellen verglichen. Als erstes werden die Konzepte Verhalten-Objekt und Script-Knoten gegenüber gestellt. Den Abschluß dieses Kapitels bildet die Untersuchung der Realisierungsmöglichkeit eines VRML-Browsers, der in Java3D geschrieben ist. Dabei greift dieses letzte Unterkapitel das erworbene Wissen der vorherigen Unterkapitel auf und zeigt eventuelle Probleme bei der Umsetzung von VRML-Komponenten in Java3D auf.

### 6.1 Verhalten-Objekte und Script-Knoten

Um die beiden Konzepte Verhalten-Klasse und Script-Knoten zu vergleichen, wird zunächst das Konzept des Script-Knotens, in Bezug auf die Nutzung des AWTs, näher betrachtet. Im Anschluß dieser Untersuchung wird ein Beispiel gezeigt, bei dem die Tastatureingabe überwacht wird. Anhand dieses Beispiels wird dann die Beschreibung des Verhalten-Klassen-Prinzip, sowie eine Umsetzung desselben Problems in OpenGL, bzw. C++, vorgenommen. Zum Schluß folgt eine Bewertung der Konzepte.

Der Script-Knoten ist nicht sehr umfangreich, er bietet nur die Möglichkeit, Berechnungen auszuführen und die sich daraus ergebenden Veränderungen anzustoßen. Die Möglichkeiten werden erst durch Einsatz von Java größer, wenn dort ein zusätzliches Objekt benutzt wird, das unabhängig von den Aufrufen des VRML-Browsers arbeitet. Dieses Vorgehen wurde bisher mit dem Begriff 'asynchrones Verhalten' bezeichnet. Schematisch wird dieses Vorgehen in der Abbildung 11 gezeigt.

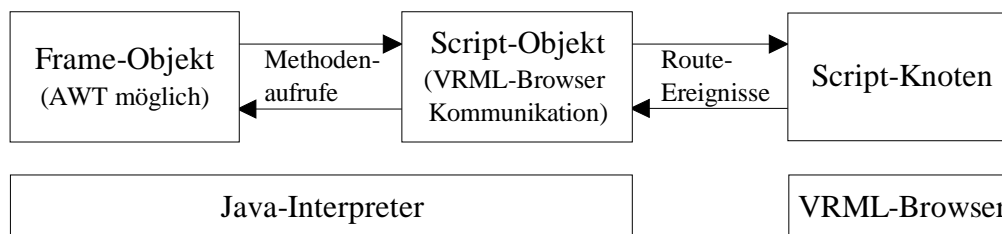


Abbildung 11: Asynchrones Verhalten von Java-Objekten zum VRML-Browser

Zusätzlich existiert auch noch die Möglichkeit mit dem EAI zu arbeiten. Dies gehört jedoch nicht zum VRML97-Standard und da das EAI auf den Route-Mechanismus zurückgreift und

nicht auf den Script-Knoten, wird es hier nicht weiter besprochen.

Es bleibt also nur der Rückgriff auf das asynchrone Verhalten von Java-Objekten. Hier handelt es sich um einen einfach anzuwendenden Trick. In der Initialisierungsphase des Script-Objektes wird ein Objekt erstellt, welches in der Regel ein Frame-Objekt ist und eine Referenz auf sich selber mitgibt. So können die beiden Objekte miteinander kommunizieren. Das Script-Objekt kann mittels Route-Ereignissen mit dem VRML-Browser kommunizieren. Das Programmieren der Kommunikation mit dem VRML-Browser geschieht wie bei JavaScript (siehe Kapitel 4.5, Seite 22). Das heißt, die Kommunikation geschieht in der Regel über die eventIn und eventOut-Variablen des Script-Knotens. Es können aber auch field-Variablen, die sich innerhalb des Script-Knotens befinden, eingelesen und verändert werden. So kann wieder über den Umweg des Use-Befehls jede beliebige Variable aus dem Transformationsbaumes verändert werden. In diesem Fall muß die directOutput-Variable auf 'TRUE' gesetzt werden, damit die Änderungen auch wirklich durchgeführt werden. Die genaue Bezeichnung der Java-Methoden um VRML-Variablen einzulesen oder zu schreiben, ist in [Lea, Kapitel 3] zu finden. Das asynchrone Verhalten von Java-Objekten ist in [Lea, Kapitel 5] mit einigen Beispielen beschrieben.

Der Vorteil dieser Vorgehensweise liegt darin, daß das Frame-Objekt auf AWT-Ereignisse reagieren kann und so mittels AWT-Methoden die VRML-Welt verändert werden kann. Es kann natürlich auch das Net-Paket von Java benutzt werden. Allerdings ergeben sich in den meisten Fällen Sicherheitsprobleme, die darin bestehen, daß eine Java-Applikation, z.B. eine Frame-Klasse, also kein Applet, nachgeladen wird und die beiden Java-Objekte miteinander kommunizieren. Das Vorgehen, um das Programmbeispiel 6 zu betrachten, ist in der Readme-Datei im 'Programmbeispiel 6'-Verzeichnis beschrieben.

```

DEF OBJ Transform {...} # Das zu bewegendende Objekt definieren
DEF SCRIPT Script # Die Schnittstelle zum Applet 'script'
{ directOutput TRUE # Dadurch Variablenänderungen vom OBJ erlaubt
  field SFNode Obj USE OBJ # Referenz auf den OBJ-Knoten
  ... }

public class script extends Script # Hier kein AWT möglich
{ ...
  public void initialize() // Aufruf beim Laden des Script-Knotens
  {trans=(Node)((SFNode)getField("Obj")).getValue();
    pos=(SFVec3f)trans.getExposedField("translation");
    key_e=new KeyEvents(this); // Hier wird das Frame-Objekt erstellt
    ... }
  public void shutdown() { key_e.dispose(); } // Frame-Objekt löschen
  public void get_translation(float[] trans) { pos.getValue(trans); }
  public void set_translation(float[] trans) { pos.setValue(trans); }}

public class KeyEvents extends Frame // Hier AWT möglich
{ ...
  KeyEvents(script owner)
  {...
    my_script = owner; // Referenz auf Besitzer-Objekt }
  public boolean keyDown(Event k_event,int key)
  { my_script.get_translation(pos_f);
    ... // Berechnung der neuen Translations-Werte
    my_script.set_translation(pos_f);
    ...}
}

```

Programmbeispiel 6: Asynchrones Verhalten von Java-Objekt und VRML-Welt

Die Programmierung ist demnach nicht schwierig. Es müssen aber bei Zugriffen auf Variablen immer zwei Methoden geschrieben und auch zur Laufzeit immer zwei Methoden von verschiedenen Objekten ausgeführt werden. Des weiteren muß zur Laufzeit auch noch mit dem VRML-Browser kommuniziert werden, so daß hier bei komplexeren Anwendungen doch ein erheblicher Rechenaufwand entsteht.

Das Programmbeispiel 6 hätte auch leichter mit dem *KeyboardSensor*, den eine VRML-Workinggroup erstellt hat, implementiert werden können. Bei diesem *KeyboardSensor* handelt es sich um einen externen Prototyp (siehe VRML-Spezifikation). Selbst die Workinggroup hält in der Beschreibung des Sensors fest, daß er nicht für den professionellen Einsatz geeignet ist [WG-KB]. Schließlich handle es sich um einen externen Prototyp, der nicht selber verwaltet werden kann und nicht flexibel, bzw. schnell genug ist, um die Aufgabe eines Tastatur-Sensor zu übernehmen. Es ist das alte Problem, daß eine wichtige Funktion, der Tastatur-Sensor, nachträglich noch in einem Standard übernommen werden soll und dadurch einige Kompromisse eingegangen werden müssen.

Es wird nun das Vorgehen bei Java3D bei gleicher Aufgabenstellung betrachtet. Der Aufbau einer Verhalten-Klasse ist im Programmbeispiel 4 (Seite 16) dargestellt. Es muß das geometrische Objekt (der Würfel) nun mit einem Objekt dieser Klasse verbunden werden. Dies macht der Aufruf:

```
KeyBehavior navigator = new KeyBehavior(objTrans);
objRoot.addChild(objTrans);
objRoot.addChild(navigator);
```

Es wird einfach ein Objekt von der Verhalten-Klasse erstellt und dann an dasselbe ZweigGruppen-Objekt gehängt wie das zu verändernde (geometrische) Objekt. Es ist zwar bei Java3D anfänglich viel Programmcode zu schreiben, bei VRML ist es nicht viel weniger, dafür ist dies eine integrierte und direkte Lösung, das heißt, daß nicht wie bei der VRML-Lösung mit zwei Interpretern und zwei Objekten gearbeitet werden muß.

Das Vorgehen bei OpenGL ist im Grunde ähnlich wie bei Java3D, es wird mit einem zentralen Scheduler gearbeitet, z.B. mit den für den MFC (MicrosoftFoundationClasses) Programmierer vertrauten Windows-Messaging. Die Aufgaben werden bei OpenGL mit Methoden abgearbeitet, die im Gegensatz zu Java3D nicht in extra Objekte gefaßt werden müssen. Ein kurzer Auszug aus dem Programm soll genügen, um das Vorgehen bei OpenGL zu verdeutlichen.

```
class CCubeView : public CView // Die Darstellung-Klasse
{...
protected:
    // Funktionsaufruf definieren
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    ...
    DECLARE_MESSAGE_MAP() //Das Messaging-System vorbereiten
};

BEGIN_MESSAGE_MAP(CCubeView, CView)
ON_WM_KEYDOWN() // Wenn eine Taste gedrückt wird, soll eine eigene
                  Methode aufgerufen werden
...
END_MESSAGE_MAP()
```



```

void CCubeView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{... // Anhand der gedrückten Taste neue Werte berechnen
    DrawScene(); // Erstellt die zu zeichnenden Objekte
    CView::OnKeyDown(nChar, nRepCnt, nFlags); }

```

#### Programmbeispiel 7: OpenGL und Tastaturabfrage mit MFC

Nachdem nun die Unterschiede beim Implementieren deutlich gemacht worden sind, soll nun eine Bewertung der Prinzipien erfolgen. Wie schon oben erwähnt, muß bei VRML zur Laufzeit mit zwei Interpretern und innerhalb des einen Interpreters mit zwei Objekten gearbeitet werden. Daß dieses Vorgehen nicht besonders effektiv ist, braucht nicht näher beleuchtet zu werden. Außerdem muß beim Implementieren jede Variablenänderungsmethode doppelt programmiert werden, da das Script-Objekt die Änderung zum VRML-Browser nur durchreicht. Bei Java3D sind zur Laufzeit mehrere Threads aktiv, welche Aufgaben wie Rendern, Soundausgabe, Verhalten-Objekte aufrufen übernehmen, so daß zwischen den Threads auch kommuniziert werden muß (siehe Anhang C). Dabei wird aber nicht auf zwei verschiedene Programme zurückgegriffen, und es wird auch nur ein Konzept, die Objektorientiertheit von Java, bei der Programmierung genutzt. Ein Zeitvergleich ist während der Diplomarbeit leider nicht möglich gewesen, da Java3D sich zu dem Zeitpunkt noch im Beta-Stadium befunden hat und somit noch nicht in allen Bereichen zeitoptimiert war (siehe Anhang A). Bei OpenGL ist das Vorgehen so, daß direkt das Betriebssystem benutzt wird und nicht mit mehreren Threads gearbeitet wird. Es gibt nur das eine C-Programm. Außerdem muß nicht, kann aber, die (laufzeitaufwendigere) objektorientierte Programmierung genutzt werden. Als Resultat kann gesagt werden, durch die direkte Einbindung von Java3D in die Programmiersprache Java, ist das Verhalten-Klassen-Konzept schneller und sauberer als das Script-Knoten-Konzept. Hingegen wird eine gut programmierte OpenGL-Anwendung mindestens genauso schnell sein, wie eine vergleichbare Java3D-Anwendung, dafür bietet OpenGL aber nicht den Komfort wie Java3D.

### 6.2 Unterschiede in der Interpolator-Realisierung

Es gibt in VRML, sowie in Java3D, einige vordefinierte Interpolatoren. Standardmäßig gibt es bei beiden 3D-Graphik-Schnittstellen Rotations-, Skalierungs-, Positions- und Farben-Interpolatoren. Dagegen gibt es auch Interpolatoren, die nur eine der 3D-Graphik-Schnittstellen bereitstellen, diese werden als erstes vorgestellt. Danach folgt ein Vergleich der beiden Interpolatoren-Realisierungen. Dieser Vergleich wird durch ein Beispiel abgerundet. Außerdem wird beschrieben, wie ein eigener Interpolator erstellt werden kann. Als Abschluß folgt die Bewertung der verschiedenen Interpolator-Realisierungen.

Java3D hat, im Gegensatz zu VRML, einen Interpolator um den Transparenzwert zu verändern, oder die Möglichkeit anhand Zeitwerten, verschiedene Switch-Kinderknoten zu aktivieren. Dieser Switch-Interpolator ist besonders geeignet, um die Animation eines 'laufenden' Menschen zu realisieren. Eine ganz neue Klasse wurde geschaffen um sogenannte Weg-Interpolatoren bereitzustellen. Dabei besteht der Unterschied zum einfacheren Java3D-Positions-Interpolator darin, daß nicht nur die Position verändert werden kann, sondern auch die Skalierung oder die Rotierung, und daß mehrere Stützpunkte angegeben werden können und nicht nur Anfangs- und Endpunkt.

Genauso gibt es auch Interpolatoren, die nur bei VRML bereitgestellt werden. Es existiert bei VRML ein Interpolator für Koordinatenwerte von geometrischen Objekten. Das heißt, es kann ein Punktvektor, der Bestandteil eines geometrischen Objektes ist, linear verändert werden, so daß z.B. bei einem Quader eine 'Beule' entsteht und diese dann evtl. auch wieder ausgebeult wird. Des weiteren existiert ein Interpolator, der einen Normalenvektor verändert. Dieser Interpolator ist dafür gedacht, das Reflexionsverhalten der Fläche zu verändern. Norma-

lerweise steht der Normalenvektor einer ebenen Fläche orthogonal auf dieser. Soll dies nicht so sein, wird ein davon abweichender Normalenvektor angegeben. Wann es sinnvoll ist, den Normalenvektor zwischen zwei Vektoren linear zu interpolieren, ist allerdings nicht recht zu sehen. Es kann zwar eine Art Schatten erzeugt werden (siehe Programmbeispiel 8), aber für diesen Effekt extra ein Interpolator bereit zu stellen, ist den Java3D-Entwicklern anscheinend nicht wichtig genug gewesen, so daß sie ihn nicht mit aufgenommen haben.

```
...
DEF NormPath NormalInterpolator
{
  key [ 0.0, 0.5, 1.0 ] # Drei Zeitpunkte <=> drei Normalen-Werte
  keyValue
  [
    # Zum Zeitpunkt 0.0
    0.0 0.0 1.0, 0.0 0.0 1.0, 0.0 0.0 1.0, 0.0 0.0 1.0,
    # Zum Zeitpunkt 0.5
    0.0 0.0 1.0, 1.0 0.0 0.0, 0.0 1.0 0.0, 0.0 0.0 1.0,
    # Zum Zeitpunkt 1.0
    0.0 0.0 1.0, 0.0 0.0 1.0, 0.0 1.0 0.0, 0.0 1.0 0.0
  ]
}
...
```

Programmbeispiel 8: Der Normalen-Interpolator von VRML

Da aber sowohl mit den Verhalten-Objekten von Java3D, wie auch mit den Script-Knoten von VRML weitere Interpolatoren geschaffen werden können, ist das standardmäßige Angebot von den 3D-Graphik-Schnittstellen relativ unwichtig. Das Vorgehen bei VRML ist besonders einfach, da dort Zeitwerte mittels des Route-Mechanismus statt einem Interpolator-Knoten einem Script-Knoten weitergegeben werden. Dort werden diese Werte in andere abgebildet, wobei diese Umwandlung nicht zwingend linear sein muß. Dann werden die neuen Werte wieder mittels des Route-Mechanismus an die entsprechenden Variablen weitergegeben. Bei Java3D ist das Vorgehen etwas anders, da eine Klasse erzeugt werden muß, deren Objekte nach vorgegebenem Ereignis aktiviert werden, welche dann die Variablen eines anderen Objektes verändern. Daher werden jetzt die Unterschiede der Interpolatoren, die bei Java3D und VRML vorhanden sind, und deren Umsetzung betrachtet.

Bei Java3D wurde, wie schon erwähnt, der Positions-Interpolator in zwei Kategorien aufgesplittet:

1. Der Positions-Interpolator, der nur Anfangs- und Endpunkt kennt
2. Die Weg-Interpolatoren (siehe Kapitel 2.2, Seite 8)

Bei VRML existiert nur einer der Weg-Interpolatoren, der Positionsweg-Interpolator, und dieser nennt sich dort Positions-Interpolator. Das heißt Java3D hat mehr und umfassendere Positions/Weg-Interpolatoren als VRML und steht VRML bei dieser Art von Interpolatoren in nichts nach. Im Gegenteil, mit der Möglichkeit nichtlineare Aktivierungsfunktionen (siehe Kapitel 4.4, Seite 21) zu benutzen, bietet Java3D mehr Möglichkeiten.

Nichtlineare Aktivierungsfunktionen sind sinnvoll bei der Realisierung einer Pendelbewegung. Bei Java3D ist eine Pendelbewegung mit der Änderung zweier Parameter realisierbar, bei VRML ist die Realisierung aufwendiger. Im Folgenden die wichtigsten Programmteile:

```
...
```

```
Alpha tickTockAlpha =
    new Alpha(-1, Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE,
        0, 0, 5000, 2500, 0, 5000, 2500, 0);
...
```

Programmbeispiel 9: Nichtlineare Aktivierungsfunktion

```
...
url "javascript:function set_fraction( frac, tm )
{
    if (frac<0.5) y = -2.0 * 3.14 * (frac*2.0) * (1.0 - (frac*2.0));
    else y = 2.0 * 3.14 * ((frac-0.5)*2.0) * (1.0 - ((frac-0.5)*2.0));
    ...
    value_changed[3] = y+(3.14/2.0);
}"
...
```

Programmbeispiel 10: Pendelbewegung mit VRML

In Programmbeispiel 9 ist das Wesentliche, daß der 6. und der 9. Parameter des Konstruktors jeweils einen Wert >0 haben. Bei dem Wert von 2500 (msec) handelt es sich um die Hälfte der Auf-/Abstiegszeit der Aktivierungsfunktion (5000 msec). Die Steigung des Auf-/Abstieges fängt also mit Null an und steigt linear bis zur Mitte des Zyklus an. Dort hat sie eine unstetige Stelle und fällt dann linear ab, bis sie Null ist (siehe Kapitel 5.4, Seite 32). Daraus ergibt sich für den Aufstieg der Aktivierungsfunktion ein Aussehen, wie es in der Abbildung 10 (rechte Seite der Abbildung, Seite 32) zu sehen ist. Für den Abstieg der Aktivierungsfunktion gilt entsprechendes. Im Programmbeispiel 10 ist derselbe Effekt realisiert, nur mußte ein extra Script-Knoten eingefügt und sich eine Formel überlegt werden, die denselben Effekt nachbildet. Probleme bereitet dabei, daß es in VRML keine Unterteilung in Aufstieg und Abstieg gibt, sondern nur eine Angabe über die relative Position in einem Zyklus, daher ist die Bedingungsabfrage nötig. Es wird mittels der Parabel:

$$y = x(1-x) = x - x^2 = (-1)x^2 + x,$$

eine Pendelbewegung erzeugt, wie im Programmbeispiel 9.

Bei den anderen gemeinsamen Interpolatoren gibt es im wesentlichen nur einen Unterschied. Dieser Unterschied ist, daß bei den Java3D-Interpolatoren immer nur ein Anfangs- und ein Endwert angegeben wird, bei VRML aber auch noch beliebig viele Zwischenwerte. Das Aufrufen der Interpolatoren durch den Zeitgeber ist bei VRML (Execution-Engine) und bei Java3D (Scheduler) vom Prinzip gleich, da die Execution-Engine bzw. der Scheduler immer vor dem Rendern eines 3D-Welt-Abbildes ihre Arbeit verrichten (für Java3D [Sun 98-1, Kapitel 1.6.2], für VRML siehe Abbildung 6, Seite 18).

In OpenGL sind keine Interpolatoren vorgegeben und auch kein Zeitgeber. Die Zeitwerte müssen mittels C-Routinen besorgt und dann in regelmäßigen Abständen andere Funktionen aufgerufen werden. Diese Aufgabe erledigt bei den MFC der 'Timer'. In der entsprechenden Funktion muß dann die Variablenänderung vorgenommen werden. Es kann so ein beliebiger Interpolator erstellt werden (siehe Programmbeispiel 3, Seite 15).

Als abschließende Bewertung kann zu den verschiedenen Implementierungen der Interpolatoren festgestellt werden, daß Java3D und VRML ein sehr ähnliches Angebot an vordefinierten Interpolatoren liefern und sich im wesentlichen die Implementierungen der Interpolatoren durch die Aktivierungsfunktion von Java3D unterscheiden. Es zeigt sich, daß die Interpolatoren von Java3D durch die Nutzung der Aktivierungsfunktion umfangreicher genutzt werden können und weniger Aufwand darin besteht, komplexere Aktivierungsfunktionen einzusetzen. Bei OpenGL, Java3D und VRML ist es grundsätzlich möglich, einen neuen/eigenen Interpolator zu

erstellen. Dabei stellen alle 3D-Graphik-Schnittstellen gut programmierbare, schnelle und konzepttreue Möglichkeiten bereit.

### 6.3 Unterschiede bei den Ereignissen

Im 2. Kapitel wurden schon die verschiedenen Sensoren von VRML und die verschiedenen Ereignisse von Java3D aufgeführt. Nun werden alle Sensoren und Ereignisse gegenübergestellt, und dabei Vorschläge gemacht, wie die nicht direkt vorhandenen Gegenstücke erzeugt werden können und welche Einschränkungen dabei existieren.

OpenGL hat natürlich selber keine solchen Mechanismen, aber mittels der Programmiersprache C, ist es möglich, jede Art von Sensoren oder Ereignissen nachzuahmen. Es entsteht bei der Nachahmung mit C keine Einschränkung. Einzig und allein die Performance kann bei C schlechter sein, wenn keine Optimierungen, wie bei Java3D gemacht werden; wenn also z.B. keine Scheduleregionen eingeführt werden. Wie die einzelnen Ereignisse in OpenGL umgesetzt werden können, ist im wesentlichen davon abhängig, wie der 'Betrachter' implementiert wird. Damit ist gemeint, wie die Kameraeinstellungen und die Navigation / Interaktion von Betrachter zur 3D-Welt umgesetzt wird. Bei Java3D wurde auch hier einiges anders gestaltet als bei den meisten anderen 3D-Graphik-Schnittstellen. Es wurde bei Java3D versucht, die verschiedene Eingabegeräte, die zur Navigation nötig sind, und Ausgabegeräte, die zum Darstellen der Szene dienen, sowie die Kameraeinstellungen soweit wie möglich durch eine zentrale Klasse, die Sicht-Klasse, zu verbinden, aber trotzdem dabei so allgemein wie möglich zu bleiben. Da es nicht Sinn dieser Diplomarbeit ist, eine 3D-Graphik-Schnittstelle nachzuprogrammieren, wird von einer Umsetzung in OpenGL abgesehen.

Die Tabelle 3 bietet eine Übersicht und eine jeweils kurze Beschreibung wie das Gegenstück erstellt wird. Bei den zu simulierenden Fällen, die in der Tabellenspalte 'Nr.' mit einem '\*' markiert sind, wird nach der Tabelle das Verfahren zum Simulieren des jeweiligen Gegenstücks beschrieben. Dabei wird sich, zur Unterscheidung der Fälle, auf die Tabellenzeilennummer bezogen.

| Nr. | Java3D   | VRML  |
|-----|--|---|
| 1   | Eine vordefinierte Region wird vom Betrachter betreten/verlassen   | Bereichs-Sensor (enter-/exit-Time)  |
| 2   | Ein Verhalten-Objekt ist de-/aktiviert worden (die Scheduleregion ist betreten/verlassen worden)                           | initialize()- und shutdown() Funktion vom Script-Knoten, siehe Kapitel 4.5 (Seite 22) |
| 3   | Die Transformation eines bestimmter Transformation-Objektes hat sich geändert  | Mittels Route-Befehls   |
| 4   | Eine Kollision von einem bestimmten Objekt und einem anderen hat stattgefunden   | Kollisions-Sensor (nur: Betrachter mit beliebigem Objekt)                             |
| * 5 | Eine Kollision findet immer noch statt und mindestens eines der Objekte bewegt sich/eine Kollision findet nicht mehr statt | Kollisions-Sensor (nur: Betrachter mit beliebigem Objekt) mit Script-Knoten           |

Tabelle 3: Java3D- und VRML-Ereignisse

| Nr.  | Java3D   | VRML  |
|------|--|---|
| 6    | Ein Verhalten-Objekt schickt ein Ereignis  | Script-Knoten setzt eigene EventOut-Variable  |
| 7    | Ein AWT-Ereignis hat stattgefunden (z.B. Menüpunkt ist ausgewählt worden)                                  | Über asynchrones Verhalten von Java-Objekten (siehe Kapitel 6.1, Seite 37)                    |
| * 8  | Eine bestimmte Zeit ist verstrichen  | Zeit-Sensor mit Script-Knoten   |
| * 9  | Eine bestimmte Anzahl von <i>Frames</i> ist dargestellt worden   | Zeit-Sensor mit Script-Knoten   |
| 10   | Ein <i>Sensor</i> , also ein Navigationsgerät, betritt/verläßt eine vordefinierte Region                   | Kein Gegenstück, da in VRML Navigationsgeräte nicht für eigene Zwecke abgefragt werden können |
| * 11 | Verhalten-Objekt   | Sichtbarkeit-Sensor   |
| 12   | LoD-Objekt   | LoD-Sensor  |
| 13   | Verhalten-Objekt mit Auswahl-Methode (siehe Kapitel 6.4, Seite 47)   | Zylinder-, Kugeln- und Ebenen-Sensor  |
| * 14 | Verhalten-Objekt mit Auswahl-Methode (siehe Kapitel 6.4, Seite 47)   | Anklick-Sensor  |
| 15   | Verhalten-Objekt mit Auswahl-Methode (siehe Kapitel 6.4, Seite 47)<br>Nur sinnvoll bei einem Java3D-Applet | Anchor = Das Aufrufen einer WWW-Seite durch das Anklicken eines Objektes                      |

Tabelle 3: Java3D- und VRML-Ereignisse

zu 5: Wenn der Kollision-Knoten von VRML eine Kollision entdeckt, dann setzt er seine Kollisionszeit-Variable auf die momentane Zeit. Leider gibt es keine andere Funktionalität als diese. Bei Java3D ist das Signalisieren von Kollisionen ganz anders aufgebaut, es wird nicht nur ein Signal gesendet, wenn eine Kollision auftritt, sondern auch, wenn sie beendet wird. Diese Ungleichheit wird mittels des im folgendem beschriebenen Verfahren versucht aufzuheben.

Es muß mittels eines Script-Knotens eine Variable bereitgestellt werden, die sich merkt, daß dieser Kollision-Knoten schon eine Kollision hatte (siehe Programmbeispiel 11, Seite 45). Wenn bei einem Scriptaufruf die Variable schon gesetzt ist, dann wird eine eventOut-Variable gesetzt, die signalisiert, daß eine Kollision immer noch stattfindet und sich eines der beiden Objekte somit bewegt hat. Um jetzt noch sicherzustellen, daß bemerkt wird, daß eine Kollision nicht mehr stattfindet, muß beim Auftreten einer Kollision ein Kollisions-Sensor für ein (transparentes) Objekt aktiviert werden, das den Betrachter und das kollidierte Objekt umschließt. Bei einer Kollision mit diesem neuen (transparenten) Objekt müßte dann die Kollision mit dem ersten Objekt als beendet signalisiert werden. Dieses Vorgehen ist sehr aufwendig, da kein solides Objekt, wie eine Box, als (transparentes) Objekt bei der zweiten Kollisionsberechnung benutzt

werden kann. Das liegt daran, daß bei soliden Objekten nicht nur der Mantel, sondern das ganze Objekt, das heißt das Volumen, zur Berechnung der Kollision benutzt wird.

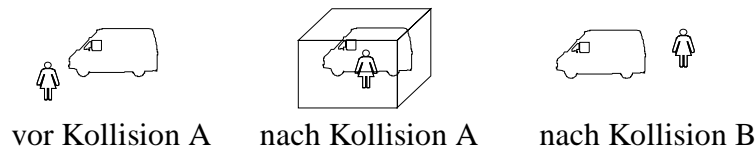


Abbildung 12: Auffinden einer nicht mehr stattfindenden Kollisions in VRML

In Abbildung 12 wird folgender Algorithmus aufgezeigt:

Das erste Bild zeigt die 3D-Welt kurz vor der Kollision mit dem Lastwagen (Kollision A). Die Frau soll dabei die Position des Betrachters darstellen. Bei der Kollision mit dem Lastwagen wird dem Kollisions-Sensor für die transparente Box signalisiert, daß er ab jetzt aktiv ist (2. Bild). Wenn jetzt die Kollision mit der transparenten Box bemerkt wird (die Kollision B, 3. Bild), dann wird dem Kollisions-Sensor der transparenten Box mitgeteilt, daß er ab jetzt inaktiv ist. Außerdem wird signalisiert, daß die Kollision mit dem Lastwagen nicht mehr stattfindet.

Zu erwähnen bleibt noch, daß es auch bei VRML möglich ist, einen Bereich anzugeben, in dem der Betrachter sich befinden muß, damit dieser Kollisions-Sensor überhaupt überprüft, ob eine Kollision stattfindet. Dieser Bereich wird durch die `bboxSize`-Variable angegeben. Es besteht also die Möglichkeit in VRML, eine Art Scheduleregion für diesen Sensor anzugeben (siehe Kapitel 6.5, Seite 48).

```
...
DEF Begrenzung Collision
{ bboxSize 3 3 3 ... } ...
```

Programmbeispiel 11: VRML und Kollisionen

zu 8: Da bei VRML der Zeit-Sensor kein gesondertes Ereignis schickt, wenn eine bestimmte Zeit verstrichen ist, muß auch hier wieder ein Script-Knoten dafür sorgen, daß zu einem bestimmten Zeitpunkt reagiert wird. Es wird dafür einfach immer der Bruchteilwert des Zeit-Sensors auf seine Größe überprüft, und wenn sie einen bestimmten Wert überschritten hat, dann wird von diesem Script-Knoten ein Ereignis erzeugt. Das wäre natürlich auch mittels EAI oder durch asynchrones Verhalten möglich. Es ist zu beachten, daß das Zeitverhalten, bei Benutzung des EAI oder des asynchronen Verhaltens, nicht mehr zwingend dasselbe sein muß, wie bei der ersten Methode. Dies liegt daran, daß die Zeitwerte von dem VRML-Browser und dem Java-Interpreter nicht gleich sein müssen.

zu 9: Hier wird die Tatsache genutzt, daß die Execution-Engine in VRML vor jeder Erstellung eines Abbildes einen Zeit-Sensor aktiviert. Wenn also in einem Script-Knoten eine Variable mitzählt, wie oft der eigene Script-Knoten von einem Zeit-Sensor aufgerufen worden ist, dann kann bei Erreichen eines vorgegebenen Wertes ein Ereignis erstellt werden, in dem eine event-Out-Variable des Script-Knotens auf 'TRUE' gesetzt wird.

zu 11: Der Sichtbarkeit-Sensor von VRML ist in Java3D nicht ganz so einfach zu implementieren. Es muß eine Verhalten-Klasse geschrieben werden, die bei jeder Abbilderstellung aufgerufen wird. Sie holt sich dann die Orientierung und den Standpunkt des Sicht-Objektes, sowie den Sichtradius. Aus diesen drei Angaben wird eine viereckige Pyramide erstellt, die ihre Spitze am Augpunkt hat und sich in die Richtung der eigenen Orientierung verbreitert. Die

Höhe der Pyramide ist dabei beliebig. Der Einfachheit halber sollte die Höhe eine Koordinateinheit betragen. Dann wird aus der eigenen Position und der des Augpunktes eine Linie erstellt und der Schnittpunkt von der Linie mit der Pyramide berechnet. Liegt ein Schnittpunkt vor, der nicht gleich dem Augpunkt ist, so ist das Objekt sichtbar (siehe Abbildung 13, Seite 46). Dieses Vorgehen ist vereinfacht und kann zu Fehlern führen, wenn das Objekt sehr groß ist (siehe Abbildung 13, Seite 46). In diesem Fall sollte ein Quader um das Objekt gelegt werden, von dessen Ecken aus dann eine Pyramide zu dem Augpunkt gebildet wird. Durch eine Schnittberechnung der Quader-Pyramide und der Sichtpyramide wird dann überprüft, ob das Objekt sichtbar ist, oder nicht. Um den Sichtbarkeit-Sensor von VRML zu implementieren, ist der einfache Algorithmus vollkommen ausreichend, da der Sichtbarkeit-Sensor von VRML auch nur auf das Zentrum des Objektes reagiert und nicht auf das Objekt an sich. Weiterhin reagiert der Sichtbarkeit-Sensor von VRML auch, wenn das interessierende Objekt von anderen Objekten verdeckt wird. Es wird also nicht überprüft, ob ein anderes Objekt zwischen dem interessierenden Objekt und dem Betrachter liegt. Da der einfache, oben beschriebene Algorithmus genauso arbeitet, ist mit ihm ein Gegenstück zu dem Sichtbarkeit-Sensor geschaffen.

Die Auswahl-Methoden sind in diesem Fall nicht anwendbar, da sie genau konträr arbeiten. Die Auswahl-Methoden überprüfen, welches Objekt auf einem Strahl liegt, welcher vom Augpunkt aus geht. Es könnte natürlich jeder Strahl erzeugt werden, der in der oben beschriebenen Augpunkt-Pyramide liegt, bzw. eine Teil von ihnen. Es wäre somit mit einer groben Näherung durch ein Gitternetz zu arbeiten. Dieses Vorgehen ist aber nicht ratsam, da entweder das Netz von Strahlen zu grob ist, oder der Aufwand, für die vielen Schnittberechnungen zu groß wird. Java3D bietet aber die Möglichkeit die Strahl-in-Pyramide-Berechnung zu übernehmen. Die entsprechende Funktion lautet 'intersect' und gehört zu der BoundingPolytope-Klasse. Interessant ist, daß die VRML-Java3D-WorkingGroup dieses VRML-Feature bisher noch nicht umgesetzt hat (siehe Anhang B).

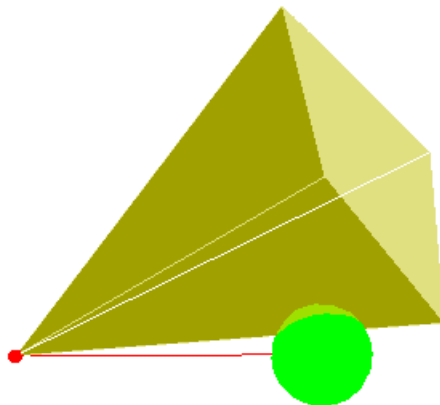


Abbildung 13: Ein einfacher Sichtbarkeit-Sensor-Algorithmus

zu 14: Der Anklick-Sensor ist wie die Drag-Sensoren (Sphäre-, Zylinder- und Ebenen-Sensor) zu realisieren. Bei VRML gibt es aber bei dem Anklick-Sensor auch ein 'Ereignis', das signalisiert, daß der Mauszeiger auf ein Objekt zeigt. Dieses 'Ereignis' ('isOver') wurde in der Mailinglist von Java3D [ML-J3D] öfter angesprochen und in dem offiziellen VRML-Browser von Java3D noch nicht implementiert. Dies liegt daran, daß eine ständige Überwachung der Maus benötigt wird. Auch wenn sie nichts anklickt, muß bei jeder Bewegung der Maus überprüft werden, ob die in Frage kommenden Objekte 'unterhalb' der Maus liegen. Das führt natürlich zu nicht unerheblichen Berechnungsaufwänden. Aber ansonsten ist die Implementierung dieses Ereignisses ohne weiteres mit Hilfe der Maus-bewegt-sich-AWT-Ereignis-Aufweckme-

thode zu bewerkstelligen.

Anhand der Tabelle ist ganz gut zu erkennen, daß in VRML viele Dinge schon implementiert sind und nur noch benutzt werden müssen. Bei Java3D sind aber die grundlegenden Funktionen, wie die Überwachung der Erstellung einzelner Abbilder, gegeben. Daraus ergibt sich für VRML, daß nicht alle Dinge, oder nur mit einigen Tricks, umsetzbar sind. Hingegen ist bei Java3D Programmiererfahrung nötig, um einige Umsetzungen effektiv zu gestalten. So ist die *isOver*-Funktion des Anklick-Sensors zwar schnell geschrieben, aber dann auch sehr rechenaufwendig zur Laufzeit. Wenn diese Funktion effektiv arbeiten soll, muß sie mit Programmieraufwand und Programmiererfahrung rechenaufwandoptimiert erstellt werden. Dafür sind in Java3D die direkten Wege zum Programmziel nutzbar, das heißt es muß nicht mit Tricks wie bei VRML gearbeitet werden.

#### 6.4 Auswahl-Methoden und Drag-Sensoren

In diesem Unterkapitel wird es um die Möglichkeiten gehen, die Java3D und VRML bereitstellen, damit mit einem Eingabegerät, z.B. der Maus, Objekte angefaßt und ihre Transformationswerte durch sogenanntes *dragging* verändert werden können. Der Begriff 'drag' bedeutet dabei, daß ohne Angabe von Werten, also nur durch Mausbewegung, eine Veränderung mit dem 'gezogenen' Objekt vollzogen wird. Für das dragging typische Transformationswert-Veränderungen sind: Verschieben oder Drehen des Objektes. In VRML übernehmen die Drag-Sensoren diese Aufgaben. In Java3D existieren für die Realisierung des dragging Auswahl-Methoden. Diese Auswahl-Methoden dienen zur Bestimmung der Objekte, die mit einem Strahl, der vom Sicht-Objekt ausgeht, einen Schnittpunkt haben. Wie diese Methoden genutzt werden können, um das dragging zu gestalten, wird erklärt werden. Um dem Umstand abzuweichen, daß ein Java3D-Programmierer eine Art Drag-Sensor implementieren muß, werden mit dem Utils-Paket einige Klassen bereitgestellt, die ähnliche Funktionen aufweisen. Als erstes wird in diesem Unterkapitel der Aufbau dieser Klassen beschrieben, danach erfolgt ein Vergleich zwischen diesen Klassen und den Drag-Sensoren.

Die Klassen gehören zu dem `com.sun.j3d.utils.behaviors`-Paket und heißen: `PickRotateBehavior` und `PickTranslateBehavior`. Sie beruhen auf der `PickMouseBehavior`-Klasse, die im wesentlichen die von allen genutzten Variablen, wie Transformation-Objekte oder X/Y-Positionen der Maus, initialisiert, liest und schreibt. Es existiert also in diesem Paket kein Zylinder-Drag-Sensor wie in VRML. Der Zylinder-Drag-Sensor ist ein Drag-Sensor, der eine Rotation auf nur einer Dimension zuläßt. Um eine `PickCylinderRotateBehavior`-Klasse zu schreiben, sollte am besten die `PickRotateBehavior`-Klasse beerbt und die Rotations-Richtung auf die der X-Achse beschränkt werden.

Die `PickRotateBehavior`-Klasse ist so aufgebaut, daß sie bei jedem Maus-Ereignis aufgerufen wird und als erstes die nicht benötigten Fälle, wie 'Alt-Taste ist gedrückt' oder 'Maus wurde nur bewegt' etc., ausgeklammert werden. Dann wird sich die Mausposition gemerkt und die `PickClosest`-Methode aufgerufen, die eine Referenz auf das 'angefasste' Objekt zurückgibt. Anschließend wird mit Hilfe eines `MouseRotate`-Objektes (siehe nächster Absatz) die Differenz der Mausposition zur letzten Mausposition in X/Y-Rotationen umgerechnet und der Translationwert dementsprechend geändert.

Die `MouseRotate`-Klasse ist dafür zuständig, die Differenz der Mausposition zu berechnen und diese Werte mit Faktoren zu multiplizieren. Dann wird der alte Translationwert besorgt und mit Hilfe der Rotationswerte umgerechnet. Die neuen Werte werden dann gespeichert. Bei den oben erwähnten Faktoren handelt es sich um Werte, welche die Rotationsstärke angeben, leider existiert keine Methode, um die Faktoren getrennt nach X und Y-Rotation zu setzen, sondern



nur eine Methode, die beide Werte denselben Wert zuweist, was die Notwendigkeit einer neuen Klasse bedeutet, um einen Zylinder-Drag-Sensor zu implementieren.

Die PickTranslateBehavior-Klasse ist genauso aufgebaut wie die PickRotateBehavior-Klasse. Daß bei diesen Maus-Verhaltens-Klassen immer nur die X- und Y-Dimension berücksichtigt wird, liegt in der Natur der Maus, die nur zwei Dimensionen hat und keine dritte, wie die geometrischen Objekte. Um eine Rotation oder eine Translation in die fehlende Dimension zu machen, müßte mittels eines Maus-Knopfes oder einer Taste, die Mausektion in zwei Kategorien unterschieden werden. Die erste Kategorie sollte sich dann so wie oben beschrieben verhalten und die zweite formt die Aktionen z.B. von der X/Y-Dimension auf die X/Z-Dimension um. Das heißt, die Eingaben mit der Maus werden normal abgelesen, aber dann nicht auf die X/Y-Werte, sondern auf die X/Z-Werte des Objektes angewandt.

Der VRML97-Standard sieht neben der Maus auch den 'Wand' vor. Der Wand ist ein Stab, der drei Richtungsdimensionen hat, sonst aber dieselbe Funktion wie eine Maus übernimmt. Dies bedeutet für Java3D-Programmierer, daß dieses Eingabegerät erst einmal mittels der Eingabegeräte- und der Sensor-Klasse abgefragt werden muß (siehe Kapitel 5.2, Seite 28), und daß dann zusätzlich ein Pendant zu den Maus-Verhalten-Klassen geschaffen werden muß.

Die Möglichkeiten der vorhandenen Maus-Verhalten-Klassen sind also sehr eingeschränkt, daher müssen in der Regel neue Klassen für die eigenen Bedürfnisse geschrieben werden. Allerdings bildet das kein größeres Problem, und es ist dadurch eine individuelle Behandlung des dragging möglich. Wenn also, wie in VRML, nur eine Rotation oder eine Translation auf der X/Y-Dimension benötigt wird, dann bieten die Klassen des Utils-Paket denselben Programmierkomfort wie VRML. Das heißt in Java3D muß, um ein Drag-Sensor zu erstellen, ein Objekt der entsprechenden Klasse per Konstruktor erzeugt werden und dann dieses Verhalten-Objekt in den entsprechenden Teilbaum des hierarchischen 3D-Welt-Baumes gehängt werden. Allerdings ist die Umsetzung des Zylinder-Drag-Sensors nicht direkt vorhanden. Daß die Java3D-Entwickler die MouseRotate-Klasse so geschrieben haben, daß keine Möglichkeit besteht, dem einen Dimension-Faktor einen anderen Wert zuzuweisen als dem zweiten Dimensionen-Faktor, ist nicht sehr nützlich. Es ist davon auszugehen, daß bei der Entwicklung der 'Setze-Faktor'-Methode die Vorstellung, daß eine Rotation in beide Richtungen gleich stark sein soll, eine Rolle gespielt hat. Denn eine Kugel läßt sich besser drehen, wenn die Rotation in beide Richtungen gleich stark ist. Leider wurde dabei übersehen, daß es manchmal nützlich ist, daß eine Rotation nur in eine Richtung gewünscht ist. Das mit Hilfe von C, auch OpenGL es möglich ist, dieses Verhalten zu simulieren, sei hier noch erwähnt. Die Programmierung der Verhalten-Klassen von Java3D entspricht dem Vorgehen bei OpenGL. Die Maus-Überwachung sollte dabei von dem Windows-Messaging übernommen werden. Da in C / OpenGL aber keine Auswahl-/Strahl-Objekt-Schnitt-Berechnung-Methoden existieren, wird die Implementierung recht aufwendig oder sehr rechenaufwendig, wenn nicht auf Optimierungen geachtet wird.

## 6.5 Scheduleregionen und Bounding-Boxes

Nachdem es in den Kapiteln 6.1 bis 6.4 um die Programmierung von Animationen ging, soll es in diesem Unterkapitel vor allem um eine Optimierung des Laufzeitverhaltens gehen, die bei dem Einsatz von Animation zum Zuge kommt. Mit dieser Optimierung ist das culling von nicht benötigten Animation gemeint. Nicht benötigte Animationen sind solche, die für den Betrachter im Moment nicht erkennbar sind. Wenn sich zum Beispiel die Farbe eines Objektes ändert, aber der Betrachter soweit davon entfernt ist, daß er diese Änderung nicht bemerkt, dann braucht diese Änderung nicht gemacht werden. Ein noch besseres Beispiel ist allerdings die Kollisionsüberwachung. Es macht keinen Sinn, ein Objekt darauf zu überprüfen, ob es eine Kollision mit dem Betrachter hat, wenn dieser sich am anderen Ende der 3D-Welt befindet. Um einen Vergleich zwischen den Konzepten, die VRML und Java3D zu diesem Verfahren

haben, zu machen, wird erst einmal ein kurzer Überblick über die Möglichkeiten von Java3D geschaffen und dann eine nähere Betrachtung der Möglichkeiten von VRML zu diesem Verfahren gemacht. OpenGL wird diesmal komplett aus der Betrachtung herausgenommen, da es dieses Optimierungsverfahren überhaupt nicht unterstützt und eine effektive Implementierung in C ein eigenes Diplomarbeitsthema bilden würde.

Im Kapitel 5.3 (Seite 30) wurde erwähnt, daß in Java3D zu jedem Verhalten-Objekt Schedulingregionen angegeben werden müssen. Diese Angabe ist zwingend [Sun 98-1, Kapitel 9.3]. Dabei kann diese Region eine Spähren-, Quader- oder eine Polytope-Form haben. Die Region hängt, wie immer in Java3D, an einem Transformation-Objekt, so daß bei einer Änderung der Transformationswerte des Transformation-Objektes sich die Schedulingregion-Position mitverändert. Zum Beispiel: Es existiert ein Objekt, das sich im Raum bewegt. Mit diesem Objekt sollen aber bestimmte Aktionen nur dann erlaubt sein, wenn der Betrachter nahe genug an dem Objekt ist. Diese Vorgehensweise ist sehr gut, da dadurch einige unnötige Berechnungen wegfallen, wenn sie sowieso nicht benötigt werden. Voraussetzung dafür ist allerdings, daß die Schedulingregionen nicht zu groß angegeben werden, denn dann würden die Berechnungen evtl. immer gemacht werden und somit keine Optimierung geschaffen werden.

Einem Anfänger in VRML scheint es, daß VRML solch ein Konzept nicht besitzt und daher schlechter für Animationen geeignet wäre. Dem ist aber nicht so, denn nach intensiver Studie der VRML-Spezifikation fällt auf, daß es sogenannte Bounding-Boxes gibt. Diese Bounding-Boxes haben unter anderem zur Aufgabe, den Renderer des VRML-Browsers zu entlasten, indem ein bestimmter Teilbaum des Transformationsbaumes von VRML nur zur Berechnung herangezogen wird, wenn der Betrachter nahe genug an den Objekten dieses Teilbaumes ist. Diese Bounding-Boxes sind sozusagen ein LoD-Knoten auf an/aus-Basis. Wobei der normale LoD mit einer Kugel und nicht einem Quader als begrenzendes Objekt arbeitet, da mit dem Abstand zum Betrachter gerechnet wird. Es gibt zwei Arten von Knoten, welche die Bounding-Boxes zu ihren Variablen zählen, denn Bounding-Boxes sind nichts anderes als Variablen von VRML-Knoten. Die eine Art von Knotentypen sind Gruppen-Knoten (Transform, Group, Inline) und die andere Art sind einige Sensoren (Billboard, Collision, Anchor) von VRML.

Der Nutzen der Bounding-Box soll nun betrachtet werden. Da die Bounding-Box einen ganzen Teilbaum aus der Berechnung des Renderers nehmen kann, wenn der Betrachter nicht nahe genug ist, könnte überlegt werden, einen Sensor und die zugehörigen ROUTE-Befehle innerhalb eines solchen Teilbaumes zu platzieren. Aber leider ist das Konzept der Bounding-Boxes auf Gruppen-Knoten-Ebene und auch das LoD-Konzept auf den Renderer begrenzt. Das bedeutet, daß alle ROUTE-Befehle, alle Sensoren, alle Interpolatoren und alle Script-Knoten immer aktiv sind. Das heißt, daß die Bounding-Boxes der Gruppen-Knoten bei der Verminderung des Rechenaufwandes bei nicht nötigen Sensorberechnungen nicht von Nutzen sind. Aber die Sensor-Knoten, wie Kollisions-Sensor, Anchor-Sensor und Billboard-Sensor, haben alle Bounding-Boxes. Es sind also auch zwei der drei rechenaufwendigsten Sensoren dabei, der Kollisions-Sensor und der Billboard-Sensor. Daß der Sichtbarkeit-Sensor nicht dabei ist, ist zu verstehen, da die Sichtbarkeit-Frage in der Regel immer wichtig ist, egal wie weit der Betrachter von dem Objekt entfernt ist. Warum allerdings der Anchor-Sensor und nicht auch der Anklick-Sensor sowie die Drag-Sensoren eine Bounding-Box haben, ist nicht verständlich. Bounding-Boxes sind zwar somit in VRML möglich, aber sie werden, wenn sie nicht extra angegeben werden, als unendlich groß angesehen und tragen dann nicht zur Optimierung bei. So kann es sein, daß ein 3D-Welten-Ersteller diese Möglichkeit evtl. gar nicht kennt, oder sie nicht nutzt, weil er nicht dazu gezwungen wird, die Bounding-Boxes zu nutzen. Das kann zwar nicht die Fähigkeiten von VRML schmälern, aber die Performanz einer gegebenen 3D-Welt.

VRML bietet zwar keine Möglichkeit allen Sensoren eine Art Aktivierungsraum zu geben, aber es bietet die Möglichkeit bestimmte Interpolatoren oder Script-Knoten nur dann auszuführen, wenn sich der Betrachter in einem bestimmten Bereich befindet. Das Prinzip, auf den das

Animationsmodell von VRML aufgebaut ist, ist das routing. Wenn dynamisch Verbindungen verändert werden können, dann wäre es möglich, eine Verbindung von einem Sensor zu einem Interpolator- oder Script-Knoten erst dann zu erstellen, wenn der Betrachter in einem bestimmten Bereich ist. Die Verbindung könnte dann wieder gelöscht werden, wenn der vorgegebene Bereich vom Betrachter wieder verlassen würde. Mit diesem Prinzip würde ein Großteil der Berechnungen eingespart werden, die nicht nötig sind, weil der Betrachter die Effekte nicht sehen kann. Es würde also genau nach dem Scheduleregionen-Prinzip vorgegangen werden, allerdings mit einigen Performanzeinschränkungen, denn dieses Prinzip benötigt einen zusätzlichen Sensor, nämlich einen Bereich-Sensor, und das dynamische Erstellen einer Verbindung verbraucht auch einige Rechenleistung. Es folgt eine Beschreibung des Algorithmus:

Der Bereichs-Sensor überwacht die Position des Betrachters und speichert Verlassen und Betreten von dem vorgegebenen Bereich. Es existiert eine Verbindung von dem Bereichs-Sensor zu einem Verbindung-Erstell-Script-Knoten, der entweder eine Verbindung erstellt oder löscht. Die Verbindung, die erstellt bzw. gelöscht wird, ist in der Regel eine, die von einem beliebigen Sensor zu einem beliebigen Interpolator- / Script-Knoten geht. Da der Interpolator / der Script-Knoten nur dann aktiviert wird, wenn eine bestehende Verbindung zwischen ihm und dem Sensor besteht, ist so gewährleistet, daß bestimmte Aktionen nur dann ausgeführt werden, wenn der Betrachter sich in einem bestimmten Bereich befindet. Das Ganze ist im Programmbeispiel 12 umgesetzt, wo ein Farben-Interpolator dann 'aktiviert' wird, wenn der Betrachter nahe vor dem Würfel steht. Für Insider: Das Beispiel soll nur die Vorgehensweise darstellen, es hätte natürlich genau dieses Beispiel einfacher mit dem direkten Verbinden von den Start- und End-Zeitpunkten des Bereichs-Sensors mit denen des Zeit-Sensors erstellt werden können. Die Erklärung des Beispiels ist durch die Beschreibung des Algorithmus von oben und den Kommentaren im Programmcode schon gegeben.

```
... # Definiere Box (BoxCol) und Bereichs-Sensor (PS)
... # Definiere Zeit-Sensor (Uhr) und Farben-Interpolator (CI)
DEF Verwalt Script {
  eventIn SFBool active # Die Benachrichtigungs-Variable
  field SFNode ci USE CI # Eine Referenz auf den Farb-Interpolator
  field SFNode uhr USE Uhr # Eine Referenz auf den Zeit-Sensor
  directOutput TRUE # Nötig, für die globale Änderung am Routegraph
  url "javascript: function active(activ,zm) {
    if (activ)
      Browser.addRoute(uhr,'fraction_changed',ci,'set_fraction');
    else
      Browser.deleteRoute(uhr,'fraction_changed',ci,'set_fraction');
  }"
}
ROUTE PS.isActive TO Verwalt.active
ROUTE CI.value_changed TO BoxCol.set_diffuseColor
```

Programmbeispiel 12: Farben-Interpolator auf Abruf (Bereichsaktivierung)

Nachdem doch VRML einige Möglichkeiten bietet, Aktivierungsbereiche zu benutzen, stellt sich die Frage, ob diese Möglichkeiten auch so gut wie in Java3D sind. Diese Frage läßt sich zwar schnell mit einem 'Nein' beantworten, aber die detaillierte Begründung bedarf einer ausführlicheren Darstellung. Als erstes sei hier erwähnt, daß bei VRML grundsätzlich nur Quader als Bereiche möglich sind. In Java3D waren zusätzlich noch Kugeln und Polytope möglich. Es sind diese Zusatzformen nicht so dringend nötig, um Optimierungen durchzuführen, im Gegenteil, wenn sie genutzt werden, dann kann es zu mehr Rechenaufwand kommen, da

schneller berechnet ist, ob sich ein Punkt in einem Quader befindet, als in einem Polytop. Bei VRML ist es außerdem nicht ganz so einfach, diese Optimierungsmöglichkeiten zu finden, da kein klares Konzept vorliegt. Das geht soweit, daß einem Programmierer überhaupt nicht klar ist, daß es diese Optimierungsmöglichkeit gibt. Java3D hingegen zwingt den Programmierer dazu, diese Art von Optimierung zu nutzen. Als letzter Kritikpunkt sei noch erwähnt, daß die Optimierungsart mit den Verbindungserstellungen und -löschungen von der Performance nicht mit der von Java3D konkurrieren kann. Dies liegt daran, daß bei VRML ein Trick genutzt wird, also keine direkte, bzw. zentrale Lösung, und daß der Scheduler in Java3D durch seine beiden Baumstrukturen (siehe Kapitel 5.3, Seite 30) sogar Optimierungen verfolgt, wenigstens in der Theorie, in der Praxis kann es noch nicht getestet werden.

## 6.6 Der Scheduler und der Route-Mechanismus

Dieses Unterkapitel soll die Unterschiede der beiden zentralen Mechanismen im jeweiligen Animationsmodell darstellen. Daraus ergeben sich auch die Probleme bei der Entwicklung eines VRML-Browser mittels Java3D (siehe Kapitel 6.7, Seite 53). Es werden zu Beginn noch einmal die beiden Konzepte vorgestellt und dann gezeigt, wie mit Java3D der Route-Mechanismus implementiert werden kann. In diesem Zusammenhang ist es nicht notwendig, OpenGL mit zu berücksichtigen, da dort weder ein Scheduler noch ein Route-Graph vorhanden und daher kein Vergleich möglich ist.

Als erstes werden die Eigenschaften des Route-Mechanismus noch einmal aufgeführt (Näheres siehe Kapitel 4.2, Seite 18). Es werden mittels des Route-Mechanismus Variablen in der Form miteinander verbunden, daß eine Variablenänderungen auf der Quellen-Seite dieselbe Variablenänderung auf der Ziel-Seite bewirkt. Es werden also keine Ereignisse an Interessenten verteilt, wie bei Java3D, sondern Variablenwerte an vorgegebene Knoten. Der Unterschied zwischen einer Ereignis-Weitergabe und einer Variablenweitergabe liegt darin, daß bei dem Variablenweitergabe-Verfahren die Weitergabe direkt an eine Variablenänderung gebunden ist. Bei den VRML-Script-Knoten-Variablen kann nun diese Variablenänderung als 'nur' Ereignis-Weitergabe genutzt werden, indem der Variablenwert selber nicht weiter beachtet wird. Wenn eine Script-Knoten-Variable durch den Route-Mechanismus geändert wird, wird immer eine Funktion des Script-Knotens aufgerufen und der Variablenwert als Parameter übergeben. Das heißt, es wird genau wie bei dem Scheduler-Mechanismus von Java3D eine Funktion aufgerufen; daß bei dem Funktionsaufruf noch ein Wert mitgeliefert wird, ist in diesem Fall nur zweitrangig. Auch bei den anderen Knoten-Typen, wie Interpolatoren oder Sensoren, kann immer das Vorgehen mittels des Script-Funktion-Aufrufes als Umweg eingesetzt werden, um eine Ereignis-Weitergabe zu simulieren.

Ein Nachteil ergibt sich bei dem Route-Mechanismus aus dem im Kapitel 6.5 (Seite 48) beschriebenen Fehlen der zentralen Regelung von Scheduleregionen, bzw. daß nicht alle Sensoren die Möglichkeit bieten eine Scheduleregion anzugeben. Es wurde dort gezeigt, daß ein ähnlicher Mechanismus dezentral, das heißt für den einzelnen Sensor, realisiert wurde.

Andererseits bietet der Route-Mechanismus die Möglichkeit, Verbindungen von einer 'beliebigen' Stelle im Programm aus zu verändern. Das heißt ein Script-Knoten kann eine Verbindung löschen oder erstellen, bei der keine Variable seines eigenen Knotens Ziel oder Quelle ist (siehe Programmbeispiel 12, Seite 50). Genau das ist der Knackpunkt, bei Java3D ist dieses Vorgehen nicht möglich [Sun 98-1, Anhang F 2.1]. Dies ist leicht einzusehen, denn die wakeupOn-Methode der Verhalten-Klasse ist vom Aufruf-Type 'protected'. Das heißt, diese Methode ist nicht von anderen Objekten aus aufrufbar. Da es keinen anderen Mechanismus in Java3D gibt, um die Einträge des Scheduler zu ändern, ist VRML mit diesem Mechanismus ein Vorteil gegeben.

Ein weiteres Handikap ist, daß in VRML nicht nur Sensoren 'Ereignisse' verschicken können, es kann auch jeder andere Knoten ein 'Ereignis' verschicken. Wenn also der Interpolator ein 'Ereignis' zu einem Script-Knoten schickt, dann kann dies nicht mit dem normalen Scheduler-Ereignissen gelöst werden, da die Scheduler-Ereignisse mit den Sensor-Ereignissen von VRML 'gleich' zu setzen sind (siehe Kapitel 6.3, Seite 43). Dieses Problem kann allerdings mit dem postId-Mechanismus des Schedulers behoben werden. Wie im Kapitel 5.3 (Seite 30) erwähnt wurde, ist dieser Mechanismus so aufgebaut, daß ein Verhalten-Objekt ein Ereignis mittels der eigenen postId-Methode erzeugt werden kann. Es wird als Parameter eine Integerzahl mit übergeben, die es ermöglicht, verschiedene postId-Ereignisse zu erstellen. Das heißt, es ist möglich eine Verbindungen durch die einmalige Vergabe einer Integerzahl eindeutig zu definieren.

Nachdem gezeigt worden ist, daß es keinen vergleichbaren Mechanismus in Java3D zum Route-Mechanismus gibt, soll jetzt eine Methode vorgestellt werden, die es trotz dieses Umstandes erlaubt, einen VRML-Browser in Java3D zu erstellen. Die Methode beruht darauf, daß VRML-Nachrichtenketten durch einen Sensor-Knoten eingeleitet werden. Eine Variable, in der für jeden Route-Befehl ein Vektor steht, realisiert den Route-Graphen. Ein Vektor besteht dabei aus den Anfangspunkt: Quellvariable und dem Endpunkt: Zielvariable. Diese Route-Variable ist allen VRML-Knoten-Objekten zugänglich. Wenn ein Ereignis stattfindet, dann wird das entsprechende VRML-Sensor-Objekt mittels des Schedulers aufgerufen. Die Weitergabe des Ereignisses ist nun Aufgabe des Sensor-Knoten-Objektes, in der Route-Variable nachzuschauen und dann kein Java3D-Ereignis erstellt, sondern einfach bei dem entsprechenden VRML-Knoten-Objekt die Variable ändert und dann weitere Berechnungen anstößt, indem eine Art processStimulus-Methode (siehe Kapitel 5.3, Seite 30) des VRML-Knoten-Objektes aufgerufen wird. Diese processStimulus-Methode der anderen Knoten-Objekte besitzt auch diesen Mechanismus. Das heißt, es wird in der Route-Variable nachgeschaut, um die nächste Verbindung in der Form zu realisieren, so daß die Variable geändert und der entsprechend simulierte VRML-Knoten 'aktiviert' wird. Dieses Vorgehen erfüllt die VRML-Anforderung für die Realisierung des Route-Mechanismus, da in VRML eine Nachrichtenkette zu ein und demselben logischen Zeitpunkt stattfindet. Es wird also eine Nachrichtenkette durch einen Sensor, der durch einen Schedule-Eintrag realisiert ist, eingeleitet und dann die Nachrichtenkette abgearbeitet. Das heißt, es wird kein zusätzlicher Scheduler-Eintrag benötigt, da alle Route-Verbindungen sequentiell ohne Unterbrechung abgearbeitet werden. Mit Hilfe dieses Verfahrens ist es dann auch möglich, dynamische Verbindungen, wie in VRML, zu erstellen und zu löschen. Bei dem Vorhandensein gleicher Quellvariable wird immer die Verbindung als erstes verfolgt, die auch als erstes in den Route-Graphen eingetragen wurde. Da es möglich ist, Verbindungen von einer Quellvariable zu zwei verschiedenen Zielvariablen zu haben, muß ein Stack dafür sorgen, daß auch alle Zweige des Route-Graphen abgearbeitet werden. Es ist zwar somit keine sequentielle Verfolgung des Route-Graphen möglich, aber die Nachrichtenkette wird immer noch sequentiell durchlaufen.

Es zeigt sich, daß der Route-Mechanismus relativ einfach in Java3D realisiert werden kann und daß die Möglichkeiten von Java3D mittels des Scheduler in eine andere Richtung gehen. Der Scheduler ermöglicht dem Programmierer die Bereitstellung von Sensoren, hingegen dient der Route-Mechanismus eher dazu, eine ununterbrochene sequentielle Folge von Nachrichten zu erstellen. Es wird nicht zwischendurch die Nachrichtenfolge unterbrochen und die Kontrolle an einen Scheduler übergeben, sondern stur nach einem vorgegebenen Muster eine Nachricht weiter geleitet. Die Möglichkeit, daß ein nicht deterministischer Zustand entsteht, ist dabei nicht gegeben. Die beiden zentralen Animations-Mechanismen, der Scheduler und der Route-Mechanismus, sind also in keiner Weise zu vergleichen, sie dienen verschiedenen Zwecken. Daraus folgt, daß die beiden Animationsmodelle vollkommen unterschiedlich aufgebaut sind. Es gibt also nicht nur den Unterschied von Knoten (VRML) zu Objekten (Java3D) und den, daß in

Java3D mehr verschiedene Objekte möglich sind und diese differenzierter eingesetzt werden können, sondern auch, daß die beiden Vermittler zwischen den einzelnen Objekten / Knoten eine gänzlich andere Struktur beinhalten.

## 6.7 VRML-Browser in Java3D

Im 1. Kapitel wurde erläutert, auf welcher Hardware-Abstraktionsebene sich die einzelnen 3D-Graphik-Schnittstellen befinden. Es wurde dort auch geschrieben, daß es Bestandteil dieser Diplomarbeit sein wird, zu untersuchen, ob ein VRML-Browser mit Java3D geschrieben werden kann. In den Kapiteln 6.1-6.6 wurden schon des öfteren Implementierungen von VRML-Komponenten in Java3D vorgestellt. Nun soll dieses Unterkapitel die bisher schon angesprochenen Teilbereiche von der VRML-Implementierung in Java3D zusammenfassen und weitere Bereiche darstellen. Es sei an dieser Stelle bemerkt, daß ein VRML-Browser, der in Java3D geschrieben ist, momentan mit Java3D heruntergeladen werden kann. Er ist von der VRML-Java3D-WorkingGroup [WG-J3D] geschrieben. Dieser VRML-Browser wurde zum ersten Mal mit der Beta1-Version von Java3D zum Herunterladen bereitgestellt. Dieser Browser ist, wie Java3D noch in einem Anfangsstadium und hat daher noch einige Mängel. Einer der gravierenden Mängel ist wohl, daß momentan keine Textur-Graphik-Dateien berücksichtigt werden. Das heißt es werden keine Texturen dargestellt, was in der Regel zum Verlust der Realitätsnähe führt. Was ist eine Wand ohne ein Steinmuster? Eben nur ein Quader oder ein Viereck. Ein Muster auf andere Art zu erstellen, ist zwar möglich, aber bedeutet entweder eine Qualitätseinbuße oder eine Performanceeinbuße. Das heißt, entweder muß durch Farben ein Muster ersetzt werden, oder das Muster durch viele kleine geometrische Objekte dargestellt werden. Das Problem liegt, laut Angaben der Ersteller des VRML-Browsers, an einem Thread-Problem. Es ist ihnen momentan nicht möglich ein Nachladen, der Textur-Graphik-Dateien, zu realisieren. Es geht also nicht um die Darstellung der Textur selber, sondern nur um das Nachladen der Datei (Näheres siehe [ML-J3D]). Eine komplette Liste der Bugs des VRML-Browsers der Beta2-Version ist im Anhang B einzusehen.

Nach den allgemeinen Worten zu dem VRML-Browser, wird sich der Rest dieses Unterkapitels mit dem Animationsmodell von VRML beschäftigen. Es soll insbesondere gezeigt werden, an welchen Punkten es einfach ist, mit Java3D VRML zu implementieren und an welchen Stellen es schwer ist. Als Vergleichssprache zu Java3D dient OpenGL. Eine Übersicht soll die Tabelle 4 bieten. Die Tabelle ist so aufgebaut, daß die VRML-Komponenten aufgelistet werden und dann die Möglichkeit beschrieben wird, wie diese VRML-Komponente in Java3D implementiert werden könnte. Die letzte Spalte gibt dann an, ob die Implementierung mit Java3D leichter oder schwerer ist als mit OpenGL. Die Begründung dafür steht in derselben Spalte. Für eine genauere Erklärung zur Implementierung von VRML-Komponenten in Java3D dienen die entsprechenden Kapitel: Sensoren (6.3, Seite 43), Route-Mechanismus (6.6, Seite 51), Interpolatoren (6.2, Seite 40) und Script-Knoten (6.1, Seite 37).

| Nr. | VRML                 | Java3D                 | leichter / schwerer    |
|-----|----------------------|------------------------|------------------------|
| 1   | Bereichs-Sensor, LoD | 'Bereichs-Sensor', LoD | leichter, da vorhanden |
| 2   | Kollisions-Sensor    | 'Kollisions-Sensor'    | leichter, da vorhanden |
| 3   | Zeit-Sensor          | Zeitgeber              | leichter, da vorhanden |

Tabelle 4: VRML-Browser mit Java3D oder OpenGL

| Nr. | VRML   | Java3D   | leichter / schwerer  |
|-----|--|--|--|
| 4   | Sichtbarkeit-Sensor  | Verhalten-Objekt                                     | leichter, wegen Vorhandensein von Strahl-Objekt-Schnitt-Berechnung |
| 5   | Anklick-Sensor   | Verhalten-Objekt                                     | leichter, wegen leichter Mausabfrage und siehe 4                   |
| 6   | Zylinder-, Kugeln- und Ebenen-Sensor                                 | Verhalten-Objekt                                     | leichter, wegen leichter Transformationswertänderung und siehe 5   |
| 7   | Anchor = Das Aufrufen einer WWW-Seite durch Anklicken eines Objektes | Verhalten-Objekt                                     | leichter, wegen Netzklassen von Java und siehe 5                   |
| 8   | Route-Mechanismus  | globale Variable und Route-Methode für jedes Objekt. | genauso  |
| 9   | Interpolatoren   | Interpolatoren                                       | leichter, da vorhanden   |
| 10  | Normalen- und Koordinaten-Interpolator                               | Verhalten-Objekt                                     | genauso  |
| 11  | Script-Knoten (nur Java)   | Verhalten-Objekt                                     | leichter, wegen 'nur Java'-Einschränkung                           |
| 12  | Script-Knoten (alle anderen)   | Nicht vorgesehen                                     | genauso  |

Tabelle 4: VRML-Browser mit Java3D oder OpenGL

Es ist zu sehen, daß es keinen einzigen Punkt gibt, der aufzeigt, daß es mit Java3D schwerer ist einen VRML-Browser zu erstellen als mit OpenGL und C. Es gibt nur drei Bereiche, bei denen es genauso schwer / leicht ist die VRML-Komponenten mit Java3D zu implementieren, wie mit OpenGL und C.

Eine Sonderrolle spielt dabei der Script-Knoten, da er verschiedene Sprachen unterstützt. Denn die Autoren des VRML-Browsers haben sich auf den Standpunkt gestellt: 'Wer einen VRML-Browser benutzt, der in Java implementiert ist, der benutzt auch nur Java.' Dieses Vorgehen ist zwar konsequent, aber sicher nicht besonders angenehm für VRML-Weltersteller, die ihre Script-Knoten in anderen Sprachen geschrieben haben und sie umschreiben müßten. Dasselbe Problem ergibt sich für den Benutzer. Es macht für ihn keinen Sinn, einen VRML-Browser zu benutzen, der nicht alle VRML-Welten darstellen kann. Aus dieser Sicht bleibt zu hoffen, daß VRML-Weltersteller sich dem anpassen und nur Java in ihren Script-Knoten benutzen, sonst ist die Entwicklung dieses VRML-Browsers nicht sinnvoll gewesen.

Ein letzter Punkt in diesem Vergleich soll der Use-Befehl sein. Durch den Use-Befehl ist es möglich, ganze Teilbäume der VRML-Transformationshierarchie an einen anderen Teilbaum anzuhängen. Durch die Anbindung einer Sprache in VRML ist es möglich jede Variable aus jedem Knoten des Teilbaumes zu erreichen, das heißt, sie kann gelesen und geschrieben werden. Aus diesem Umstand ergibt sich die Zwangsläufigkeit, daß die interne Repräsentation der VRML-Transformationshierarchie bei Java3D genauso sein muß, wie die der VRML-Datei. Es

ist also nicht möglich, eine andere (optimiertere) Transformationshierarchie zu benutzen oder die Abspeicherung der Variablen anders zu gestalten, als es durch die VRML-Datei vorgegeben ist. Gerade die Möglichkeit, die Java3D bietet, um die Transformationshierarchie zu optimieren, der compiled-retained mode, läßt sich bei der Umsetzung eines VRML-Browsers nicht nutzen, da bei einem VRML-Transformationshierarchiebaum alle Variablen immer geändert werden dürfen, aber bei Java3D die Unveränderbarkeit von einigen Variablen zu Optimierungen führt.

Dieses Unterkapitel hat gezeigt, daß eine Umsetzung von VRML in Java3D möglich ist, und daß diese Umsetzung einfacher ist als mit OpenGL. Als Problempunkt ist die Nicht-Unterstützung anderer Sprachen, wie Java-Script, zu erwähnen. Leider kann bei der Umsetzung die so gepriesenen Optimierungen von Java3D in der Regel nicht genutzt werden. So ist die Nutzung der Schedulerregionen nur eingeschränkt möglich, da die Bounding-Boxes per default als unendlich groß definiert sind. Außerdem kann nicht der compiled-retained mode von Java3D genutzt werden.

## 6.8 Zusammenfassung

In diesem Kapitel wurde im wesentlichen ein Vergleich der Animationsmodelle von VRML und Java3D vorgenommen. Da OpenGL kein Animationsmodell besitzt, wurde meist nur schemenhaft eine Umsetzung der Animationsmodellkomponenten in C++ und OpenGL gezeigt.

Bei den einzelnen Vergleichen hat sich gezeigt, daß bei Java3D durch die direkte Einbettung in Java, das Konzept der Verhalten-Klasse effizienter ist. Am wichtigsten ist aber der Vorteil von Java3D in diesem Bereich, der durch die Möglichkeit entsteht, die komplette Java-Umgebung nutzen zu können. Bei VRML kann nur ein kleiner Teilbereich der Klassen von Java genutzt werden, oder es muß mit Tricks gearbeitet werden.

Bei dem Vergleich der Interpolator-Realisierung hat sich gezeigt, daß sich die beiden Schnittstellen in diesem Bereich nur unerheblich unterscheiden. Der wichtigste Unterschied bildet das Vorhandensein der Aktivierungsfunktion bei Java3D, da diese nicht nur lineare Interpolation zuläßt.

Die Betrachtung der verschiedenen Ereignisse bei VRML und Java3D hat zu dem Resultat geführt, daß VRML umfassendere Werkzeuge bereitstellt, die aber auch schwieriger den eigenen Bedürfnissen angepaßt werden können. Java3D hingegen bietet die Möglichkeit, eine Reihe von Ereignissen abfragen zu können. Diese Ereignisse können beliebig kombiniert und an beliebige Aktionen gekoppelt werden.

Es wurde auch überprüft, ob die Auswahl-Klasse nützlich ist, um vergleichbare Drag-Sensoren zu denen in VRML zu erstellen. Die Überprüfung hat ergeben, daß Drag-Sensoren in Java3D mit geringem Aufwand erstellt werden können.

Einer der Abschnitte hat sich mit dem Optimierungswerkzeug 'Animation-Culling' beschäftigt. Es wurde also das Konzept der Schedulerregion in Java3D und der Bounding-Box in VRML unternommen, um zu sehen, wie effektiv das culling von nicht sichtbarer Animation ist. Die Schedulerregion bietet den Vorteil, daß sie aufgezwungen wird, hingegen die Bounding-Box den meisten VRML-Welt-Erstellern unbekannt ist. Der größte Vorteil der Schedulerregion ist, daß sie Bestandteil des zentralen Scheduler-Konzeptes ist. Die Bounding-Boxes sind hingegen dezentral verwirklicht und existieren nicht für alle Sensoren.

Bei dem Vergleich der zentralen Mechanismen der Animationsmodelle, den Scheduler von Java3D und dem Route-Mechanismus von VRML, hat sich gezeigt, daß dieser Mechanismen nicht vergleichbar sind. Das liegt daran, daß der Route-Mechanismus nur Variablenänderungen weitergibt, es handelt sich dabei nicht zwingend um Ereignisse. Der Scheduler hingegen ruft bei Ereignissen Verhalten-Objekte oder Interpolatoren auf.



Den Abschluß bildet die Betrachtung, ob ein VRML-Browser in Java3D geschrieben werden kann. Diese Betrachtung hat ergeben, daß der Implementierung eines VRML-Browsers in Java3D nichts im Wege steht und daß sie einfacher ist als in C++ und OpenGL.

## 7. Die abschließende Bewertung der 3D-Graphik-Schnittstellen

Im vorhergehenden Kapitel wurden die einzelnen Komponenten der Animationsmodelle von den drei 3D-Graphik-Schnittstellen verglichen. Dieses Kapitel soll ein Ausblick auf weitere Entwicklungen in dem 3D-Graphik-Schnittstellen-Bereich geben. Außerdem sollen zu den einzelnen 3D-Graphik-Schnittstellen noch zusätzliche Informationen gegeben werden, die zur abschließenden Bewertung noch notwendig sind.

Wie schon gezeigt wurde, ist die Hardwareabstraktionshierarchie der drei Schnittstellen wie folgt, als erstes die am weitesten von der Hardware entfernte Schnittstelle:

VRML → Java3D → OpenGL

Das bedeutet, daß OpenGL von Java3D- und VRML-Implementierungen und Java3D von VRML-Implementierungen genutzt werden kann. Es wird bei der folgenden Betrachtung mit der abstraktesten Schnittstelle begonnen.

### 7.1 VRML

VRML unterscheidet sich von den beiden anderen Schnittstellen dadurch, daß sie zu keiner Programmiersprache gehört. Das heißt, programmiersprachentypische Mechanismen wie Schleifen oder Bedingungen existieren im VRML-Standard nicht. Daraus ergibt sich natürlich ein starres System, was sehr gut an der VRML Version 1.0 zu sehen ist. Die Version 1.0 von VRML stellt keine Möglichkeiten bereit, Animationen zu erstellen. Sie beschreibt nur 3D-Objekte. Als Ausnahme ist die Möglichkeit der Navigation durch die beschriebene und angezeigte 3D-Welt zu nennen. Dieses starre System wurde mit der Version 2.0 von VRML ein wenig aufgeweicht, indem Sensoren, Interpolatoren und ein Nachrichtensystem eingeführt worden sind. Ganz außen vorgelassen ist dabei der Punkt des Script-Knotens oder des EAI's. Diese beiden Mechanismen bieten die Möglichkeit eine Programmiersprache mit VRML zu verbinden. Es ist also eine gleichgestellte Beziehung zwischen Programmiersprache und 3D-Graphik-Schnittstelle gegeben und nicht eine Ist-Teil-Beziehung von 3D-Graphik-Schnittstelle und Programmiersprache. Diese Art von Beziehung, die VRML zu Programmiersprachen hat, verleiht ihr zwar Vorteile, aber auch Nachteile. Zu den Vorteilen ist die Unabhängigkeit von der Programmiersprache zu zählen. Der Vorteil liegt darin, daß der 3D-Weltersteller seine bevorzugte Programmiersprache benutzen kann, oder es kann die für das gestellte Problem am besten geeignete Programmiersprache benutzt werden. Ein weiterer Vorteil ist, daß VRML durch sein hohes Abstraktionsniveau relativ einfach gehalten ist. Zu den Nachteilen zählen die schlechtere Performance und die schwache Dynamik. Das heißt, die 3D-Welten werden, bei gleicher Hardware- und Software-Umgebung, langsamer dargestellt als vergleichbare 3D-Welten, die in anderen Sprachen geschrieben sind, und es sind viele Möglichkeiten verschlossen für eigene Ansätze. Das äußert sich darin, daß es der Programmiersprache in Interaktion mit VRML nur erlaubt ist, Variablen zu verändern. Wenn aber komplette Systeme wie ein Drag-Sensor und ein Sichtbarkeit-Sensor miteinander verbunden werden, dann wird das Problem nicht nur mit zu umfassenden Werkzeugen 'erschlagen', sondern es gibt auch oft nicht genau die Verbindungsstelle, die gerade benötigt wird. Denn es gibt nicht viele Verbindungsstellen (=Variablen) bei den einzelnen Komponenten. Gerade dieses Vorhandensein von nur wenigen Verbindungsstellen verursacht auch ein anderes Problem. Dieses Problem ist, daß es einige Sonderfälle in VRML gibt.

Unter die Sonderfälle sind zu zählen:

- Der Route-Mechanismus kann mittels Def/Use-Befehlen umgangen werden.
- Es existieren nur dezentral geregelte Bounding-Boxes.
- Das EAI umgeht den Script-Knoten-Mechanismus.

- Die Möglichkeit des asynchronen Verhalten von Java-Objekten zum VRML-Browser.

Diese Liste ist nicht vollständig, sie zeigt aber gut, daß VRML kein einheitliches Konzept bei seinem Animationsmodell hat. Im Gegensatz zu Java3D, das ein Animationsmodell hat, welches zentrale Konzepte besitzt, die nicht umgangen werden können und müssen (siehe Kapitel 7.2).

## 7.2 Java3D

Java3D hat einen Vorteil zu VRML, es gibt eine Schnittstelle zu einer Programmiersprache, die sich im Internet-Bereich langsam durchsetzt (siehe Anhang E). Java ist durch seine Plattformunabhängigkeit geradezu prädestiniert, im Internet-Bereich eingesetzt zu werden. Obwohl es nicht immer so aussah, daß sich Java durchsetzen würde, hat es doch an Beliebtheit gewonnen. Allerdings wird Java in den meisten Fällen noch nicht professionell eingesetzt, das heißt, es wird meist mit Java nur experimentiert. Was wohl daran liegen könnte, daß die Unternehmen Angst haben, Geld in die Entwicklung von Programmen zu investieren, die evtl. bald nicht mehr genutzt werden können, weil Java nicht mehr existiert. Diese Skepsis gegenüber neuen Technologien im Software-Bereich ist immer wieder zu beobachten. Leider wurde die Frage nach der Skepsis gegenüber neuen Technologien in der oben angesprochenen Statistik nicht berücksichtigt, so daß über diesen Bereich keine Werte vorliegen. Ein anderer Grund für die noch nicht so starke professionelle Nutzung von Java liegt sicher auch in der Tatsache, daß Java noch relativ jung ist und erst einmal in der Praxis gezeigt werden muß, wo die Stärken und Schwächen liegen und evtl. Fehler entfernt werden müssen, soweit das möglich ist. Aber bisher spricht nichts dagegen Java zu nutzen, auch wenn Microsoft diese Entwicklung nicht so gerne sieht. Denn Microsoft baut momentan auf die Inkompatibilität von Windows zu anderen Betriebssystemen, um ihre Monopolstellung im Betriebssystem-Bereich zu halten. Diese Inkompatibilität würde durch Java aufgehoben werden und jeder könnte wieder das Betriebssystem nutzen, was er will. So hat Microsoft versucht diese Entwicklung, durch eine nicht korrekte Implementierung der Java-Virtuelle-Maschine in ihren Internet-Explorer 4.0 [Java-Law] aufzuhalten. Java hat aber auch einen großen Nachteil, es läuft in der Regel langsamer als andere Programme, die dasselbe machen. Dies liegt daran, daß Java interpretiert wird und nicht in Maschinencode vorliegt. Dieses Manko soll zwar mit dem Just-in-Time-Compiler (JIT) behoben werden, aber dieser muß natürlich vor der Ausführung des Programmes den Bytecode erst übersetzen. Auch ist das Starten des Java-Interpreters in der Regel mit einigem Zeitaufwand verbunden. Zwar fällt das Starten des Java-Interpreters weg, wenn ein zweites Java-Applet gestartet wird, doch die Geschwindigkeit, die bei dem Ausführen eines Java-Programmes zu beobachten ist, liegt unterhalb vergleichbarer Programme. Gerade bei dem Rendern von 3D-Szenen ist es wichtig, die Hardware so gut wie möglich auszunutzen. Da Java3D mindestens das Java-Runtime-Environment (JRE) Version 1.2 benötigt, um zu laufen, ist zu hoffen, daß im Zuge der Etablierung von Java 1.2 auch bessere Hardwareausnutzung stattfindet. Das heißt, daß die Interpreter grundsätzlich durch JIT-Compiler ersetzt werden, die schnell übersetzen und hardwarenähere Programme erzeugen als die bisherigen JIT-Compiler. IBM ist in diesem Bereich schon bei der Entwicklung eines Programmes, das Java-Bytecode in ausführbare Programme übersetzt. Leider ist dieses Programm nur auf AIX-Maschinen lauffähig, kann aber Programme für die Plattformen AIX, Windows NT und OS/2 übersetzen. Das Programm kann im Moment Java 1.0 und 1.1 Programme übersetzen und wird von IBM unter dem Namen HPC (High Performance Compiler) vertrieben [HPC]. Dieses Programm ist logischerweise nur für Java-Applikationen nutzbar, denn ausführbare Dateien können nicht in einem Internet-Browser ausgeführt werden. Auch das Visual-Café 2.5 von Symantec bietet die Möglichkeit Ausführbare-Programme für Win32 zu erstellen. Zum Ausführen der Java-Pro-

gramme ist allerdings immer noch eine spezielle Runtime-Umgebung notwendig, die sich aus mehreren DLL-Dateien zusammensetzt. Gerade das Java3D und Java 1.2 noch in der Entwicklung stecken, macht es 3D-Welt-Erstellern schwer, ihre Welten im Internet zu präsentieren. Es ist nur mit einigem Zusatzaufwand für den Internet-Surfer möglich, Java3D-Applets zu betrachten. Es muß zwar nur einmal Java3D installiert werden, aber dieser Aufwand ist nicht unerheblich. So brauchen Java3D-Applets mindestens das JRE 1.2 (9 MB) und natürlich die Java3D-Library (1.5 MB). Die Final-Release-Versionen vom JDK 1.2 und Java3D sind erst Anfang Dezember 1998 erschienen. Die Vorteile die Java3D dadurch, daß es eine Schnittstelle zu einer Programmiersprache hat, sind genau die Nachteile die VRML hat, daß es keine Schnittstelle zu einer Programmiersprache hat. Genauso sind in dem Bereich Programmiersprachenanbindung, die Nachteile von Java3D die Vorteile von VRML. Ein weiterer Vorteil von Java3D ist der compiled-retained mode. Durch ihn ist eine Möglichkeit geschaffen, animierte 3D-Welten in der internen Darstellung zu optimieren und dadurch die Darstellung der 3D-Welt zu beschleunigen. Diese Beschleunigung ist dabei relativ zu den anderen Modi von Java3D zu sehen. Wie oben schon angesprochen ist Java momentan immer langsamer als vergleichbar ausführbare Programme. Da aber die Konzepte, die in den beiden anderen Modi von Java3D mit denen von OpenGL (immediate mode) und VRML (retained mode) vergleichbar sind, ist eine relative Optimierung geschaffen worden. Es bleibt zu hoffen, daß bald aus der relativen Optimierung ein absoluter Geschwindigkeitsvorteil wird. Der angenehmste Vorteil für den 3D-Welt-Ersteller ist, daß Java3D zentrale Konzepte hat und keine Sonderfälle existieren.

Zu den zentralen Konzepten gehören:

- Scheduler (-regionen)
- Verhalten-Klasse (ein Interpolator ist eine vordefinierte Verhalten-Klasse)
- Objektorientierte Programmierung

### 7.3 OpenGL

Zu OpenGL gibt es nicht viel zu sagen, da es kein Animationsmodell hat. OpenGL ist nur zum Erstellen von einzelnen Bildern geschaffen worden und bildet beim Rendern ein abgeschlossenes System, so daß nur sehr eingeschränkt von außen, mittels der Programmiersprache, in das Rendern eingegriffen werden kann. Es kann aber ein Animationsmodell durch die Programmiersprache geschaffen werden, da sich OpenGL sehr gut einbinden läßt. So ist zum Beispiel Java3D in der Regel in OpenGL geschrieben. OpenGL ist der älteste Standard von den dreien und ist auch der zur Zeit am weitesten verbreitete Standard im Bereich des 3D Rendern.

### 7.4 Spiele und 3D-Grafik-Schnittstellen

Gerade im Bereich Spiele auf dem Computer kommt es auf Animationsgeschwindigkeiten an. Allgemein kann zu den drei Schnittstellen bemerkt werden, daß alle erst mit einer 3D-Graphikkarte gut laufen. Außerdem unterstützt nur OpenGL die Möglichkeit eines Vollbildschirmes, der für die meisten Computerspiele notwendig ist, das heißt VRML- und Java3D-Welten können nur im Fenster-Modus benutzt werden. Diese Einschränkung macht es undenkbar, daß VRML oder Java3D als 3D-Graphik-Schnittstelle für Spiele genutzt wird. Ganz davon abgesehen, daß 3D-Add-On-Graphikkarten, die nur im Vollbildmodus ihre 3D-Funktionen bereitstellen, nicht von Java3D-Applikationen oder VRML-Welten genutzt werden. Gerade um dieses Thema ist einige Diskussion in der Java3D-Mailing-Liste entstanden. Diese Diskussion hat *Sun* dazu verleitet, ein Statement dazu abzugeben. In diesem Statement sagen sie, daß eine Vollbildunterstützung frühestens ab der Java Version 1.3 möglich sein wird (siehe Anhang D).

Davon abgesehen, daß gerade erst Java 1.2 herausgekommen ist, es also noch längere Zeit dauern wird, bis Version 1.3 auf den Markt kommt, gibt es wenigstens die Hoffnung, daß Java3D irgendwann auch für Spiele geeignet sein wird.

### **7.5 Schlußbemerkung**

Wie oben erwähnt, ist Java noch dabei, sich im WWW-Bereich durchzusetzen. Es wird sich somit Java3D nur dann im Internet-Bereich durchsetzen, wenn auch Java sich durchgesetzt hat. Wichtig für die Durchsetzung von Java3D ist auch, daß das VRML-Consortium sich nicht gegen Java3D ausspricht, sondern sogar eine eigene WorkingGroup zur Kooperation geschaffen hat. Diese WorkingGroup, die VRML-Java3D WG, hat zur Aufklärung der Verhältnisse von VRML und Java3D beigetragen und somit den Konkurrenzgedanken entschärft. Denn VRML ist ein Internet-Standard, der auf einer abstrakteren Ebene benutzt wird und jeder 3D-Welt-Ersteller kann sich überlegen, welche Ebene seinem Problem am hilfreichsten ist. Der Entwickler hat nicht die Wahl zwischen VRML und Java3D zu treffen, sondern zwischen der Abstraktionsebene. Diese Wahl fällt den meisten Entwicklern in der Regel leichter, da sie nicht durch ein Konkurrenzdenken erschwert wird. Ob sich aber Java und somit Java3D im Applikations-Bereich durchsetzen wird, bleibt abzuwarten. VRML hat sich als ein Internet-Standard durchgesetzt, denn die Konkurrenz, Active 3D von Microsoft, ist nicht auf allen Plattformen vorhanden. Daß ein Bedarf an 3D-Graphik im WWW besteht zeigt die Statistik, die im Anhang E zu sehen ist.

Java3D hat also die Brücke zwischen der Rendermaschine OpenGL, die sehr aufwendig zu programmieren ist, und der Beschreibungssprache VRML, die nicht alle Möglichkeiten bietet, geschlagen.

## **Anhang A: Known Issues and Bugs in Java3D 1.1 Beta 2 Release**

These are the known bugs for the Java3D™ 1.1 Beta 2 early access release.

### **Core Bugs (j3d and vecmath)**

- Java 3D does not currently run in browsers. Only applications and appletviewer are currently supported. Java 3D applets may run in a browser using Java Plugin, but that is not yet supported.
- Setting a Shape3D node's geometry to null generates a NullPointerException
- Only the first Link node that points to a SharedGroup gets rendered
- Problem with multiple views: viewplatform only refers to last view attached
- Both old and new child of switch node sometimes rendered when whichChild changes
- Transparency fails to render correctly in Mixed Mode
- After switch from Mixed mode to normal mode, IllegalSharingException thrown
- Appletviewer's clone operation sometimes fails with Java 3D.
- Java 3D has some interoperability problems with Swing components.
- Billboard node not synchronized with view platform changes
- Billboard Node affected by other TransformGroups
- NullPointerException if Canvas repaint occurs before audio device initialized
- Spatialized sounds don't track view platform changes
- Both old and new child of switch node sometimes rendered when whichChild changes  
SingularMatrixException
- Threads can leave behind null pointer exception at quit time
- Java 3D classes and threads are not cleaned up and cannot be unloaded
- Sound examples do not run--or are pathologically slow--in appletviewer
- View stopBehaviorScheduler hangs the caller thread when no behavior defined
- Non-Congruent matrix above view platform will kill the traverser thread
- Texture2D.setImage() can cause a NullPointerException
- wakeupOn fails to register a condition when called with the same reference twice
- Soundscape.getAuralAttributes() throws a NullPointerException when attributes are null when called
- GraphicsContext3D getSound(int index) throws ClassCastException
- Alpha.set\*() operators don't recompute values when running
- value(atTime) method of Alpha Class returns wrong value
- Alpha.value() hangs if start time is not set, or is set too far in the past
- AlphaRampDuration does not always work properly
- Alpha loops too many times

### **Utility Bugs**

- IllegalArgumentException when using Triangulator with indexed colors
- VRML97 Simple and Viewer demos require path to be fully specified on command line on win32
- ObjectFile loader only reads coordinate data - other data ignored
- FourByFour generates SecurityException when trying to write out the high score file
- NormalGenerator producing strange creases

### **Solaris-specific bugs**

- Programs using LineStripArray may crash on Creator-3D (will be fixed in OpenGL 1.1.1)

patch 106022-04)

- The 8-bit TrueColor colormap may not be loaded on an 8-bit frame buffer running CDE
- This isn't Java3D-specific, but several font warnings of the following form occur on some Solaris 2.5.1 systems when running any Java program with JDK1.2beta4. The warnings seem to be harmless.

```
Font specified in font.properties not found
[-b&h-lucida sans-medium-r-normal-sans-*-%d-*-*p-*-iso8859-1]
Font specified in font.properties not found
[-b&h-lucida sans-medium-i-normal-sans-*-%d-*-*p-*-iso8859-1]
...
```

### Windows-specific bugs

- Bug in Symantec JIT causes the GearBox example to render incorrectly.
- Symantec JIT issues warning when running FourByFour example
- Java 3D crashes NT when ObjLoad is closed on Accell Graphics Card
- VRML97 examples with sound don't work
- VRML97 viewer gets exception on crazy\_cube.wrl
- Can get  

```
java.lang.RuntimeException: (description unavailable)
in
javax.media.j3d.Canvas3D.createContext (Native Method)
if not enough framebuffer resources. Reduce the framebuffer depth to 16-bits and
decrease the display resolution as a work around.
```
- Text2D utility's text is partially clipped on NT

### Direct3D-specific bugs

Please Note: The Java 3D version for Direct3D is still in its initial stages of development. You may encounter several errors besides the ones listed here.

- The Morph Node does not work
- Textured polygons come out modulated by base color when the mode is REPLACE
- Transparency value from RGBA textures is not blended
- Backfacing, textured polygons don't show up when culling is turned off
- Separate strips of LineStripArray are connected
- PureImmediate mode does not work after paint() event on Win95/98 with OpenGL
- Shaded objects with a unique color per-vertex aren't rendered correctly
- Background image is rendered incorrectly
- PointSize is ignored (1 is always used)
- Textures appear washed out in PickWorld
- Text2D example: text only seen 1/2 of the time
- Fog doesn't work
- Lighting position is sometimes incorrect (shows up in many examples)
- Light scoping doesn't work
- Light attenuation doesn't work
- Texture filter and mode attributes have no effect

## Anhang B: Known Bugs (VRML mit Java3D)

### AudioClip

- pitch is automatically updated but PointSound, though currently no eventOut is generated.

### Browser

- replace world broken, but load vrml from string and load by wrl ok.

### Background

- no ground or geometry or gradient, just solid color

### Billboard, BillboardAxis

- TBD: complete the eventOut generation. (eventOut nyi) even though the behavior has updated the transform the eventOut may still go somewhere

### ContentNegotiator

- Sound loading is left to the j3d sound, but is massaged by the browser who gets the audioDevice to prepareSound on them; this seems to help getting the duration of the clip in time, but is not advised that we use that interface.
- there may be an api to MediaContainer in 1.2 for setting via byte array, which we'll probably want to use.

### CoordinateInterpolator

- TBD

### Extrusion

- TBD

### ElevationGrid

- TBD

### Group

- some capabilities have been commented out, which are useful during debugging. This should be commented as such.

### ImageTexture

- the scale is preset to 512x512, this should be an option "scale to closest image size", "scale to faster (256^2)", "scale to nicer (512^2)"

### IndexedFaceSet

- currently defining the behavior for updates to coordinate.point normal.vector texcoord.point if they have been route\_ prefixed.

### Loader

- header check code broken

### MFNode

- No method: delete(int i)



#### MFRotation

- Warning: bogus rotations unchecked, unless they go through SFRotation. eg set1Value(..) did not check.

#### MovieTexture

- tbd, awaiting better integration with JMF

#### NavigationInfo

- speed should really tweak the evagation (TBD), as should avatarSize

#### PixelTexture

- no parser

#### PlaneSensor

- tbd

#### Script

- needs vrml.\*, vrml.node.\*, vrml.field.\* wrappers to be iso compliant, however, scripts may import directly from the public com.sun.j3d.loaders.vrml97.\*, com.sun.j3d.loaders.vrml97.node.\* and com.sun.j3d.loaders.vrml97.field.\* classes.

#### SphereSensor

- bug: only works correctly if in same coordinate system as viewer, ie, its z points towards the viewer.
- nyi: offset events,

#### SpotLight

- attenuation tbd

#### TouchSensor

- isOver needs to get delivered from Evagation as well. This should be doc'd in Evagation, but is fairly simple loop.

#### VisibilitySensor

- tbd

#### EAI

Entire External Authoring Interface, tbd

TBD: to be decided

NYI: not yet implemented

## Anhang C: Email von Henry Sowizral (Java3D)

From Henry.Sowizral@Eng.Sun.COM Tue Oct 13 10:24:01 1998  
Date: Tue, 6 Oct 1998 10:32:31 -0700 (PDT)  
From: Henry Sowizral <Henry.Sowizral@Eng.Sun.COM>  
To: stein\_c@mathematik.uni-marburg.de  
Subject: Re: J3D Issue: Informations about animation engineering

Thank you very much for your interest in Java 3D and its internals.  
Sorry for the delay in responding but I was away on a business trip in Europe (including a part of Germany).

Java 3D itself has no explicit concept of an event. It has a well specified and constrained set of WakeupConditions that it knows how to track and when appropriate cause the BehaviorScheduler to schedule a Java 3D behavior.

Java 3D's current implementation uses a number of cooperating threads that together perform all the tasks needed to generate sound, render objects on the screen, cause behaviors to occur, check for collisions, etc.

The AND/OR tree can be implemented in a variety of ways. In general, whenever a behavior registers a WakeupCondition, Java 3D constructs a new branch in the overall tree. As various items that Java 3D tracks effect that tree the associated item's condition gets set accordingly. This information bubbles up the tree to the highest level possible. If that level corresponds to a completely satisfied WakeupCondition, then the Behavior scheduler at its next opportunity runs the associated Java 3D behavior.

Unfortunately, it would be quite lengthy to give you all the details of how our implementation works, but this general explanation should give you a good set of hints....

--- Henry

Henry A. Sowizral  
Sun Microsystems, MS UMPK27-101      office: 650/786-6579  
901 San Antonio Road                      fax: 650/786-7359  
Palo Alto, CA 94303-4900

## **Anhang D: Email von Kevin Rushforth (Java3D)**

From Kevin.Rushforth@Eng.Sun.COM Mon Nov 16 14:33:13 1998  
Date: Wed, 11 Nov 1998 21:02:49 -0800  
From: Kevin Rushforth <Kevin.Rushforth@Eng.Sun.COM>  
To: java3d-interest@sun.com  
Subject: Full Screen mode

Java does not currently have the capability to manage a full screen "window". The Java 3D team is working with the JDK team to get this feature added to the next (1.3) release of Java. In the mean time, we are looking into ways to provide this capability through other means for specialized applications.

--

Kevin Rushforth  
Java 3D Team  
Sun Microsystems

kcr@eng.sun.com

### Anhang E: Statistikwerte einer Umfrage zum Internet (Teilbereich Java)

Die Statistik Werte von der 'Graphic, Visualization & Usability Center's (GVU) 8th WWW User Survey'-Untersuchung wurden hier in eine Tabellenform gebracht, die nur die für diese Arbeit interessanten Daten enthält. Bei weiterem Interesse ist die Homepage der Untersuchungskommission [WWW-Statistics] aufzusuchen. Die Untersuchung hat Ende 1997 stattgefunden und es haben sich ca. 10.000 Internet-Nutzer an ihr beteiligt. Es gibt einen Teilbereich, der Fragen nur für Web-Autoren beinhaltet. Die erste Tabelle bezieht sich auf alle Internet-Nutzer. Die zweite Tabelle bezieht sich nur auf die Antworten von Web-Autoren, das heißt die Prozentangaben beziehen sich nur auf die Gesamtheit der Web-Autoren und nicht auf die Gesamtheit aller Internet-Nutzer.

|                                | Genutzte Internet Technologien | Als notwendig angesehene Internet Technologie |
|--------------------------------|--------------------------------|---|
| 3D (VRML, Active3D und andere) | 18,6 %                         | 4,5 %   |
| Java (/Java Script*)           | 68,5 %                         | 21,6 %  |

Tabelle 5: Vergleich von VRML und Java bezüglich Nutzung und Notwendigkeit

\* Java Script wurde nur bei 'Genutzte Internet Technologien' mit gefragt

|              | Benutzung von fertigen Java Applets | Selber schon in Java programmiert | Ist ein Java-Einsatz geplant |
|--------------|-------------------------------------|-----------------------------------|------------------------------|
| Ja           | 41 %                                | 34 %                              | 55 %                         |
| Nein         | 55 %                                | 66 %                              | 25 %                         |
| Nicht sicher | 4 %                                 |                                   | 20 %                         |

Tabelle 6: Vergleich von den Nutzungsarten von Java

# Abbildungsverzeichnis

|               |  |    |
|---------------|--|----|
| Abbildung 1:  | Hierarchie der 3D-Graphik-Schnittstellen.....                    | 3  |
| Abbildung 2:  | Die Baumstruktur des (compiled-) retained mode in Java3D .....   | 6  |
| Abbildung 3:  | VRML-Interfaces-Übersicht.....                                   | 12 |
| Abbildung 4:  | Die OpenGL-Pipeline .....  | 14 |
| Abbildung 5:  | Das Animationsmodell von VRML.....                               | 17 |
| Abbildung 6:  | Die VRML-Umgebung.....   | 18 |
| Abbildung 7:  | Der Route-Mechanismus als Kernel dargestellt .....               | 20 |
| Abbildung 8:  | Ein Route-Graph-Beispiel .....                                   | 20 |
| Abbildung 9:  | Das Animationsmodell von Java3D .....                            | 28 |
| Abbildung 10: | Beispiele von Aktivierungsfunktionsteigungen.....                | 32 |
| Abbildung 11: | Asynchrones Verhalten von Java-Objekten zum VRML-Browser.....    | 37 |
| Abbildung 12: | Auffinden einer nicht mehr stattfindenden Kollision in VRML..... | 45 |
| Abbildung 13: | Ein einfacher Sichtbarkeit-Sensor-Algorithmus.....               | 46 |

# Tabellenverzeichnis

|  |    |
|--|----|
| Tabelle 1: 3D-Graphik-Schnittstellen und Programmiersprachen .....               | 11 |
| Tabelle 2: Methoden der Browser-Klasse von Javascript .....                      | 24 |
| Tabelle 3: Java3D- und VRML-Ereignisse .....                                     | 43 |
| Tabelle 4: VRML-Browser mit Java3D oder OpenGL .....                             | 53 |
| Tabelle 5: Vergleich von VRML und Java bezüglich Nutzung und Notwendigkeit ..... | 67 |
| Tabelle 6: Vergleich von den Nutzungsarten von Java .....                        | 67 |

# Programmbeispielverzeichnis

|   |    |
|---|----|
| Programmbeispiel 1: Der Rotations-Interpolator in Java3D.....                 | 8  |
| Programmbeispiel 2: VRML und VRML-Script .....                                | 13 |
| Programmbeispiel 3: OpenGL und C.....   | 15 |
| Programmbeispiel 4: Die Verhalten-Klasse von Java3D.....                      | 16 |
| Programmbeispiel 5: Sensor und Eingabegerät mit Java3D .....                  | 29 |
| Programmbeispiel 6: Asynchrones Verhalten von Java-Objekt und VRML-Welt.....  | 38 |
| Programmbeispiel 7: OpenGL und Tastaturabfrage mit MFC.....                   | 40 |
| Programmbeispiel 8: Der Normalen-Interpolator von VRML.....                   | 41 |
| Programmbeispiel 9: Nichtlineare Aktivierungsfunktion .....                   | 42 |
| Programmbeispiel 10: Pendelbewegung mit VRML .....                            | 42 |
| Programmbeispiel 11: VRML und Kollisionen .....                               | 45 |
| Programmbeispiel 12: Farben-Interpolator auf Abruf (Bereichsaktivierung)..... | 50 |

# Literatur

## Ames

Ames, Andrea L., David R. Nadeau, John L. Moreland  
VRML 2.0 Sourcebook (2. Auflage)  
Wiley, 1997  
ISBN 0-471-16507-7

## Angel

Angel, Edward  
Interactive Computer Graphics  
Addison Wesley, 1997  
ISBN 0-201-85571-2

## Foley

Foley, James D., Andries van Dam, Steven K. Feiner, John F. Hughes  
Computer Graphics - Principles and Practice (2. Auflage)  
Addison Wesley, 1990  
ISBN 0-201-12110-7

## Hase

Hase, Hans-Lothar  
Dynamische virtuelle Welten mit VRML 2.0  
Verlag für digitale Technologie, 1997  
ISBN 3-920993-63-2

## Kloss

Kloss, Jörg, Robert Rockwell, Kornél Szabó, Martin Duchrow  
VRML 97  
Addison-Wesley, 1997  
ISBN 3-8273-1187-X

## Lea

Lea, Rodger, Kouichi Matsuda, Ken Miyashita  
Java for 3D and VRML worlds  
Indianapolis (Ind. New Riders) , 1996  
ISBN 1-56205-689-1

## OpenGL 97-1

OpenGL Reference Manual (Version 1.1), Second Edition  
Addison Wesley, 1997  
ISBN 0-201-46140-4

## OpenGL 98-1

The OpenGL Graphics System: A Specification (Version 1.2)  
<ftp://sgigate.sgi.com/pub/opengl/doc/opengl1.2/opengl1.2.pdf>



#### Roehl

Roehl, Bernie, Justin Couch, Cindy Reed-Ballreich, Tim Rohaly, Geoff Brown  
Late Night VRML 2.0 with Java  
Macmillian Computer Publ. (ZD), 1997  
ISBN 1-56276-504-3

#### Sommer

Sommer, Manfred  
Unterlagen zur 3D-Graphik Programmierung Vorlesung im Wintersemester 1996/97  
Philipps Universität Marburg, Fachbereich Mathematik und Informatik

#### Sun 98-1

Sowizral, Henry, Kevin Rushforth, Michael Deering  
The Java3D API Spezifikation  
Addison-Wessley, 1997  
ISBN 0-201-32576-4  
<http://java.sun.com/products/java-media/3D/j3dguide.pdf>

#### Tannenbaum

Tannenbaum, Andrew S.  
Betriebssysteme - Entwurf und Realisierung, Teil 1: Lehrbuch  
Hanser Verlag 1990  
ISBN 3-446-15268-7

#### VRML 97-1

VRML97  
<http://www.vrml.org/Specifications/VRML97>

# URLs

## CGL

Computer Graphics Lab of the University of Waterloo:  
An (Unofficial) OpenGL Port for Java (1996)  
<ftp://cgl.uwaterloo.ca/pub/software/meta/OpenGL4java.html>

## HPC

High Performance Compiler for Java  
<http://www.alphaWorks.ibm.com/formula/hpc.html>

## Java-Law

Lawsuit Related Information  
<http://java.sun.com/lawsuit>

## Marrin

Marrin, Chris: Anatomy of a VRML Browser  
<http://www.marrin.com/vrml/Interface.html>

## ML-J3D

Java3D-interest mailing list  
<http://java.sun.com/products/java-media/mail-archive/3D>

## MS 98-1

Microsoft Windows 95 Graphics Architecture  
<http://www.microsoft.com/directx/pavilion/hardware/win95arch.htm>

## Nadeau

Nadeau, David R.  
Introduction to VRML 97 (and bits of Java3D)  
<http://www.sdsc.edu/~nadeau/Courses/Visualization97/vrml97.htm>

## SGI 97-1

SGI and Microsoft form strategic alliance to define the future of graphics  
[http://www.sgi.com/Headlines/1997/December/fahrenheit\\_release.html](http://www.sgi.com/Headlines/1997/December/fahrenheit_release.html)

## VRML 98-1

VRML-Consortium: Current Working Groups  
<http://www.vrml.org/WorkingGroups>

## WG-J3D

VRML-Java3D Working Group  
<http://www.vrml.org/WorkingGroups/vrml-java3d>

## WG-KB

Keyboard Input Working Group  
<http://www.vrml.org/WorkingGroups/kbinput/index.html>

WWW-Statistics

Graphic, Visualization, & Usability Center's (GVU) 8th WWW User Survey

[http://www.gvu.gatech.edu/user\\_surveys/survey-1997-10/](http://www.gvu.gatech.edu/user_surveys/survey-1997-10/)

## **Allgemeine URLs**

### **OpenGL**

OpenGL-Homepage

<http://www.opengl.org>

### **Java3D**

Java3D-Homepages

<http://www.sun.com/desktop/java3d>

<http://java.sun.com/products/java-media/3D>

The Java3D FAQ (von NCSA)

<http://tintoy.ncsa.uiuc.edu/~srp/java3d/faq.html>

Java3D-Repository

<http://java3d.sdsc.edu/>

### **VRML**

VRML-Homepage

<http://www.vrml.org/>

VRML-Repository

<http://www.sdsc.edu/vrml/>

VRML-Online-Book (Rikk Carey)

<http://www.best.com/~rikk/Book/book.shtml>

# Erklärung

Hiermit erkläre ich, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Marburg, den 18. Dezember 1998