Diplomarbeit

Ein Framework für die Erfassung
komponentenbasierter Modelle und deren
Präsentation mittels Java3D

Peter Blanchebarbe
Kaiserstr. 15A
66111 Saarbrücken
Matrikelnummer: 1118471

10. April 2001

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und nur unter Nutzung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Saarbrücken, den 10. April 2001

.....
Peter Blanchebarbe

Danksagung

Mein Dank gilt Herrn Dr. Stephan Diehl für die engagierte Betreuung der Diplomarbeit und für die tatkräftige Unterstützung bei der Anfertigung des Konferenzpapiers, das im Zuge dieser Diplomarbeit entstand.
Desweiteren bedanke ich mich bei Herrn Prof. Dr. Reinhard Wilhelm für die Möglichkeit, diese Diplomarbeit an dem Lehrstuhl für Programmiersprachen und Compilerbau anfertigen zu können.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Schaut man sich die Statistik an, so steigt die Akzeptanz und der Nutzen des Internet als Einkaufsmöglichkeit. Waren im Herbst 1996 nur ein Drittel der Internet-Nutzer der Ansicht, dass sich das WWW gut oder sehr gut zum Einkaufen eignet, waren es im Herbst 99 schon 87,3 %. Es gibt viele Prognosen über die Wachstumsraten im Bereich e-Commerce. Diese fallen oft sehr unterschiedlich aus. Sicher sind sich jedoch die Analysten und Meinungsforscher in dem Punkt, dass die Umsätze deutlich ansteigen werden. Die erfolgreichsten Firmen (wie Amazon und CDNow) bieten medienbasierte Produkte an, also Produkte kontextfreier Natur. So sind bei Büchern und CDs nicht das Aussehen, sondern lediglich der Inhalt interessant. Es gibt aber eine Vielzahl von Produkten, bei denen das Aussehen vorrangig oder zumindest wichtig ist, z.B. bei Möbeln, Autos oder im Bereich der Unterhaltungselektronik. Weiterhin werden viele Produkte per Internet vertrieben, die als Bausatz ausgeliefert werden, z.B.: Hardwarekomponenten für Computer. In einem solchen Fall wäre es günstig dem Kunden vor Augen führen zu können, wie dieses Produkt zusammengesetzt oder in ein bestehendes System integriert wird. Nur selten stellen Hersteller ihre Produkte in einem CAD Datenformat bereit, von 3D-Animationen ganz zu schweigen. Von daher wäre es wünschenswert, z.B. für Online-Shops, Produkte auf möglichst einfache Art und Weise zu erfassen und zu visualisieren. Aus diesem Grund wurde ein Programm namens GenAu-3D (Generisches Autorentool) entwickelt. Mit seiner Hilfe werden Produktkomponenten näherungsweise visualisiert. Diese Komponenten können dann in einer Animation zu einem Produkt zusammengesetzt werden.
Betrachtet man komponentenbasierte Modelle im allgemeinen, so finden sich auch außerhalb des Bereichs e-Commerce Anwendungsmöglichkeiten. So können einzelne Atome als Komponenten eines Moleküls betrachtet werden, die mittels ihrer Bindungskräfte verbunden sind.

1.2 Überblick

Komponenten spielen eine wichtige Rolle beim Aufbau von komplexen Modellen. Daraus ergeben sich folgende Fragen, die in der vorliegenden Arbeit beantwortet werden:

- **Wie können Komponenten modelliert werden ?**
  In Kapitel 4 wird das Objektmodell beschrieben, in dem die Daten für die Komponenten, Produkte etc. abgelegt werden. Es ist in XML beschrieben und wurde in einer DTD spezifiziert.

- **Wie können Komponenten erfasst und visualisiert werden ?**
  Hierzu wurde eine Benutzerschnittstelle namens GenAu-3D unter Java und Java3D entwickelt, mit deren Hilfe Komponenten erstellt und visualisiert werden können. In Kapitel 5 werden die Tools beschrieben, die Bestandteil von GenAu-3D sind:

  - Meta-Autorentool: Mit Hilfe dieses Tools werden domänenspezifische Autorentools erstellt. In dem Meta-Autorentool werden Objekte vordefiniert, die in dem domänenspezifischen Autorentool verwendet werden können. Ausserdem gibt es Strukturierungsmöglichkeiten, um einen schnellen Zugriff auf diese Objekte zu gewährleisten.
  - Autorentool: In einem domänenspezifischen Autorentool können Komponenten erstellt werden. Hierzu werden die vordefinierten Objekte, sowie Primitive bereitgestellt, die auf verschiedene Art und Weise verarbeitet werden können. In diesem Tool können auch die sogenannten Kontaktpunkte definiert werden. Mit Hilfe der Kontaktpunkte werden Verbindungen zwischen den Komponenten realisiert. Die von dem Meta-Autorentool vorgegebene Struktur hilft auch, die erstellten Komponenten zu verwalten.
  - Konfigurationstool: Mit Hilfe dieses Tools können Komponenten zu einem Produkt zusammengestellt werden.
  - Animationstool: In dem Animationstool wird Schritt für Schritt gezeigt wie ein Produkt aufgebaut ist.

- **Wie können Komponenten im Szenengraph repräsentiert werden ?**
  Der Szenengraph ist eine oft verwendete Struktur im Bereich der Visualisierung. Auch in Java3D wird diese Struktur verwendet, um 3D-Objekte grafisch darzustellen. In Kapitel 6 wird erläutert wie Komponenten im Szenengraph repräsentiert werden. Desweiteren werden mittels der Kontaktpunkte Verbindungen zwischen den Komponenten hergestellt, so dass real existierende Verbindungen im Szenengraph abgebildet werden. Dies ermöglicht eine intuitive Handhabung dieser Objekte und bildet eine gute Basis für Operationen auf komponentenbasierten Modellen.

Zunächst sollen jedoch die Technologien vorgestellt werden, die in dieser Arbeit verwandt wurden.

2 Technologien

In diesem Kapitel werden die Programmiersprache Java und das zur Visualisierung benötigte Java3D-API vorgestellt. Um die Transformationen in der virtuellen Welt zu verstehen, werden die theoretischen Grundlagen kurz erklärt. Weiterhin wird auf die Sprache XML eingegangen, mit deren Hilfe das Objektmodell definiert wurde.

2.1 Java

Die von Sun Microsystems entwickelte Programmiersprache Java ist seit 1995 frei verfügbar und dient zur Entwicklung plattformunabhängiger Programme. Hier kurz die wichtigsten Eigenschaften dieser Programmiersprache:

- **Portierbar**: Entwicklung architekturunabhängiger Programme.
- **Objektorientiert**: Klassen, die instanziiert werden können, liefern Objekte. Daten dieser Objekte werden mit Hilfe von Methoden verändert. Ausserdem bietet die Objektorientiertheit die Möglichkeit, Datenfelder und Methoden von Klassen an Subklassen zu vererben.
- **Interpretativ**: Der Java Compiler erzeugt Java-Bytecode. Dieser Bytecode kann auf verschiedenen Plattformen von der Java Virtual Machine ausgeführt werden. Um die Performance zu steigern, gibt es inzwischen auch Compiler, die den Java-Bytecode in den Maschinencode der zugrundeliegenden Plattform übersetzen. Es gibt zwei verschieden Arten von Compilern:
  - **Just in Time Compiler**: Dieser Compiler ermöglicht es den Java-Bytecode zur Laufzeit zu übersetzen. Vor allem Applets profitieren von diesem Compiler, da während der Ausführung von Applets Klassen nachgeladen werden können, die dann erst übersetzt werden.
  - **Optimierender Compiler**: Dieser Compiler kann Java-Quellcode oder Java-Bytecode in den Bytecode der zugrundeliegenden Plattform übersetzen. Dadurch, dass der komplette Sourcecode dem Compiler vorliegt, können hier auch mehr Optimierungen durchgeführt werden, als dies bei dem Just in Time Compiler der Fall ist. Dieser Compiler ist jedoch nur für Applikationen verwendbar.

Eine ausführliche Beschreibung dieser Programmiersprache ist unter [GM96] zu finden. Insbesondere bietet Java durch seine Portabilität die Möglichkeit die Programme als Applets über das Internet zu laden und auf dem heimischen Rechner auszuführen. Die Klassen, mit deren Hilfe Programme erstellt werden, liegen in Bibliotheken, den sogenannten *Application Programming Interfaces* (API) vor. Standardmässig liegt einer Java-Entwicklungsumgebung eine recht umfassende API bei, die auch das Erstellen von grafischen Benutzeroberflächen erlaubt. Diese Entwicklungsumgebung kann durch Hinzufügen von weiteren APIs beliebig erweitert werden.

2.2 Geometrische Transformationen

Da in den folgenden Kapiteln von Rotationen, Translationen und Skalierungen die Rede sein wird, soll hier beschrieben werden, wie diese Transformationen zu berechnen sind. Die Objekte und die Transformationen dieser Objekte werden in einem rechtshändigen Koordinatensystem beschrieben. Aus der Sicht des Betrachters gilt:

- x+: rechts
- y+: oben
- z+: auf den Betrachter zu

Zuerst müssen wir klären, wie ein Objekt dargestellt wird. Als Beispiel soll ein einfacher Quader dienen. Ein Objekt wird durch seine Definitionspunkte beschrieben. Anhand dieser Punkte werden die entsprechenden Verbindungslinien gezogen. Für einen Quader werden also acht Eckpunkte im Koordinatensystem definiert, die dann verbunden werden. Wird der Quader transformiert, so geschieht das anhand seiner Definitionspunkte. Sei $P' = (x', y', z')$ der Definitionspunkt, der durch eine Transformation aus $P = (x, y, z)$ hervorgeht. Dann werden die drei wichtigsten Transformationen auf folgende Art gebildet:

- **Translation** ($T(t_x, t_y, t_z)$): Der Punkt P wird um den Translationsvektor T verschoben ($P' = P + T$). Die Koordinaten des verschobenen Punktes sind dann:
  $x' = x + t_x, \quad y' = y + t_y, \quad z' = z + t_z$

- **Rotation** ($R_{x,y,z}(\theta)$) : Das Objekt wird im Ursprung um eine der Achsen (x,y,z) um den Winkel θ rotiert ($P' = P * R$). Dabei bedeutet ein positiver Winkel eine Rotation gegen den Uhrzeigersinn und ein negativer Winkel eine Rotation im Uhrzeigersinn. Ein Punkt wird dann je nach Achse folgendermaßen berechnet:
  $R_x(\theta)$: $x' = x, \quad y' = y * cos\theta - z * sin\theta, \quad z' = y * sin\theta + z * cos\theta$
  $R_y(\theta)$: $x' = x * cos\theta + z * sin\theta, \quad y' = y, \quad z' = -x * sin\theta + z * cos\theta$
  $R_z(\theta)$: $x' = x * cos\theta - y * sin\theta, \quad y' = x * sin\theta + y * cos\theta, \quad z' = z$

- **Skalierung** ($S(s_x, s_y, s_z)$): Jede Koordinate wird mit einem Skalierungsfaktor multipliziert. Dadurch erreicht man für das gesamte Objekt eine Vegrösserung bei $s > 1$ oder eine Verkleinerung bei $s < 1$ ($P' = P * S$):
  $x' = x * s_x, \quad y' = y * s_y, \quad z' = z * s_z$
  Besitzen alle Faktoren den gleichen Wert spricht man von einer uniformen Skalierung, und das Objekt wird proportionalitätserhaltend vergrössert oder verkleinert.

Transformationen sind *kongruent*, wenn die Winkel- und Längenverhältnisse der transformierten 3D-Objekte nicht verändert werden. Um Transformationen effizienter berechnen zu können, werden sie in Matrizenform gebracht. Ein Vorteil dabei ist, dass die Transformationen durch eine einzige Multiplikation berechnet werden können. Weiterhin können aufeinanderfolgende Transformationen eines Objektes in einer Matrix zusammengefasst werden. So kann z.B.

die Folge Translation, Rotation, Translation in einer Matrix ausgedrückt werden. Die Matrizendarstellung ist auch zu bevorzugen, da die heutige Hardware für solche Matrixoperationen optimiert ist. Zuerst müssen wir uns aber noch mit der Darstellung der Punkte beschäftigen. Der erste Schritt besteht darin, die Punkte in homogenen Koordinaten auszudrücken.Ein Punkt wird dann als Quadrupel $P = (x, y, z, W)$ mit $W \neq 0$ dargestellt. Zwei Punkte sind genau dann gleich, wenn der eine Punkt ein Vielfaches des anderen Punktes ist, z.B.: (2,3,4,2) und (4,6,8,4). Der Punkt wird homogenisiert, indem durch W mit $W = 1$ dividiert wird. Daraus folgt das Quadrupel $(x/W, y/W, z/W, 1)$ was nach obiger Definition den gleichen Punkt wie $(x, y, z, 1)$ darstellt. Mit Hilfe dieser Punkte lassen sich die Transformationen in Matrizenform darstellen.
Wie werden nun die einzelnen Transformationen in einer Matrix ausgedrückt? Für die Translation ergibt sich folgende Matrix:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Inverse Matrix erhält man durch die negativen Werte von t_x, t_y, t_z.
Für die Rotation gibt es gemäss der oben aufgeführten Erläuterung für jede Achse eine Matrix:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Betrachtet man die linke obere 3x3 Teilmatrix von R_x, R_y, R_z, so ist festzustellen, dass die Spalten- bzw. Zeilenvektoren paarweise senkrecht aufeinander stehende Einheitsvektoren sind. Es handelt sich hier also um drei speziell orthogonale Matrizen. Der Vorteil solcher Matrizen liegt darin, dass durch ihre Anwendung die Längen und Winkel des transformierten Objektes erhalten bleiben. Dies gilt auch für ein Folge von Rotationen. Die Rotationen können invertiert werden, indem man die negierten Winkel einsetzt. Die andere Möglichkeit ist es, die Indizes der Matrix zu vertauschen, d.h. man bildet die transponierte Matrix.

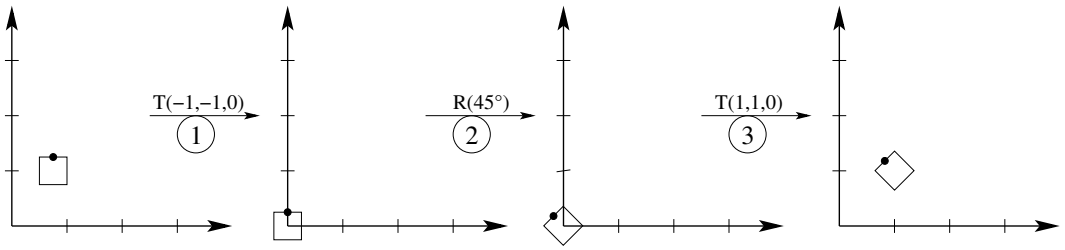

Abbildung 1: z-Rotation mit 45° um $P = (1, 1, 0, 1)$

Die Skalierungsmatrix sieht folgendermaßen aus:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die inverse Skalierungsmatrix wird gebildet indem s_x, s_y, s_z durch ihren reziproken Werte ersetzt werden.
Der Vorteil dieser Matrizen liegt darin, dass sie kombiniert werden können. Als Beispiel betrachten wir einen Quader, der an einem beliebigen Punkt P=(x,y,z,1) um den Winkel θ rotiert werden soll. Seine Position soll sich jedoch nicht verändern. Eine solche Rotation lässt sich durch drei Matrizen darstellen, die zu einer Matrix zusammenmultipliziert werden.
Folgende Schritte werden dazu für jeden Definitionspunkt ausgeführt.

- Verschieben: $T(-x, -y, -z)$.
- Rotieren: $R_z(\theta)$.
- Zurückverschieben: $T(x, y, z)$.

Ein Beispiel für diese Schrittfolge ist in Abbildung 1 zu sehen. Eine solche Rotation kann durch die Matrix M dargestellt werden, die folgendermaßen ermittelt wird:

$$M = T * R * T^{-1}$$

$$P' = M * P$$

Daraus ergibt sich:

$$M = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P' = \begin{bmatrix} cos\theta & -sin\theta & 0 & -x * cos\theta + y * sin\theta + x \\ sin\theta & cos\theta & 0 & -x * sin\theta - y * cos\theta + y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Bei den Matrizenmultiplikationen gilt die Assoziativität. Die Kommunikativiät gilt in der Regel nicht. Als Beispiel betrachten wir die Schritte ② und ③ in Abbildung 1. In der Abbildung ergibt sich für diese beiden Schritte folgende Matrizenmultiplikation: T(1,1,0)*R(45°)*(0,0,0,1). Das Ergebnis ist ein um 45° rotiertes Quadrat im Punkt (1,1,0,1). Werden diese Schritte vertauscht, so erhält man folgende Matrizenmultiplikation: R(45°)*T(1,1,0)*(0,0,0,1). In diesem Fall erhält man ein Quadrat, das leicht rotiert im Punkt P=(0,1,0,1) liegt. Ausführliche Informationen über Transformationen sind bei [Wat93] zu finden.

2.3 Das Java3D-API

Das Java3D-API besteht aus Java-Klassen, die ein Interface zu einer 3D-API wie *OpenGL*, *Direct3D* und *QuickDraw3D* herstellen. Auf der anderen Seite ermöglicht Java3D dem Programmierer auf komfortable Art, 3D-Objekte zu erzeugen. Diese Objekte werden in eine Baumstruktur eingefügt, den Szenengraphen. Der Szenengraph wiederum befindet sich in einem virtuellen Universum. Der Szenengraph bestimmt die Art und Weise, wie die in ihm enthaltenen Objekte gerendert werden. Unter Rendern versteht man die Darstellung des in einem Szenengraph enthaltenen Objekte auf dem Bildschirm. Das Koordinatensystem, in dem sich das Universum befindet hat folgende Eigenschaften:

- Rechtshändig.
- Winkel werden in Radians gemessen.
- Distanzen werden in Metern berechnet.

2.3.1 Der Szenengraph

Bei dem Szenengraph handelt es sich um einen gerichteten, azyklischen Graphen, der eine Baumstruktur bildet. Alle Knoten sind auf genau einem Pfad von der Wurzel aus zu erreichen. Neben den Knoten, die durch eine Elter-Kind-Relation verbunden sind (Kanten) gibt es noch die Knotenkomponenten, die von mehreren Knoten referenziert werden können und zusätzliche Daten enthalten. Der Szenengraph kann in zwei Teile aufgespalten werden:

- **Sicht-Ast**: Der Sicht-Ast ist der Teilgraph, der die Sicht des Benutzers auf eine Szene bestimmt. Hierbei wird die Position und Orientierung einer imaginären Person (Avatar) beschrieben, die auf einer Plattform im virtuellen Universum steht. Ein Objekt namens *View* beinhaltet alle Parameter, die die Sicht des Betrachters beeinflussen. So lässt sich z.B. die Grösse des Avatars verändern. Diese Eigenschaften haben Auswirkung auf das Rendern des Teilgraphen, der die grafischen Elemente beinhaltet. Weiterhin wird um den Avatar ein Region in Form einer Kugel definiert, die der Renderer benutzt um zu entscheiden, welche Objekte dargestellt werden müssen. Diese Region dient auch dazu, sogenannte Behaviors, die in Kapitel 2.3.4 erläutert werden, zu aktivieren.

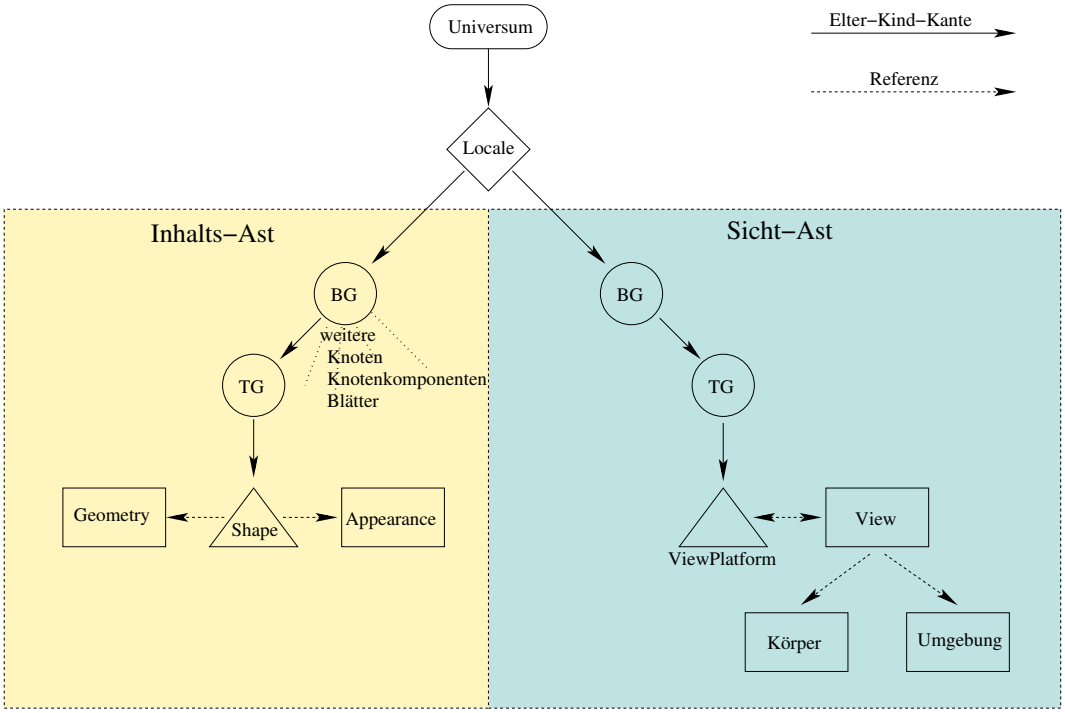

Abbildung 2: Inhalts- und Sicht-Ast eines Szenengraphen

- **Inhalts-Ast**: Er enthält die Geometrie, Position, Orientierung und das Aussehen der darzustellenden Objekte. Damit der Renderer nur eindeutige Objekte erzeugt, darf ein Knoten immer nur genau ein Elter besitzen. Der Zustand eines grafischen Objektes wird also durch die Information der Knoten beschrieben, die auf dem Pfad von der Wurzel bis zu diesem Objekt liegen. Ein Teilgraph, der in ein Universum eingefügt wird, ändert seinen Status. Er wird *lebendig*. Wenn ein Teilgraph *lebendig* wird, versucht das Java-3D Rendering System intern Optimierungen auszuführen, um die Performance der Darstellung zu erhöhen. Diese Optimierung bedeutet aber auch gleichzeitig, dass der Szenengraph nicht mehr in der Form vorliegt, in der er erzeugt wurde. Bestimmte Knoten können zu einem einzigen zusammengefasst werden. Damit ist eine spätere Manipulation dieser zusammengefassten Knoten nicht mehr möglich. Um dennoch zur Laufzeit Operationen auf einzelnen Knoten ausführen zu können, ist es möglich *Capability Bits* einzelner Knoten zu setzen. Diese *Capability Bits* ermöglichen es, bestimmte Werte einzelner Knoten im Szenengraph zur nachträglichen Änderung freizugeben. Ein Knoten beschreibt eine bestimmte Funktionalität. So können mit Hilfe eines TransformGroup-Knotens Position, Grösse und Ausrichtung der im Szenengraph unter ihm liegenden Objekte verändert werden.
  Die Methode "`setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE)`" erlaubt spätere Änderungen der Position. Auf diese Art müssen alle Funktionalitäten der Knoten freigeschaltet werden, die später noch geändert

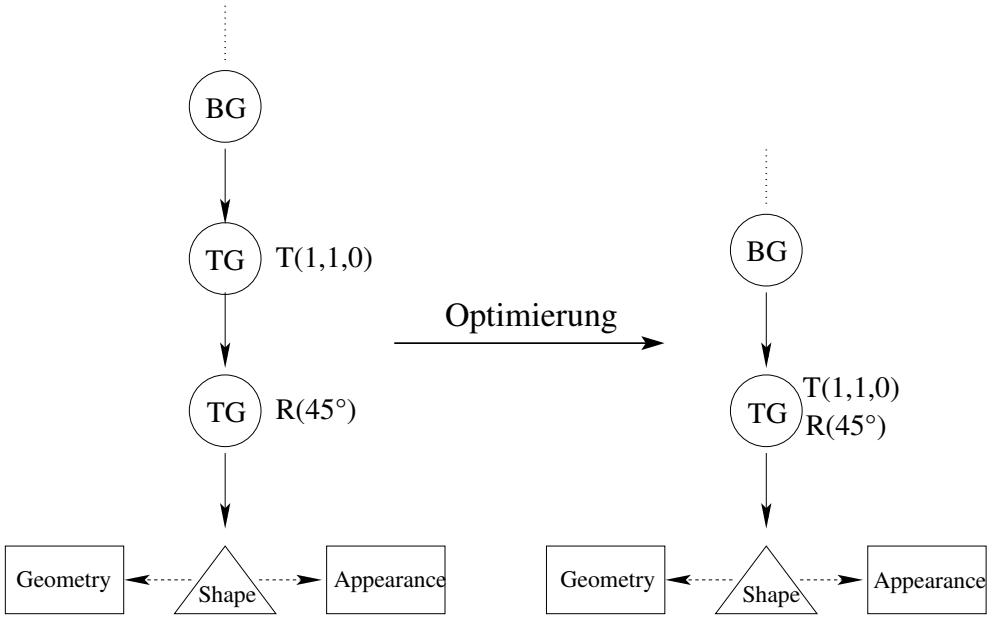

Abbildung 3: Optimierung des Renderers

werden sollen. Solche Änderungen zur Laufzeit sind die Grundlage einer dynamischen Szenengraphverwaltung. Sie wird in Kapitel 6 beschrieben.

In Java3D wird zwischen drei Klassen von Szenengraphelementen unterschieden:

- Gruppe
- Knotenkomponente
- Blatt

Diese Elementklassen und ihre wichtigsten Vertreter werden im folgenden beschrieben. Gruppen und Blätter sind ein echter Bestandteil des Szenengraph, d.h. sie sind durch Kanten und eine Elter-Kind-Relation miteinander verbunden. Knotenkomponenten dienen der näheren Beschreibung der anderen beiden Klassen. Wenn im weiteren von Knoten die Rede ist, so sind instanziierte Subklassen der oben erwähnten Klassen Gruppe oder Blatt gemeint.

2.3.2 Gruppen-Knoten

Die Gruppenknoten sind die einzigen Knoten, die Kinder besitzen können. Sie bestimmen also Tiefe und Breite des Szenengraphen. Diese Knoten lassen durch das Setzen der entsprechenden Capability Bits folgende Aktionen zu:

- `ALLOW_CHILDREN_EXTEND`: Hinzufügen von Kindern.
- `ALLOW_CHILDREN_READ`:
  - Auslesen der Anzahl der Kinder.

  - Referenzieren einzelner Kinder.

- `ALLOW_CHILDREN_WRITE`:
  - Ändern der Reihenfolge, in der die Kinder an einem Knoten hängen.
  - Ersetzen und Löschen von Kindern.

Vertreter dieser Klasse sind:

BranchGroup-Knoten: Ein BranchGroup-Knoten bildet die Wurzel eines Subgraphen. Er kann als einziger zur Laufzeit an einen anderen Gruppen-Knoten hinzugefügt oder von ihm gelöst werden. Also werden alle Operationen, die den Szenengraphen verändern, mit Hilfe dieses Typs vollzogen. Für die dynamische Verwaltung wird den oben aufgeführten Bits noch folgendes zugefügt: `ALLOW_DETACH`: Erlaubt es, diesen Knoten zur Laufzeit vom Szenengraph abzulösen. Voraussetzung dafür ist, dass das entsprechende Bit beim Elter gesetzt ist.

TransformGroup-Knoten: Dieser Knoten ist für die Positionierung, Orientierung und Grösse der unter ihm liegenden grafischen Objekte zuständig. Liegt ein solcher Knoten auf dem Pfad zwischen der Wurzel des Szenengraph und der ViewPlatform (siehe Abbildung 2) darf die Transformation nur kongruent sein. Im inhaltlichen Teil des Szenengraphs können Transformationen affin sein. Dieser Knoten enthält noch zwei zusätzliche Capability-Bits:

- `ALLOW_TRANSFORM_READ`: Auslesen der Transformation
- `ALLOW_TRANSFORM_WRITE`: Schreiben der Transformation

Die Transformationen werden durch ein Datenobjekt der Klasse *Transform3D* beschrieben. Dieses Objekt wird den TransformGroup-Knoten zugewiesen.
Ein Transform3D-Objekt wird intern als eine floating point 4x4 Matrix verwaltet. Mit Hilfe dieses Objektes werden die in Kapitel 2.2 beschriebenen Transformationen umgesetzt. Dem Programmierer wird damit die Arbeit abgenommen, einzelne Felder der Matrix zu ändern. Ein Transform3D-Objekt enthält ausserdem die entsprechenden Methoden, um ihre Werte zu ändern. Diese Klasse bietet eine grosse Anzahl von Methoden, um die Matrizenwerte der Klasseninstanz zu manipulieren. Hier ein kleiner Auszug:

- `invert()`: Invertiert den Inhalt der Matrix dieses Transform3D-Objektes.
- `mul(Transform3D t1)`: Multipliziert die zwei Matrizen der Transform3D-Objekte.
- `rotX(double angle)`: Setzt den Rotationswert der Matrix für die x-Achse.
- `setScale(double scale)`: Setzt die Skalierungswerte der Matrix. Hiermit wird eine uniforme Skalierung erzeugt.

- `setScale(Vector3d scale)`: Setzt die Skalierungswerte der Matrix. Dadurch erzielt man eine nichtuniforme Skalierung.
- `setTranslation(Vector3d trans)`: Setzt die Positionswerte der Matrix.

Eine zusammengesetzte Transformation der Form T*R*S wird also mit Hilfe eines TransformGroup-Knotens folgendermaßen ausgedrückt:

```
  Transform3D t3d = new Transform3D();

//positioniert das zu transformierende Objekt im Ursprung
  t3d.setTransform(new Vector3f(0,0,0));

//rotiert um 90°
  t3d.rotZ(Math.PI/2);

//skaliert das Objekt auf die Hälfte seiner Grösse
  t3d.setScale(0.5);

//fügt das t3d-Objekt in einen neuen TransformGroup-Knoten ein
  TransformGroup tg = new TransformGroup(t3d);

//fügt den TransformGroup-Knoten in den Szenengraph ein
  bg.addChild(tg);
  rootBg.addChild(bg);
```

Eine Alternative zu der obigen Vorgehensweise besteht darin, mehrere TransformGroup-Knoten zu benutzen.

```
//erzeugt einen neues Transform3D-Objekt
  Transform3D t3d = new Transform3D();

//positioniert das zu transformierende Objekt im Ursprung
  t3d.setTransform(new Vector3f(0,0,0));
//fügt das t3d-Objekt in einen neuen TransformGroup-Knoten ein
  TransformGroup translationTg = new TransformGroup(t3d);

//löscht bisherige Informationen aus dem Transform3D-Objekt
  t3d.setIdentity();
//rotiert um 90°
  t3d.rotZ(Math.PI/2);
//fügt das t3d-Objekt in einen neuen TransformGroup-Knoten ein
  TransformGroup rotationTg = new TransformGroup(t3d);

//löscht bisherige Informationen aus dem Transform3D-Objekt
  t3d.setIdentity();
//um 50% verkleinern
  t3d.setScale(0.5);
```

```
//fügt das t3d-Objekt in einen neuen TransformGroup-Knoten ein
  TransformGroup scaleTg = new TransformGroup(t3d);

//fügt die TransformGroup-Knoten in den Szenengraph ein
  translationTg.addChild(rotationTg);
  rotationTg.addChild(scaleTg);
  bg.addChild(translationTg);
  rootBg.addChild(bg);
```

Wenn auf einem Pfad, der bei der Wurzel beginnt, mehrere TransformGroup-Knoten liegen, so werden deren Inhalte (Matrizen) akkumuliert. Dies entspricht einer Matrizenmultiplikation wie sie in 2.2 geschildert wurde.
Die obige Betrachtungsweise ist etwas grob. Daher wird noch eine zweite Betrachtungsweise beschrieben. Um die Zusammenhänge zwischen mehreren TransformGroup-Knoten, die auf einem Pfad liegen, zu verstehen, betrachten wir TransformGroup-Knoten so, als würde bei jedem dieser Knoten ein Wechsel des Koordinatensystems stattfinden. Der Unterschied zur vorherigen Betrachtung liegt darin, dass die Transformationen nicht mehr in einem Koordinatensystem (Weltkoordinatensystem) beschrieben werden. Um den Wechsel in ein neues Koordinatensystem zu beschreiben, wird im folgenden wieder die Matrix betrachtet, die die Auswirkung eines TransformGroup-Knotens beschreibt. Es soll folgendes gelten:

- $M_{j\leftarrow i}$ ist die Transformation, die die Darstellung des Punktes von dem Koordinatensystem i in das Koordinatensystem j überführt.
- P_i, P_j sind Punkte im Koordinatensystem i und j.

Daraus folgt: $P_j = M_{j\leftarrow i} * P_i$.
Wenn $P_j = M_{j\leftarrow i} * P_i$ und $P_k = M_{k\leftarrow j} * P_j$ dann folgt:
$P_k = M_{k\leftarrow i} * P_i$, denn es gilt $M_{k\leftarrow i} = M_{k\leftarrow j} * M_{j\leftarrow i}$.

In Abbildung 4 sind 2 Systemwechsel beschrieben.
$M_{1\leftarrow 2} = T(3,2,0)$, $M_{2\leftarrow 3} = T(3,1,0) * S(0.5,0.5,1) * R(45°)$
Im 3. Koordinatensystem ist ein Punkt an der Position $(4,2,0)$ eingezeichnet.
P_2 ist dann: $P_2 = M_{2\leftarrow 3} * P_3$.
mit $P_2 = T(3,1,0) * S(0.5,0.5) * R(45°) * (4,2,0) = (3.7,3.1,0)$
Für P_1 gilt: $P_1 = M_{1\leftarrow 2} * M_{2\leftarrow 3} * P_3$.
Daraus folgt: $P_1 = T(3,2,0) * T(3,1,0) * S(0.5,0.5) * R(45°) * (4,2,0) = (6.7,5.1,0)$

Weiterhin gilt: $M_{j\leftarrow i} = M^{-1}_{i\leftarrow j}$.

Basierend auf der Betrachtung, dass bei jedem TransformGroup-Knoten auf einem Pfad ein Koordinatensystemwechsel stattfindet, werden in Kapitel 6 Szenengraphoperationen angewandt.

Weitere Vertreter dieser Gruppenklasse sind die Primitive. Folgende Objekte gehören zu den Primitiven:

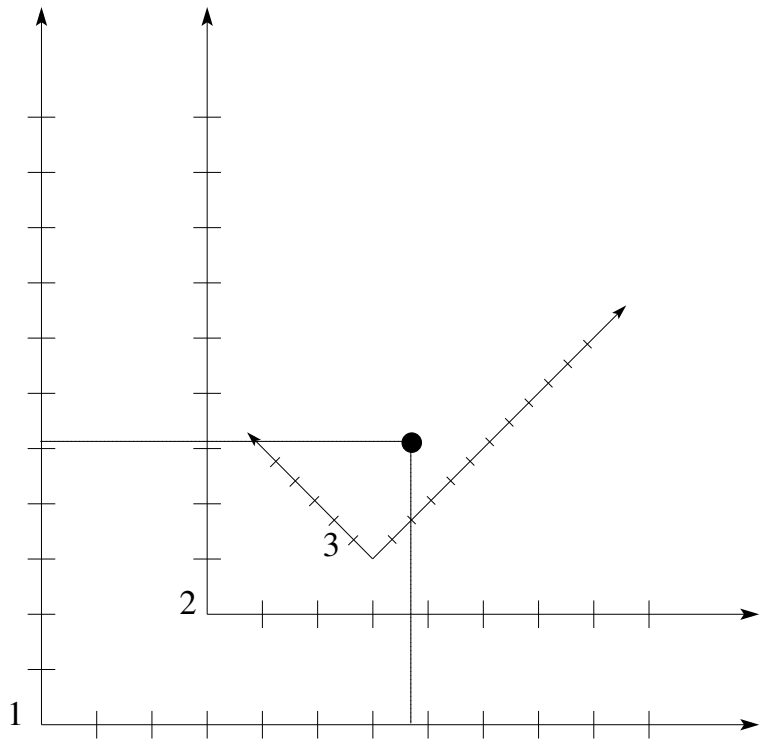

Abbildung 4: Wechsel des Koordinatensystems

- **Quader**: Ein Quader wird durch seine Ausdehnung in die x-,y- und z-Richtung beschrieben. Ferner kann ihm noch ein Aussehen (*Appearance*) zugeordnet werden. Wie dieses Aussehen gestaltet wird, wird in Kapitel 2.3.3 beschrieben.
- **Kugel**: Eine Kugel wird durch ihren Radius und das Aussehen beschrieben.
- **Zylinder**: Ein Zylinder wird durch Radius, Höhe und Aussehen beschrieben.
- **Kegel**: Ein Kegel wird wie ein Zylinder beschrieben. Der Radius ist jedoch der Radius des Bodens. Von dort aus läuft der Kegel bis zur angegebenen Höhe spitz zusammen.

Diese Primitive werden in dieser Arbeit nicht als Vertreter der Gruppenklasse behandelt. Vielmehr werden sie so behandelt, als würden sie der Blattklasse angehören. In der Blattklasse befindet sich unter anderem der *Shape3D*-Knoten, der auch dazu benutzt wird, Objekte grafisch darzustellen. Dies vereinheitlicht die Handhabung.

2.3.3 Knotenkomponenten

Die Knotenkomponenten sind das Salz in der Suppe. Die Knoten bilden nur die Grobstruktur. Um diese Struktur zu verfeinern, gibt es die Knotenkomponen-

ten. Knotenkomponenten werden von Knoten referenziert, die mehr Informationen zum Rendern bedürfen. Eine Box wird normalerweise einfach nur weiss dargestellt. Ihre Oberfläche besitzt keine weiteren Eigenschaften, die der Renderer verarbeiten könnte. Um diese Box nun genauer zu beschreiben, gibt es eine Knotenkomponente namens *Appearance*. Im weiteren werden einige Knotenkomponenten beschrieben und erläutert, von welchen Knoten sie benötigt werden:

Die Knotenkomponente **Appearance** dient zur genaueren Beschreibung eines 3D-Objektes. Im vorherigen Kapitel wurden vier Objekte vorgestellt (Quader, Kugel, Zylinder, Kegel). Sie werden durch *Appearance* näher beschrieben. Ausserdem wird in Kapitel 2.3.4 noch der *Shape3D*-Knoten vorgestellt, der auch durch *Appearance* näher beschrieben wird. Ein *Appearance*-Objekt bietet unter anderem folgende Eigenschaften:

- Farbe: Legt die Farbe eines unbeleuchteten Objektes fest.
- Schattierungsmodell: Es gibt verschiedene Schattierungsmodelle, die sich in ihrem Berechnungsaufwand und ihrer Genauigkeit unterscheiden.
- Transparenz: Ein Wert im Intervall [0,1], der beschreibt wie durchsichtig ein Objekt dargestellt werden soll.
- Material: Hiermit werden Eigenschaften des Objektes beschrieben, wenn es in der Reichweite einer Lichtquelle liegt.
- Textur: Eine Grafik, die das Objekt oder einen Teil des Objektes bedecken soll.

Wie auch im vorherigen Kapitel gibt es hier Capability-Bits, die es ermöglichen, zur Laufzeit einzelne Eigenschaften zu ändern.

Die Knotenkomponente **Geometry** dient dazu, die Geometrie eines *Shape3D*-Objektes festzulegen. Die von *Geometry* abgeleiteten Klassen unterstützen verschiedene Arten, ein Objekt zu beschreiben. Sie unterscheiden sich in der Redundanz, in der vordefinierte Punkte vom Renderer wiederverwendet werden können. Hier werden einige dieser Klassen und deren Subklassen exemplarisch erläutert.
GeometryArray: Hiermit können einfache Geometrien kreiert werden. Alle Punkte werden nur einmal verwendet, was für grössere Flächen ungünstig ist. Die erzeugten Flächen werden automatisch gefüllt.

- PointArray: Beschreibt einzelne Punkte im Koordinatensystem.
- LineArray: Je zwei Punkte werden durch eine Linie verbunden.
- TriangleArray: Je drei Punkte des Koordinatenfeldes werden verbunden und bilden ein Dreieck.

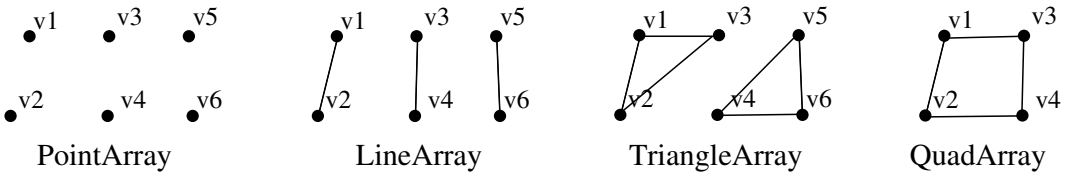

Abbildung 5: GeometryArray-Subklassen und ihre Darstellung

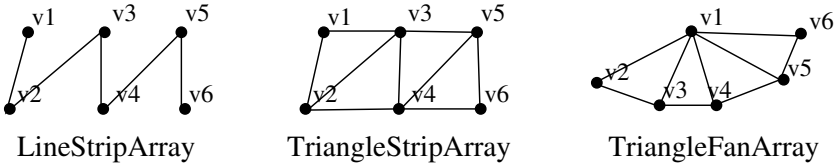

Abbildung 6: GeometryStripArray-Subklassen und ihre Darstellung

- QuadArray: Je vier Punkte des Koordinatenfeldes werden verbunden und bilden ein Viereck. Die Punkte für je ein Viereck müssen in einer Ebene liegen.

GeometryStripArray: Mit Hilfe der Subklassen dieser Klasse können Flächen konstruiert werden, die sich Punkte teilen. Die Punkte werden in der Reihenfolge verarbeitet, in der sie im Feld vorliegen. Ein *Strip* ist durch die Anzahl der Punkte definiert, die er verwendet. Werden für ein 8-Koordinaten Feld 2 Strips mit 2 und 6 Punkten definiert, so wird zuerst ein Strip aus den ersten beiden Feldelementen konstruiert und dann ein Strip mit den restlichen 6 Feldelementen.

- LineStripArray: Konstruiert verbundene Linien.
- TriangleStripArray: Erzeugt Flächen aus Dreiecken. Grenzen zwei Dreiecke mit einer Seite aneinander, so werden die Punkte, durch die diese Seite beschrieben wird, für beide Dreiecke verwendet. Dies bedeutet weniger Arbeit für den Renderer.
- TriangleFanArray: Das selbe Prinzip wie bei TriangleStripArray, nur dass die Dreiecke um einen Punkt angeordnet werden.

IndexedGeometryArray: Die obigen Klassen lassen es nur beschränkt zu, Punkte wieder zu verwenden. Die Subklassen dieser Klasse erlauben eine Indizierung einzelner Punkte. Somit werden die gleichen Punkte von mehreren Objekten benutzt.

- IndexedPointArray: Aus dem Koordinatenfeld können Punktmengen durch die Indices erzeugt werden.
- IndexedLineArray: Hiermit können einzelne Liniensegmente definiert werden.

- IndexedTriangleArray: Je drei indizierte Punkte werden zu einem Dreieck zusammengeschlossen.
- IndexedQuadArray: Analog zu IndexedTriangleArray.

Als Beispiel betrachten wir einen Würfel. Mit Hilfe von IndexedQuadArray ist es ausreichend ein Feld mit 8 Koordinaten zu erzeugen, während mit QuadArray ein Feld mit 24 Elementen notwendig gewesen wäre. Da jeder Punkt vom Renderer abgearbeitet werden muss, ist die erste Lösung eindeutig die bessere.

2.3.4 Blatt-Knoten

Ein Blatt bezeichnet einen Knoten der keine Kinder besitzt. Vertreter dieses Typs sind:

- Shape3D
- Light
- Behavior

Shape3D: Ein Shape3D-Knoten referenziert zwei Knotenkomponenten, um ein Objekt zu spezifizieren. Die erste Knotenkomponente dient dazu, die Geometrie zu gestalten und stammt aus der oben erläuterten *Geometry*-Klasse. Die zweite Knotenkomponente ist die *Appearance*-Knotenkomponente. Sie verleiht der konstruierten Geometrie ein Aussehen.

Light: Es gibt verschiedene Arten (Klassen) von Licht. Das Licht hat nur dann Auswirkungen auf Objekte in einer Szene, wenn diese Objekte eine *Material*-Knotenkomponente referenzieren. Sie beinhaltet die Werte, die der Renderer verwendet, um die Auswirkungen einer Lichtquelle zu berechnen. Alle Lichtquellen besitzen eine Position, Farbe und Reichweite. Die Arten sind:

- AmbientLight: Dieses Licht ist an allen Stellen in seinem Definitionsbereich gleichstark und wirkt von allen Seiten auf die Objekte ein.
- DirectionalLight: Dieses Licht strahlt parallel in eine Richtung.
- PointLight: Dieses Licht besteht aus einer Lichtquelle, die gleichmäßig in alle Richtungen strahlt.
- SpotLight: Ein Spotlight strahlt kegelförmig aus.

Behavior: Die bisherigen Knotentypen und Knotenkomponenten dienten lediglich zur Erstellung eines statischen Szenengraph. Es fehlen noch Möglichkeiten, 3D-Objekte zu bewegen. Dies kann in Interaktion mit dem Benutzer oder eigenständig durch den Algorithmus geschehen. Hierzu gibt es die Behavior-Knoten. Sie erlauben es, Objekte zu animieren, sowie Key- und Maus-Events zu verarbeiten. Alle Behavior-Knoten besitzen unabhängig von ihrer Funktionalität bestimmte Gemeinsamkeiten. Sie spezifizieren eine sogenannte "scheduling region". Diese beschreibt einen Raum, in dem ein Behavior-Knoten aktiv

ist. Betritt ein Betrachter diesen Raum, schneidet also der Aktivierungsradius der View-Platform den scheduling-Radius des Behavior-Knoten, so wird dieser aktiviert. Um nun das *Behavior* dieses Knotens auszulösen, muss noch ein weiteres Kriterium zutreffen, das Wakeup-Kriterium. Zu den Wakeup-Kriterien gehören:

- WakeupOnActivation: Das Behavior wird nur durch seine Aktivierung ausgelöst.
- WakeupOnBehaviorPost: Das Behavior wird durch den Event eines anderen Behavior-Knotens ausgelöst.
- WakeupOnCollisionEntry: Das Behavior wird ausgelöst wenn ein in ihm spezifiziertes Objekt mit einem anderen in der Szene kollidiert.

Kriterien können auch kombiniert werden, um eine Aktion auszulösen.
Die Behavior-Knoten lassen sich in zwei Gruppen unterteilen:

- Interaktions-Knoten: Mit Hilfe dieser Knoten wird eine Interaktionsmöglichkeit hergestellt, um mit der Maus oder der Tastatur Objekte in der Szene zu manipulieren.
- Interpolations-Knoten: Durch diese Knoten können Eigenschaften von Gruppen-Knoten oder Knotenkomponenten in einem definierten Zeitintervall kontinuierlich verändert werden. Einer Interpolation liegt eine Funktion zu Grunde, die aus einem Zeitwert Alpha-Werte im Interval [0..1] berechnet. Mit Hilfe dieser Alpha-Werte werden Stützpunkte berechnet. Diese Stützpunkte dienen dazu, die Eigenschaft (Farbe, Transparenz, Position, Rotation etc.) von einem Startzustand in den Endzustand zu überführen. Weiterhin gibt es verschiedene Parameter um das Zeit-Alpha-Mapping zu beeinflussen.

Weitere Informationen zu dem Java-3D API gibt es unter [Bou99] und [BP98].
In dem zurückliegenden Kapitel wurde die Java-3D API beschrieben, mit deren Hilfe die vorliegende Arbeit erstellt wurde. Insbesondere mit den Gruppen-Knoten werden wir uns in Kapitel 6 noch beschäftigen.

2.4 Alternative Darstellungsmöglichkeiten

In Kapitel 2.3 wurde erklärt, aus welchen Elementen ein Szenengraph in Java3D aufgebaut ist. Es ist nicht sehr komfortabel, in Java3D eine Szene zu kreieren, dafür ist aber das API sehr mächtig und bietet mannigfache Möglichkeiten.
Einen anderen Ansatz bietet VRML (Virtual Reality Modeling Language). VRML beschreibt ein standardisiertes Dateiformat, mit dessen Hilfe Szenen beschrieben werden können. Auch hier liegen wie in Java3D die Informationen in einem Szenengraph vor. Für die Darstellung eines Szenengraphen ist jedoch ein eigener VRML Browser nötig. Dieser Browser bietet begrenzte Interaktionsmöglichkeiten an. Weitergehende Interaktionen können über Skriptknoten im Szenengraph mittels Javascript und über ein Java-Interface, das *EAI (External Authoring Interface)*, integriert werden.

Um den Nachteil des Plugins, das auch gleichzeitig Plattformabhängigkeit bedeutet, zu eliminieren, gehen die Anbieter dazu über anstatt Plugins Applets zu verwenden. Ein solches Applet enthält einen in Java implementierten Renderer. Dieser Renderer arbeitet auf einem Szenengraphen, der aus einer Teilmenge der in VRML bekannten Knoten besteht. Dieser Teilmenge liegt ein noch nicht verabschiedeten Standard zugrunde, der X3D(Extensible 3D)-Standard. Einige Firmen haben die Vorschläge, die sie zu diesem Standard beim Web3D-Consortium eingereicht haben, in diesen Applets, bzw. in den Entwicklungsumgebungen implementiert. Einer Entwicklungsumgebung liegt ein API bei, mit dessen Hilfe auf einen Szenengraph zugegriffen werden kann, um die Knoten zu manipulieren. Solche Entwicklungsumgebungen werden unter anderem von folgenden Firmen angeboten:

- Shout Interactive Inc.
- Blaxxun Interactive Inc.
- Parallel Graphics Inc.

Diese Applets haben jedoch den Nachteil, dass sie die etwaig vorhandenen Grafik-Resourcen nicht vollständig ausnutzen können, da sie kein OpenGL unterstützen, im Gegensatz zu Java3D. Dies führt bei komplexeren Szenen zu einer drastischen Verlangsamung bei Interaktionen oder Animationen. In der nachfolgenden Tabelle werden einige wesentliche Vor- und Nachteile der verschiedenen Darstellungsmöglichkeiten zusammengefasst:

	positive Aspekte	negative Aspekte
VRML (Applet)	-plattformunabhängig	-keine Hardwarebeschleunigung -begrenzte Möglichkeiten durch kleines API
VRML (Plugin)	-Hardwarebeschleunigung	-plattformabhängig -VRML Spezifikation ist nicht in allen Browsern gleich implementiert
Java3D	-umfassendes API -Hardwarebeschleunigung -plattformunabhängig	-noch in der Entwicklungsphase

2.5 XML

In der vorliegenden Arbeit werden Objekte visualisiert. Die Objektdaten werden mit den Tools, die in Kapitel 4 beschrieben werden erfasst und mit Hilfe von Java3D visualisiert. Um die erfassten Daten in späteren Anwendungen wiederverwerten zu können, müssen sie in irgendeiner Form abgespeichert werden. Hier bietet sich XML (Extensible Markup Language) an.

2.5.1 Was ist XML?

XML ist eine Markup-Sprache, die vom W3-Konsortium entwickelt wurde, um strukturierte Dokumente oder Daten im Internet benutzen zu können. XML ist

nicht einfach eine weitere Markup-Sprache wie HTML. XML ist eine Metasprache, d.h. mit XML lassen sich Markup-Sprachen definieren, mit deren Hilfe Dokumente erzeugt werden können. Mit XML lassen sich die Struktur und die Bedeutung von Dokumenten beschreiben. Ziel der Entwicklung von XML war es, eine Sprache zu entwickeln, die die Möglichkeit und Leistungsfähigkeit von SGML (Standard Generalized Markup Language) als allgemeine Auszeichnungssprache im Web bietet. Sie sollte jedoch nicht so komplex wie SGML sein. Ein Anlass zur Entwicklung von XML war unter anderem der, dass HTML den Anforderungen nicht mehr genügte. HTML konnte mit der rasanten Entwicklung des Internets nicht Schritt halten. Es wurde immer wieder versucht durch immer mehr Befehle oder durch diverse Scriptsprachen, HTML mehr Möglichkeiten zu geben. Die Grenzen der Sprache konnten aber nicht überwunden werden. Die fest definierte Menge an Befehlen konnte kaum die Fantasien der Webentwickler auf allen Gebieten befriedigen.
Zunächst dachte man an SGML. Anders als HTML ist SGML eine Metasprache, mit der man Regeln zum Schreiben von Dokumenten mit komplizierten Strukturen festlegen kann. Ein Dokumenttyp, der mit SGML definiert wurde, lässt sich beliebig erweitern. SGML wurde in den 60er Jahren entwickelt, in denen auch das Prinzip der allgemeinen Markup-Sprachen erfunden wurde. Die Tatsache, dass die letzte Spezifikation von SGML aus dem Jahr 1986 fast ohne jegliche Änderung ein Jahrzehnt überlebt hat, zeigt ihre Robustheit. Allerdings darf man gewisse Nachteile von SGML auch nicht übersehen. Da die Entwickler von Anfang an versuchten, die Sprache so allgemein wie möglich zu halten, gibt es in der Sprache eine Unmenge von optionalen Features. Deshalb ist die Entwicklung von SGML-Anwendungen nicht nur sehr zeitintensiv, sondern auch sehr fehleranfällig. Der Grund, warum die Browserhersteller SGML nicht in die Browser integrieren wollten, war ebenfalls die Komplexität der Sprache.
So hat das W3-Konsortium 1996 damit begonnen, eine neue Sprache für das Web zu entwickeln. Die neue Sprache soll die Vorzüge von HTML und SGML verbinden und deren Schwächen überwinden. Seit Februar 1998 liegt die erste XML-Version vor. Mit XML lassen sich komplizierte Dokumentstrukturen realisieren. Datenbanken können mit Hilfe von XML Daten miteinander austauschen. Ein grosses Einsatzgebiet von XML ist das Internet. Die Webentwickler können eigene Dokumenttypen entwickeln und damit Dokumente erstellen. Mit der selbstbeschreibenden Struktur kann effizienter auf Elemente zugegriffen werden. Parallel zur Entwicklung der XML-Spezifikation werden noch weitere Standards entwickelt. Sie basieren auf XML und ihre Syntax kann mit XML komplett beschrieben werden. So wird die Zusammenarbeit und Kompatibilität untereinander garantiert. So soll XLink flexiblere Linkmechanismen ermöglichen, als man sie von HTML kennt. Das DOM (Dokument Object Modell) soll ein einheitliches Objektmodell für HTML- und XML-Dokumente schaffen. SMIL soll das Erstellen von aufwendigen Multimediapräsentationen im Web ermöglichen. XSL soll dabei helfen, XML-Datenstrukturen in HTML-Form darzustellen.
Detaillierte Informationen zu XML sind unter [BPS00] zu finden.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!NOTATION ISODATUM SYSTEM
   "http://www.cl.cam.ac.uk/~mgk25/iso-time.html">
 <!NOTATION DTDATUM SYSTEM
   "http://www.dummy.de/germandate.html">
 <!ENTITY INTRO "misc/logo.gif">
 <!ELEMENT ADRESSBUCH PERSON*>
 <!ELEMENT PERSON (NAME, STRASSE?, ORT?, GEBDATUM?, TELNUM*)>
 <!ELEMENT NAME (#PCDATA)>
 <!ELEMENT STRASSE (#PCDATA)>
 <!ELEMENT ORT EMPTY>
  <!ATTLIST ORT
    PLZ CDATA #REQUIRED
    NAME CDATA #REQUIRED>

 <!ELEMENT GEBDATUM (#PCDATA)>
  <!ATTLIST GEBDATUM
    FORMAT NOTATION (ISODATUM | DTDATUM) #IMPLIED>

 <!ELEMENT TELNUM (#PCDATA)>
```

Abbildung 7: Adressbuch-DTD

2.5.2 Die DTD

Um Dokumente eines Typs zu erstellen muss es Regeln geben, nach denen sie erstellt werden können. Ein solches Regelwerk wird durch eine DTD (Document Type Definition) beschrieben.
In Abbildung 7 ist eine DTD dargestellt, die zur Adressverwaltung dient. Jede Person wird durch Name, Strasse, Wohnort, Geburtsdatum und Telefonnummer beschrieben.
In Abbildung 8 ist eine Instanz der Adressbuch-DTD dargestellt. Bei Dokumenten kann zwischen zwei verschiedenen Arten unterschieden werden:

- wohlgeformt: Ein XML-Dokument ist wohlgeformt, wenn es mit einem verständliches Markup versehen ist. In diesem Fall muss auch keine DTD vorhanden sein. Falls jedoch komplexere Dokumente erstellt werde sollen ist es hilfreich Produktionsregeln in einer DTD zu beschreiben.

- gültig: Ein XML-Dokument ist gültig, falls es in seinem Aufbau konform mit einer vorliegenden DTD ist. Eine DTD kann entweder direkt in der Dokumentdeklaration angegeben werden, d.h. die Produktionsregeln stehen im Dokument selbst, oder die Regeln stehen in einer gesonderten Datei und werden in dem Dokument nur referenziert (wie im vorliegenden Fall). Wir sprechen hierbei von internen und externen Untermengen in der Dokumentdeklaration. Eine Kombination ist ebenfalls möglich.

Eine DTD besteht aus einer Reihe von Entities, Notationen und Elementtypen. Zuerst sollen die Entities behandelt werden.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE  ADRESSEN SYSTEM "adress.dtd">
<ADRESSBUCH>
 <PERSON>
  <NAME>R. Lemmle<NAME>
  <STRASSE> Winterstr. 10 </STRASSE>
  <ORT
   PLZ="43538"
   NAME="Obershausen">
  </ORT>
  <GEBDATUM FORMAT="DTDATUM"> 01.01.1948 </GEBDATUM>
  <TELNUM> 302345679 </TELNUM>
  <TELNUM> 302000987 </TELNUM>
 </PERSON>
 <PERSON>
  <NAME>S. Stuck<NAME>
  <STRASSE> Schulstr. 12B </STRASSE>
  <ORT
   PLZ="54332"
   NAME="Salmfeld">
  </ORT>
  <GEBDATUM FORMAT="DTDATUM"> 08.07.1960 </GEBDATUM>
  <TELNUM> 067843299 </TELNUM>
 </PERSON>
</ADRESSBUCH>
```

Abbildung 8: Adressbuch-Instanz

- Interne geparste allgemeine Entities: Diese Entities stellen den einfachsten Typ dar. Sie können als Kürzel im Dokument und in anderen allgemeinen Entities verwendet werden und Markup enthalten. Eintrag in der DTD:
```
<!ELEMENT STUDIENPLATZ (#PCDATA)>
<!ENTITY UNI "Universität des Saarlandes">
<!ENTITY FR "Fachrichtung Informatik">
<!ENTITY SP "&FR; an der &UNI;">
```

  Eintrag im Dokument:
```
<STUDIENPLATZ>&SP;</STUDIENPLATZ>
```

- Externe geparste allgemeine Entities: Mit diesen Entities verhält es sich wie mit den internen Entities, nur dass die Daten nicht in der selben Datei stehen. Sie werden durch einen URI referenziert. So könnten die obigen Daten auch in einer getrennten Datei "studienplatz.dat" stehen.
  Inhalt von "studienplatz.dat":
  Fachrichtung Informatik an der Universität des Saarlandes

  Eintrag in der DTD: `<!ENTITY SP SYSTEM "studienplatz.dat">`

  Die Verwendung dieses Entity ist dann wie im obigen Beispiel. Auf das Schlüsselwort `SYSTEM` folgt ein System-Identifier. Hierbei handelt es sich um eine URI, der verwendet wird, das Entity aufzufinden. Er kann relativ zu dem Dokument sein, in dem er definiert ist, oder absolut. Alternativ hierzu gibt es noch das `PUBLIC`-Schlüsselwort. Auf das Schlüsselwort folgt ein Public- und ein System-Identifier. In diesem Fall versucht der XML-Prozessor den Inhalt des Entity mit Hilfe des Public-Identifier aufzufinden. Dazu versucht der XML-Prozessor aus dem Public-Identifier einen gültigen URI zu gewinnen. Falls der Prozessor dazu nicht in der Lage ist, muss er den als System Identifier angegebenen URI verwenden.

- Nicht geparste Entities: Bei diesem Entity-Typ können externe Daten eingebunden werden, ohne dass sie geparst werden. Dies können z.B. Bild-Dateien seien. Über `NOTATION` wird der verarbeitenden Applikation mitgeteilt, um welches Format es sich handelt. Im Entity muss daher auf diese Deklaration verwiesen werden, z.B.:
```
<!NOTATION GIF SYSTEM "c:\programme\viewer\viewer.exe">
<!ENTITY WAPPEN SYSTEM "logo.gif" NDATA GIF>
```

  Wie ein solches Entity in einem Dokument angewendet wird, wird weiter unten erklärt.

- Interne Parameter-Entities: Diese Entities können nur innerhalb der DTD verwendet werden. Sie haben die folgende Schreibweise:
  `<!ENTITY % name "Ersetzungstext">`. Sie unterscheiden sich also von den allgemeinen Entities durch das %-Zeichen und können nicht in einem

XML-Dokument angewendet werden. Vor dem Parsen wird zuerst die Ersetzung der Parameter-Entities vorgenommen, während dies bei den allgemeinen Entities nicht geschieht. Daher muss auch im Gegensatz zu den allgemeinen Entities vor der Referenzierung das Entity zuerst deklariert werden. Beispiel:

```
<!ENTITY % EL "<!ELEMENT EINELEMENT (#PCDATA)">
 %EL;
```

- Externe Parameter-Entities: Diese Entities erlauben eine flexible Gestaltungen von DTDs. Angenommen es gibt zwei DTD-Dateien "buch.dtd" und "zeitschrift.dtd", so können diese einfach in einer DTD durch folgenden Ausdruck zusammengefügt werden.

```
<!ENTITY % BUCH SYSTEM "buch.dtd">
<!ENTITY % ZEITSCHRIFT SYSTEM "zeitschrift.dtd">
%BUCH;
%ZEITSCHRIFT;
```

Falls in den geparsten Entities Markup verwendet wird, ist darauf zu achten, dass Start- und End-Tag im selben Entity vorkommen.
Mit Hilfe der Notationsdeklaration können Datenformate spezifiziert werden, die nicht durch den XML-Prozessor geparst werden können. Sollen zum Beispiel Bild- oder Audiodaten in einem XML-Dokument eingebunden werden, wird mit Hilfe der Notationsdeklaration einer Applikation mitgeteilt, wie sie dieses Format behandeln soll. Beispiel:

```
<!NOTATION GIF SYSTEM "c:\programme\viewer\viewer.exe">
```

Die Komponenten einer DTD, mit deren Hilfe eine Struktur erstellt werden kann, sind die Elementtypen. Sie sind für die Gliederung zuständig und legen die Produktionsregeln fest. Elementtypen werden in einem XML-Dokument instanziiert, indem sie als Tags verwendet werden. Ein wohlgeformtes oder gültiges Dokument kann in einer Baumstruktur dargestellt werden. In Abbildung 9 ist eine solche Baumstruktur veranschaulicht. Ein Element ist dabei das *Wurzelelement*. In Abbildung 8 und 9 ist die Instanz des Elementtyps `ADRESSBUCH` das Wurzelelement. Dieses Element enthält die Elemente `NAME, STRASSE etc.`. Diese Elemente können ihrerseits auch wieder Elemente enthalten. Die Elemente, die selbst Unterelemente sind und weitere Unterelemente beinhalten, werden als *Zweige* bezeichnet. Diejenigen, die keine Unterelemente beinhalten, sind die *Blätter*.

Es gibt verschieden Arten, den Inhalt eines Elementtyps zu bestimmen. :

- EMPTY: Dieser Spezifikationstyp erlaubt keinen Inhalt. `EMPTY` wird für die Elementtypen verwandt, die nur Attribute verwenden. Attribute werden noch später in diesem Kapitel vorgestellt.

- ANY: Dieser Spezifikationstyp erlaubt jeden Inhalt, d.h. Zeichenketten und weitere Unterelemente.

- kombinierter Inhalt: In diesem Fall werden Zeichenketten und bestimmte Elemente kombiniert, z.B.:
  `<!ELEMENT NAME (#PCDATA | VORNAME | NACHNAME)>`. PCDATA steht für Parsed Character Data. Diese Daten werden geparst und evtl. enthaltene Tags ausgewertet.

- Elementinhalt: Hier sind nur Unterelemente erlaubt, z.B.:
  `<!ELEMENT ARTIKEL (EINLEITUNG, KAPITEL*, FAZIT)>`

Weitere Möglichkeiten Dokumente zu strukturieren bieten Elementtypen durch Festlegung von Sequenzen (',') oder Alternativen ('|') . So können Reihen von Elementen vorgeschrieben oder flexibel festgelegt werden. Weitere Indikatoren können Teilausdrücken auf der rechten Regelseite nachgestellt werden. Ein '?' bedeutet, dass der Teil des Dokumentes, der durch diesen Elementtyp produziert wird, optional ist. D.h. er kommt ein- oder keinmal im Dokument vor. Ein '*' bewirkt, dass ein Teil des Dokumentes keinmal oder beliebig oft vorkommen kann. Ein '+' verlangt, dass der Dokumentteil wenigstens einmal vorkommen muss.
Elemente können zusätzlich noch mit Attributen versehen werden. Diese beschreiben Eigenschaften der Elemente. In Abbildung 7 werden Attribute z.B. dazu genutzt, Eigenschaften des Elementtyps `ORT` zu beschreiben. Das nachgestellte Schlüsselwort `#REQUIRED` bedeutet, dass diese Eigenschaft im Dokument angegeben werden muss. Andere Schlüsselworte sind `#IMPLIED` und `#FIXED`. `#IMPLIED` bedeutet, dass diese Eigenschaft vom Autor des Dokumentes nicht angegeben werden muss. Stattdessen kann ein, das Dokument verarbeitendes Programm, den Wert einsetzen. Das Schlüsselwort `#FIXED` bedeutet, dass der Autor des Dokuments die in der DTD vorgegebene Eigenschaft angeben muss. So wird durch den Eintrag `<!ATTLIST BUCH ERSCHEINUNGSORT CDATA #FIXED "London">` in der DTD der Autor dazu veranlasst, "London" in seinem Dokument als Erscheinungsort anzugeben. Tut er es nicht, wird der Standardwert automatisch eingesetzt.
Eine weitere Eigenschaft von Attributen ist, dass sie Typen besitzen. Durch diese Typen können die Eingaben im Dokument beschränkt werden. Es gibt 9 verschiedene Typen:

- `CDATA`: Dies ist der einfachste Attributtyp. Wurde ein solcher Typ in der DTD verwendet, können im Dokument beliebige Zeichenstrings benutzt werden. '<', ' " ' und '&' müssen jedoch durch ihre gebräuchlichen Entity-Referenzen angegeben werden.

- `NMTOKEN`: Namenstoken sind auf eine bestimmte Menge von Zeichen beschränkt. Es werden Ziffern, Buchstaben und die Sonderzeichen '.', '−', '_' und ':' erlaubt.

- `NMTOKENS`: Bei der Verwendung diese Typs ist es im Dokument möglich einem Attribut mehrere Namenstoken zuzuordnen (Leerzeichen sind in

einem Namenstoken nicht erlaubt). Zum Beispiel soll einer Person eine Liste von Ortsbezeichnungen zugewiesen werden können, in der sie sich in den letzten Monaten aufhielt.
Eintrag in der DTD:
`<!ATTLIST PERSON ORTE NMTOKENS #REQUIRED>`
Der entsprechende Dokumenteintrag könnte dann folgendermaßen lauten:
`<PERSON ORTE="SB M B SLS SB">`
Hierbei muss jedes Ortskürzel ein Namenstoken sein.

- Aufzählungen: Bei Aufzählungen werden in der DTD verschiedene Alternativen für ein Attribut bereitgestellt, sowie ein Defaultwert. Wird das Attribut in dem Dokument nicht angegeben, so wird der Defaultwert angenommen. In der DTD könnte zum Beispiel folgendes Attribut definiert sein: `<!ATTLIST PERSON AUGENFARBE (BLAU | BRAUN | GRÜN) "BRAUN">`
Ein entsprechend Eintrag in einem Dokument wäre:
`<PERSON AUGENFARBE="BLAU">`

- `ID, IDREF`: Diese Typen erlauben es, in einem Dokument Verweise einzubetten. Ein Element, das ein Attribut mit `IDREF` und einem dazugehörigen Namen enthält, verweist auf das Element, das ein Attribut mit `ID` und dem entsprechenden Namen besitzt. Betrachten wir einen Stammbaum. Hier soll die Eltern-Kind-Beziehungen dargestellt werden.
Elementtypeintrag in der DTD:

```
<!ELEMENT PERSON EMPTY>
  <!ATTLIST PERSON
    NAME CDATA #REQUIRED
    PID ID #REQUIRED
    VATER IDREF #REQUIRED
    MUTTER IDREF #REQUIRED>
```

Elementeintrag im Dokument:

```
<PERSON NAME="Markus Meier">
        PID="mame"
        VATER="peme"
        MUTTER="lime">
<\PERSON>
```

- `IDREFS`: Dieser Typ lässt mehrere Referenzen zu. Die Schreibweise ist analog zu dem Typ `NMTOKENS`.

- `NOTATION`: Mit Hilfe dieses Attributtyps wird das Datenformat beschrieben, das dieses Attribut im Dokument enthält. Es können ein oder mehrere Werte angegeben werden, die in der DTD deklariert wurden (siehe Abbildung 7 und 8).

- `ENTITY`: Um Verweise auf externe Datenobjekte anzulegen, werden in einer DTD `ENTITIES` verwendet. Mit `NOTATION` wird angegeben, wie die Applikation, die das Dokument verarbeitet, mit dem Entity-Typ verfahren soll.

Eintrag in der DTD:

```
<!NOTATION GIF SYSTEM "c:\programme\viewer\viewer.exe">
<!ENTITY WAPPEN SYSTEM "logo.gif" NDATA GIF>
<!ELEMENT PERSON EMPTY>
  <!ATTLIST PERSON BILD ENTITY #REQUIRED>
```

Eintrag in XML-Dokument:

```
<!PERSON BILD="WAPPEN">
```

- `ENTITIES`: Dieser Typ erlaubt es, mehrere externe Datenobjekte zu benutzen.

2.5.3 Das DOM

XML-Dokumente repräsentieren aufgrund ihrer Wohlgeformtheit Bäume. Die Knoten sind die oben erläuterten Elemente. In der Hierarchie dieser Bäume lässt sich die Schachtelung der Elemente ablesen. Nach dem Parsen wird ein Dokument in dieser Form repräsentiert. Hierzu gibt es eine Objektdefinition, mit deren Hilfe dieser Baum darzustellen ist. Durch generische Operationen kann auf Knoten des Baumes zugegriffen.
Hierzu wurde vom w3c ein Standard, das *"Document Object Model"* (DOM) entwickelt. Das DOM ist ein API für gültiges HTML und wohlgeformte XML-Dokumente. Es bietet die Möglichkeit Elemente hinzuzufügen, zu löschen oder zu modifizieren. Das Interface wurde unabhängig von einer Programmiersprache entwickelt. Somit wird es in allen gängigen Programmiersprachen möglich sein, ein solches Interface zu implementieren, um mit ihm XML-Dokumente manipulieren zu können. Java- und Javascript-Interfaces sind wegen ihrer Plattformunabhängigkeit besonders gut für XML-Dokumente geeignet, die über das Internet ausgetauscht werden sollen. In Abbildung 9 ist die Baumstruktur der Adressbuchinstanz dargestellt.
Mit Hilfe der DOM-Methoden kann die Darstellung der Informationen (z.B. auf Webseiten) dynamisch verändert werden. So können erfasste Personen zum Beispiel nach verschiedenen Kriterien aufgelistet werden.

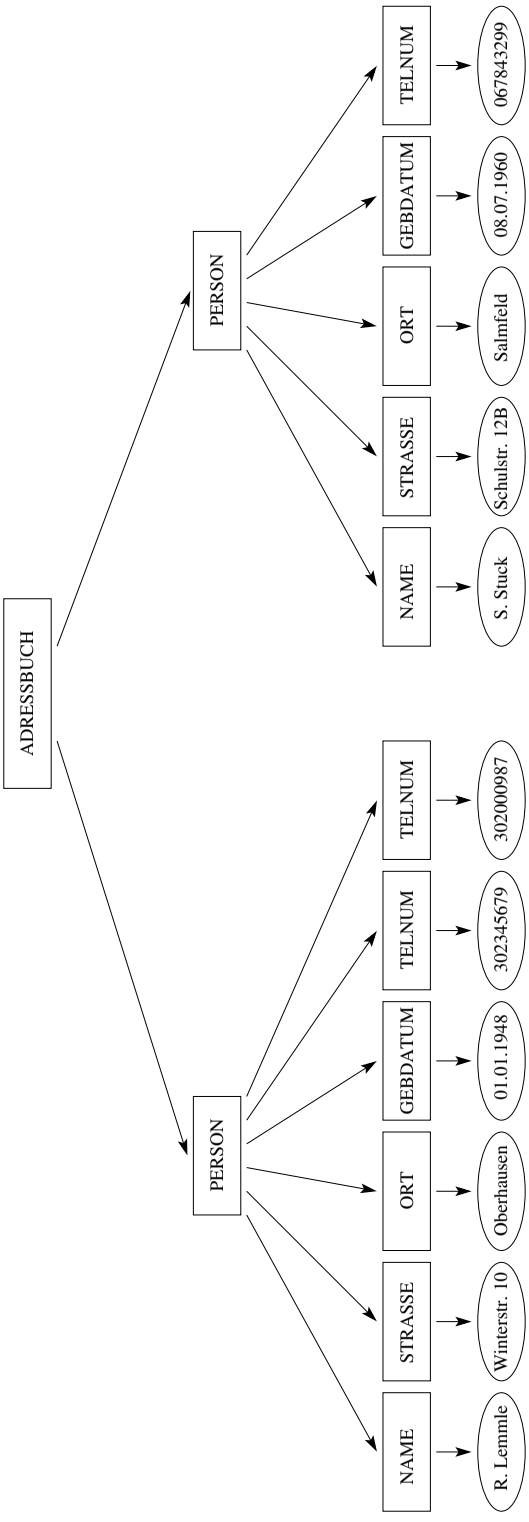

Abbildung 9: Baumstruktur

3 Begriffsbestimmung

Am Anfang dieses Kapitels werden zuerst einige Begriffe definiert werden, die im weiteren Verlauf gelten sollen:

- **Primitiv**: Ein Objekt, das eine der folgenden Formen repräsentiert: Zylinder, Quader, Kugel, Kegel.
- **Komponententeil**: Ein Objekt, das aus Primitiven zusammengesetzt ist, alleine aber keine brauchbare Funktion erfüllt, z.B. die Platine eines Mainboards.
- **Komponente**: Ein Objekt, das aus Komponententeilen aufgebaut ist und selbst wiederum Teil eines Produktes ist, z.B.: ein Mainboard mit einem Sockel und Slots.
- **Produktteil/Teilprodukt**: Ein Teil eines Produktes, das aus einer oder mehreren Komponenten aufgebaut ist, aber noch kein fertiges Produkt ergibt. Dies ist aber vom Kontext abhängig. In manchen Fällen kann ein Mainboard mit Karten schon ein fertiges Produkt darstellen, während es im Kontext eines Komplettsystems nur einen Teil des Produktes darstellt.
- **Produkt**: Die Gesamtheit aller Komponenten, z.B. ein Computer.

Die Idee der Objekterfassung durch GenAu-3D besteht in der Erstellung von domänenspezifischen Autorentools. Innerhalb einer Domäne unterscheiden sich die Produkte oft nur in ihrer Farbe, Textur oder sind teilweise aus den selben Komponenten aufgebaut. Das Meta-Autorentool, das Bestandteil von GenAu-3D ist, wird dazu benutzt, Komponenten oder Komponententeile vorzudefinieren. Daraus wird ein Autorentool erstellt, das diese vordefinierten Komponenten(teile) sowie eine Reihe von Primitiven Objekte beinhaltet. Aus diesen Objekten können dann die einzelnen Produkte kreiert werden. Alle Komponenten(teile) werden eindeutig durch Hersteller und Name identifiziert. Desweiteren können auch noch weitergehende Informationen hinzugefügt werden, die später in der Animation angezeigt werden können. In den beiden Tools, die zur Erstellung der Komponenten dienen, werden diese Komponenten durch Primitive angenähert. Folgende Eigenschaften der Primitiven können verändert werden.

- **Kegel**: Höhe, Radius, Farbe, Textur
- **Quader**: Höhe, Breite, Tiefe, Farbe, Textur
- **Zylinder**: Höhe, Radius, Farbe, Textur
- **Kugel**: Radius, Farbe, Textur

Nachdem eine Komponente erstellt wurde, muss sie noch als Produktteil positioniert werden. Ein Ansatz wäre ihr eine absolute Position zuzuweisen. Dies ist jedoch sehr umständlich und arbeitsaufwendig. Zudem müsste diese Arbeit für jedes Produkt wiederholt werden. Um diese unsinnige Arbeit zu vermeiden, und

Produktkonfigurationen zu vereinfachen, werden die sogenannten *Kontaktpunkte* eingeführt. Kontaktpunkte werden durch ihre Rotation und Position in Relation zur Komponente beschrieben. Wie eine Komponente mittels eines Kontaktpunktes positioniert werden kann, wird in Kapitel 6.2.1 genau beschrieben. Zudem ergeben sich auch andere Vorteile, die in Kapitel 6.2.2 erläutert werden. Kontaktpunkte bilden die Basis, um Komponenten verbinden zu können. Daher können mit ihrer Hilfe auch Animationen erstellt werden. Zwei Komponenten werden miteinander verbunden, wenn sich für jede Komponente je ein Kontaktpunkt findet, der bestimmte Kriterien erfüllt. Diese Kriterien sind: Name, Typ, Status. Kontaktpunkte werden im Autorentool beschrieben. Sie gelten dann für alle Animationen, in denen diese Komponente vorkommt. Zum Beispiel würde eine PCI-Karte in jeden PCI-Slot eines Mainboards passen. Sie können aber auch im Konfigurationstool beschrieben werden. In diesem Fall gelten diese Kontaktpunkte nur für die erstellte Animation, zum Beispiel wenn eine Karte in einen ganz bestimmten Slot eingeschoben werden soll.

4 Das Objektmodell

In Kapitel 2.5 wurden die wichtigsten Konstrukte von XML vorgestellt. Sie ermöglichen eine individuelle Strukturierung von Daten. In der vorliegenden Arbeit sollen Eigenschaften von 3D-Objekten erfasst werden. Hierbei ist ein gut gegliedertes und erweiterbares Objektmodell erwünscht. Das vorliegende Objektmodell wurde mit XML beschrieben und durch eine DTD spezifiziert. Im folgenden werden die Elementtypen des Objektmodells und ihre Bedeutung erklärt. Die Beispiele sind der Computer-Domäne entnommen.

4.1 Konfigurationsfile

```
<!ELEMENT CONFIGURATION-FILE (AUTHORINGTOOL-CONFIGURATION
  |PRODUCT-CONFIGURATION|ANIMATION-CONFIGURATION)>
```

Dieser Elementtyp bildet die Grundlage für jede Konfigurationsdatei, die mit einem der Tools erstellt wird. In einem XML-Dokument bildet er das Wurzelelement.

4.2 Autorentool

Dieser Elementtyp enthält alle vordefinierten Objekte, die in einem Autorentool vorkommen.

```
<!ELEMENT AUTHORINGTOOL-CONFIGURATION  (OBJECTS)+>
<!ATTLIST AUTHORINGTOOL-CONFIGURATION
  AUTHORINGTOOLNAME CDATA #REQUIRED
  OBJECTGROUPS NMTOKENS #REQUIRED
  DISTANCE CDATA #IMPLIED
  UNITOFDISTANCE (cm|m)  "cm">
```

Die Objektgruppen dienen zur Strukturierung des grafischen Benutzerinterfaces, mit dem die 3D-Objekte erfasst werden. Der Name des Autorentools, der hier angegeben wird, ist der Domänennamen. Die Distanz ist die Entfernung, die der Betrachter (Avatar) zum Arbeitsplatz hat. Der Arbeitsplatz ist dabei die Position in der virtuellen Welt, an dem die 3D-Objekte zusammengestellt werden. Zusätzlich zur Entfernung muss noch die entsprechende Einheit angegeben werden.

Beispiel der Attribute eines Autorentoolelementes in einem XML-Dokument.

```
<AUTHORINGTOOL-CONFIGURATION
 AUTHORINGTOOLNAME="Computer"
 OBJECTGROUPS="board card ram processor harddisk"
 DISTANCE="50.0"
 UNITOFDISTANCE="cm">
```

4.3 Objekte

```
<!ELEMENT OBJECTS  (OBJECT)+>
```

Dieser Elementtyp dient lediglich dazu mehrere Objekte zu umschliessen, die wiederum zur Strukturierung der vordefinierten Objekte in den Autorentools dienen.

4.4 Objekt

Der Objekt-Elementtyp enthält zusammengesetzte Objekte.

```
<!ELEMENT OBJECT  (COMPOUNDEDOBJECTS)*>
<!ATTLIST OBJECT
  OBJECTNAME CDATA #REQUIRED
  COMPOUNDEDOBJECTGROUPS CDATA #IMPLIED>
```

Seine Attribute beschreiben den Objektnamen, z.B.: "board" und Gruppen, denen die vordefinierten Objekte angehören. Das wären im Falle eines Mainboards Kartenslots oder Speicherbänke. Warum diese Unterteilung ? Es geht darum, in einem Autorentool möglichst schnell Zugriff auf bestimmte vordefinierte Objekte erlangen zu können. Für ein Mainboard wird zum Beispiel eine Gruppe "slot" angegeben. In ihr befinden sich dann die vordefinierten Slots, die für alle Mainboards gleich sind (ISA-, PCI, AGP-Slot).
Beispiel der Attribute eines Objektelementes:

```
<OBJECT
 OBJECTNAME="board"
 COMPOUNDEDOBJECTGROUPS="slot bank socket">
```

4.5 Zusammengesetzte Objekte

Zusammengesetzte Objekte bestehen aus mehreren primitiven Objekten und bilden die vordefinierten Objekte für ein Autorentool. Eine Komponente besteht aus einer Anzahl vordefinierter und/oder primitiver Objekte.

```
<!ELEMENT COMPOUNDEDOBJECT  (PRIMITIVEOBJECTS)*>
<!ATTLIST COMPOUNDEDOBJECT
  COMPOUNDEDOBJECTNAME CDATA #REQUIRED
  COMPOUNDEDOBJECTGROUP CDATA #IMPLIED
  DESCRIPTIONAUTHOR CDATA #IMPLIED
  DESCRIPTIONDATE CDATA #IMPLIED
  DESCRIPTIONURL CDATA #IMPLIED
  XTRANSLATION CDATA "0.0"
  YTRANSLATION CDATA "0.0"
  ZTRANSLATION CDATA  "0.0"
  XROTATION CDATA  "0.0"
  YROTATION CDATA  "0.0"
  ZROTATION CDATA  "0.0">
```

Ein zusammengesetztes (vordefiniertes) Objekt besitzt einen Namen und ist einer Gruppe zuzuordnen. Im unten geschilderten Beispiel handelt es sich um einen PCI-Slot, der für Mainboards vordefiniert ist. Die Tags `<COMPOUNDEDOBJECT>` und `</COMPOUNDEDOBJECT>` klammern die primitiven Objekte, aus denen dieser Slot aufgebaut ist. Da ein solches Objekt Bestandteil einer Komponente ist, kann es Sinn machen, sie textuell zu beschreiben. Diese Beschreibungen werden in gesonderten Dateien abgespeichert und in dem XML-Dokument referenziert. Zudem können auch Autor und Datum der Beschreibung vermerkt werden. Da ein zusammengesetztes Objekt Bestandteil einer Komponente und damit eines Produktes wird, muss sie auch die entsprechenden Positions- und Rotationsdaten beinhalten, die die Lage im Raum eindeutig beschreibt.

Beispiel der Attribute eines zusammengesetzten Elementes:

```
<COMPOUNDEDOBJECT
 COMPOUNDEDOBJECTNAME="pci"
 COMPOUNDEDOBJECTGROUP="slot"
 DESCRIPTIONAUTHOR="Peter Blanchebarbe"
 DESCRIPTIONDATE="13.12.00"
 DESCRIPTIONURL="Computer\board\Asus\descriptions\co1.dsc"
 XTRANSLATION="-0.028"
 TRANSLATION="-0.09299999"
 ZTRANSLATION="0.0049"
 XROTATION="-1.123"
 YROTATION="0.0"
 ZROTATION="0.287">
```

4.6 Primitive Objekte

Dieser Elementtyp dient dazu, mehrere primitive Objekte zu umschliessen.

```
<!ELEMENT PRIMITIVEOBJECTS  (PRIMITIVEOBJECT)+>
```

4.7 Primitives Objekt

Dieser Elementtyp bietet die Möglichkeit, die kleinste grafische Einheit in einem XML-Dokument zu erfassen. Primitive Objekte können Bestandteil von zusammengesetzten Objekten sein oder auch für sich allein stehen.

```
<!ELEMENT PRIMITIVEOBJECT  EMPTY>
<!ATTLIST PRIMITIVEOBJECT
  DESCRIPTIONAUTHOR CDATA #IMPLIED
  DESCRIPTIONDATE  CDATA #IMPLIED
  DESCRIPTIONURL  CDATA #IMPLIED
  SHAPETYPE  CDATA #REQUIRED
  UNIT (mm|cm|m) "cm"
  X CDATA "0.0"
  Y CDATA "0.0"
  Z CDATA "0.0"
```

```
RADIUS CDATA "0.0"
HEIGHT CDATA "0.0"
FRONTTEXTURE  CDATA #IMPLIED
BACKTEXTURE  CDATA #IMPLIED
TOPTEXTURE  CDATA #IMPLIED
BOTTOMTEXTURE  CDATA #IMPLIED
LEFTTEXTURE  CDATA #IMPLIED
RIGHTTEXTURE  CDATA #IMPLIED
TEXTURE  CDATA #IMPLIED
XTRANSLATION  CDATA "0.0"
YTRANSLATION  CDATA "0.0"
ZTRANSLATION CDATA "0.0"
XROTATION CDATA "0.0"
YROTATION CDATA "0.0"
ZROTATION CDATA "0.0"
RED CDATA "1.0"
GREEN CDATA "1.0"
BLUE CDATA "1.0">
```

Der Vollständigkeit halber können auch diese Objekte textuell beschrieben werden. Ein primitives Objekt kann ein Quader, ein Zylinder, ein Kegel oder eine Kugel verschiedener Grösse, Farbe oder Textur sein. Der Begriff "`SHAPETYPE`" beinhaltet ein Schlüsselwort, das besagt, um welches dieser Primitive es sich handelt. Im virtuellen Raum werden die Längen in Metern gemessen. Da es aber möglich sein muss, auch kleinste Objekte zu beschreiben, z.B. Schrauben die nur wenige Millimeter lang sind, können sie in verschiedenen Grössenordnungen erfasst werden. Anhand dieser Grössenordnung werden die Angaben umgesetzt. In der folgenden Tabelle stehen die Schlüsselworte und die Angaben, durch die die Primitive beschrieben werden.

SHAPETYPE	Beschreibung
Box	X, Y, Z, FRONTTEXTURE, BACKTEXTURE, TOPTEXTURE, BOTTOMTEXTURE, LEFTTEXTURE, RIGHTTEXTURE, RED, GREEN, BLUE
Cylinder	RADIUS, HEIGHT, TEXTURE, RED, GREEN, BLUE
Cone	RADIUS, HEIGHT, TEXTURE, RED, GREEN, BLUE
Sphere	RADIUS, TEXTURE, RED, GREEN, BLUE

Ein Quader (Box) kann auf jeder Seite eine andere Textur besitzen, während die anderen Primitive nur mit einer einzigen Textur versehen werden können. Die Farbgebung wird mittels des RGB-Farbmodells beschrieben. Eine Textur überdeckt die Farbe der texturierten Fläche. Die Umrechnung der Maße im untenstehenden Beispiel in den virtuellen Raum ist: 0.01*21.2, 0.01*30.5, 0.01*0.1 Der Faktor wird durch die Einheit "cm" gegeben.
Zusammengesetzte Objekte und Komponenten bestehen aus Primitiven. Aus diesem Grund muss die Lage der Primitiven im Raum beschrieben werden können. Dies geschieht durch die Translations- und Rotationsattribute.
Beispiel eines primitiven Objektes:

```
<PRIMITIVEOBJECT
 DESCRIPTIONAUTHOR="Peter Blanchebarbe"
 DESCRIPTIONDATE="13.12.00"
 DESCRIPTIONURL="Computer\board\Asus\descriptions\po0.dsc"
 SHAPETYPE="Box"
 UNIT="cm"
 X="21.2"
 Y="30.5"
 Z="0.1"
 FRONTTEXTURE="Computer\board\Asus\textures\sp97xv.jpg"
 BACKTEXTURE="Computer\board\Asus\textures\back.jpg"
 TOPTEXTURE=""
 BOTTOMTEXTURE=""
 LEFTTEXTURE=""
 RIGHTTEXTURE=""
 XTRANSLATION="1.0"
 YTRANSLATION="2.5"
 ZTRANSLATION="0.0"
```

```
XROTATION="1.27856"
YROTATION="0.0"
ZROTATION="0.0"
RED="0.9529412"
GREEN="0.9529412"
BLUE="0.8509804">
```

4.8 Komponentenkonfiguration

Mit Hilfe dieses Elementtyps werden die Komponenten eines Produktes beschrieben, die zur späteren Verwendung bereitstehen sollen. Da es möglich sein soll in einer Animation Zwischenschritte zu machen, also ein Teilprodukt zu erstellen und später wiederzuverwenden, gibt es `STEPS`. Sie werden in Kapitel 4.10 näher erläutert.
Kontaktpunkte (`CONTACTOBJECTS`) sind ein wichtiger Bestandteil einer Komponente. Sie sind für die Anordnung der Komponenten untereinander zuständig und werden in Kapitel 4.12 und 6.2.1 näher beschrieben.
Die Elementtypen `PRIMITIVEOBJECTS` und `COMPOUNDEDOBJECTS` wurden in den Kapiteln 4.7 und 4.5 bereits beschrieben.

```
<!ELEMENT COMPONENT-CONFIGURATION  (STEPS?, CONTACTOBJECTS?,
  PRIMITIVEOBJECTS?,COMPOUNDEDOBJECTS?)>
<!ATTLIST COMPONENT-CONFIGURATION
  PRODUCTNAME  CDATA #REQUIRED
  MANUFACTURER  CDATA #REQUIRED
  AUTHORINGTOOLNAME  CDATA #REQUIRED
  PRODUCTGROUP  CDATA #REQUIRED
  RANK CDATA "0.0"
  NUMBER CDATA "0.0"
  TIME CDATA "1.0"
  INSERTIONTYPE (seriell|parallel) "seriell"
  DESCRIPTIONAUTHOR  CDATA #IMPLIED
  DESCRIPTIONDATE CDATA #IMPLIED
  DESCRIPTIONURL CDATA #IMPLIED>
```

Eine Komponentenkonfiguration enthält zur Identifikation eine Namens- und eine Herstellerbezeichnung. Ausserdem enthält sie den Namen der Domäne, zu der sie gehört(`AUTHORINGTOOLNAME`), sowie die Produktgruppe, in der sie erstellt wurden. Die Attribute `RANK`, `NUMBER`, `TIME`, `INSERTIONTYPE` sind speziell für Animationen. Durch `RANK` wird der Rang angegeben, an dem diese Komponente in der Animation eingefügt werden soll. Das Attribut `NUMBER` enthält die Anzahl an Komponenten, die eingefügt werden soll. `TIME` enthält den Wert in Sekunden, in dem ein oder mehrere Komponenten des gleichen Typs von ihrer Ausgangs- zu ihrer Endposition gelangen. Bei einem höheren Wert kann man der Animation leichter folgen. Wenn zum Beispiel mehrere Schrauben eingefügt werden sollen, so kann mit Hilfe des Attributs `INSERTIONTYPE` festgelegt werden, ob sie hintereinander (seriell) oder gleichzeitig (parallel) eingefügt werden

sollen.
Zusätzlich zu den Beschreibungsmöglichkeiten für zusammengesetzte und primitive Objekte kann für jede Komponente auch noch eine Beschreibung verfasst werden. Diese kann einfach nur Angaben über eine Komponente enthalten oder im Falle einer Bauanleitung genauere Instruktionen, wie diese Komponente eingefügt werden muss.
Beispiel der Attribute einer Komponentenkonfiguration:

```
<COMPONENT-CONFIGURATION
 PRODUCTNAME="P5A"
 MANUFACTURER="Asus"
 AUTHORINGTOOLNAME="Computer"
 PRODUCTGROUP="board"
 RANK="1"
 NUMBER="1"
 TIME="3"
 INSERTIONTYPE="seriell"
 DESCRIPTIONAUTHOR="Peter Blanchebarbe"
 DESCRIPTIONDATE="13.06.00"
 DESCRIPTIONURL="Computer\board\Asus\P5A.dsc">
```

4.9 Zwischenschritte

Dieser Elementtyp ermöglicht es, mehrere Zwischenschritte zu erfassen.
`<!ELEMENT STEPS (STEP)*>`
Zwischenschritte stehen in der XML-Datei, die eine Komponentenkonfiguration beschreibt. In der Animation werden sie nach dem Einfügen dieser Komponente ausgeführt.

4.10 Zwischenschritt

Dieser Elementtyp ermöglicht das Erfassen eines Zwischenschrittes.

```
<!ELEMENT STEP EMPTY>
<!ATTLIST STEP
  FOLLOWINGSTEPTYPE (insertpart|partfinished) "partfinished"
  FOLLOWINGSTEPNAME CDATA #REQUIRED>
```

Es gibt zwei verschiedene Arten von Zwischenschritten:

- `partfinished`: Dieser Ausdruck bedeutet, dass ein Teilprodukt fertiggestellt ist und zur weiteren Verwendung zur Verfügung stehen soll. In diesem Fall wird das Teilprodukt vom Arbeitsplatz genommen, d.h. der root-Knoten der Komponente wird im Szenengraph von seinem Elter gelöst. Der Name setzt sich aus dem Ausdruck `STEP` sowie einer Zahl zusammen.

- `insertpart`: Dieser Ausdruck bedeutet, dass ein Teilprodukt, das durch `partfinished` vom Arbeitsplatz genommen wurde, wieder an das aktuell bearbeitete Teilprodukt angefügt wird. Der Name dieses Typs setzt sich aus dem Ausdruck `INSERT` und einer Zahl zusammen.

Es werden also immer korrespondierende Paare aus `insertpart` und `partfinished` gebildet. Die Zahl in dem jeweiligen Namen besagt, wie diese Zwischenschritte zusammengehören. Die Paare werden normalerweise aus verschiedenen Komponentenkonfigurationen gebildet, wobei jede Komponentenkonfiguration in einer eigenen Datei steht.
Beispiel eines Zwischenschritt-Paares:

```
<STEP
 FOLLOWINGSTEPTYPE="partfinished"
 FOLLOWINGSTEPNAME="STEP 2">

<STEP
 FOLLOWINGSTEPTYPE="insertpart"
 FOLLOWINGSTEPNAME="INSERT 2">
```

4.11 Kontaktpunkte

Dieser Elementtyp dient zur Erfassung mehrerer Kontaktpunkte.

```
<!ELEMENT CONTACTOBJECTS (CONTACTOBJECT)*>
```

4.12 Kontaktpunkt

Dieser Elementtyp erlaubt das Erfassen von Kontaktpunkten. Kontaktpunkte sind ein wichtiger Bestandteil der komponentenbasierten Produktkonfiguration. Mit ihrer Hilfe wird versucht die physikalischen Verbindungen, die in der Realität existieren, in der virtuellen Realität abzubilden. Eine exakte Abbildung ist nicht möglich, daher müssen verschiedene Verbindungen abstrahiert werden. Auch kann auf Grund der Beschaffenheit eines Szenengraph nicht jede Verbindung abgebildet werden. Betrachten wir zum Beispiel eine ringförmige Verbindung von Komponenten. Solche Strukturen können nicht vollständig abgebildet werden, da dies einen Zyklus im Szenengraphen bedeuten würde, von daher muss eine Verbindung aufgebrochen werden.
Durch die Kontaktpunkte wird festgelegt, wie die Komponenten zueinander angeordnet werden. Kontaktpunkte werden sowohl im Autorentool als auch im Konfigurationstool definiert. Eine Komponente kann mehr als einen Kontaktpunkt besitzen, da Komponenten mit verschiedenen anderen Komponenten zusammenhängen können. So muss ein Prozessor zum Beispiel einen Kontaktpunkt für den Sockel und einen Kontaktpunkt für den Lüfter besitzen.
Um eine Verbindung mit Hilfe dieser Kontaktpunkte zu erstellen, muss folgende Bedingung erfüllt sein.

CONTACTNAME(A)== CONTACTNAME(B) &
CONTACTTYPE(A) $\neq$ CONTACTTYPE(B) &
CONTACT(A)==free & CONTACT(B)==free
Der Kontakttyp ist entweder `male` or `female`.

Wie diese Kontaktpunkte erstellt werden können und wie sie im Szenengraph umgesetzt werden, wird in den Kapiteln 5.2 und 6 beschrieben.

```
<!ELEMENT CONTACTOBJECT EMPTY>
<!ATTLIST CONTACTOBJECT
  CONTACTNAME CDATA #REQUIRED
  CONTACTTYPE (male|female) "male"
  UNIT (mm|cm|m) "cm"
  FACTOR CDATA "1.0"
  XTRANSLATION CDATA "0.0"
  YTRANSLATION CDATA "0.0"
  ZTRANSLATION CDATA "0.0"
  XROTATION CDATA "0.0"
  YROTATION CDATA "0.0"
  ZROTATION CDATA "0.0">
```

Wie oben schon erwähnt, besitzt jeder Kontaktpunkt zur Identifikation einen Namen sowie eine Typbezeichnung. Da Kontaktpunkte im Autorentool und Konfigurationstool erfasst werden, besitzen sie auch eine grafische Darstellung. Sie werden in verschiedenen Domänen eingesetzt, die wiederum in verschiedenen Grössenordnungen vorliegen können. Daher müssen auch die Kontaktpunkte in verschiedenen Grössen dargestellt werden. Zu diesem Zweck gibt es die Attribute `UNIT` und `FACTOR`. Ein Kontaktpunkt besitzt ein fest-definiertes Aussehen. Er kann aber durch die Einheiten Millimeter, Zentimeter oder Meter sowie einen Faktor skaliert werden.
Die Translations- und Rotationswerte bestimmen die Position und Ausrichtung des Kontaktpunktes relativ zum Ursprung des Koordinatensystems, in dem die Komponente definiert ist. Die folgenden Kontaktpunkte wurden definiert, um eine Verbindung zwischen einem AGP-Slot auf dem Mainboard und einer Grafikkarte herstellen zu können. Sie sind in der XML-Datei des Mainboards und der der Grafikkarte beschrieben.

```
<CONTACTOBJECT
 CONTACTNAME="agp"
 CONTACTTYPE="female"
 UNIT="mm"
 FACTOR="1.0"
 XTRANSLATION="-0.016"
 YTRANSLATION="-0.14599998"
 ZTRANSLATION="0.0049"
 XROTATION="1.5707964"
 YROTATION="0.0"
 ZROTATION="-1.5707964">
</CONTACTOBJECT>

<CONTACTOBJECT
 CONTACTNAME="agp"
 CONTACTTYPE="male"
 UNIT="mm"
 FACTOR="2.0"
 XTRANSLATION="0.014000001"
```

```
 YTRANSLATION="-0.0405"
 ZTRANSLATION="-0.0040"
 XROTATION="0.0"
 YROTATION="0.0"
 ZROTATION="-1.5707964">
</CONTACTOBJECT>
```

4.13 Animationskonfiguration

Mit Hilfe dieses Elementtyps können noch einige Eigenschaften für die Animation angegeben werden.

```
<!ELEMENT ANIMATION-CONFIGURATION EMPTY>
<!ATTLIST ANIMATION-CONFIGURATION
  ANIMATIONNAME  CDATA #REQUIRED
  DISTANCE CDATA "0.0"
  UNITOFDISTANCE (cm|m) "cm"
  PRODUCTGROUP  CDATA #REQUIRED
  BACKGROUNDIMAGE CDATA #REQUIRED
  DESCRIPTIONAUTHOR  CDATA #IMPLIED
  DESCRIPTIONDATE CDATA #IMPLIED
  DESCRIPTIONURL CDATA #IMPLIED>
```

Der Name der Animation und die zugehörige Produktgruppe wird in der entsprechenden XML-Datei festgehalten. Ausserdem enthält diese die Entfernung, die der Betrachter (Avatar) zur Animation hat. Zur weiteren Ausgestaltung kann noch eine Hintergrundbild angegeben werden. In der Animation stehen die Informationen zu allen Komponenten bereit. Desweiteren kann zusätzlich noch das ganze Produkt beschrieben werden. Hierbei kann es sich um Herstellungsort, Preis oder andere Informationen handeln.

Im folgenden Beispiel ist die Animationskonfiguration für einen Computer angegeben.

```
<ANIMATION-CONFIGURATION
 ENDPRODUCTNAME="computerconfiguration"
 DISTANCE="60"
 UNITOFDISTANCE="cm"
 PRODUCTGROUP="Computer"
 BACKGROUNDIMAGE="background.jpg"
 DESCRIPTIONAUTHOR="Peter B."
 DESCRIPTIONDATE="11.12.00"
 DESCRIPTIONURL="computer\computerconfiguration.dsc">
</ANIMATION-CONFIGURATION>
```

5 Erfassung und Generierung

Wir haben in den letzten Kapiteln gesehen, wie Daten in einem Objektmodell erfasst werden können. Es ist jedoch nicht davon auszugehen, dass jemand dazu bereit ist, komplexere Objekte mittels eines Texteditors zu erfassen. Dies erfordert nicht nur einen immensen Arbeitsaufwand sondern auch eine sehr gute Vorstellungskraft, um zum Beispiel ein Mainboard mit allen Slots korrekt zu erfassen. Ganz zu schweigen von den Informationen, die nötig sind, um eine fertige Animation zu erstellen. Zu diesem Zweck wurde GenAu-3D entwickelt. In Abbildung 10 ist schematisch dargestellt, wie eine Animation erstellt wird. GenAu-3D beinhaltet vier Tools.

- Meta-Autorentool
- Autorentool
- Konfigurationstool
- Animationstool

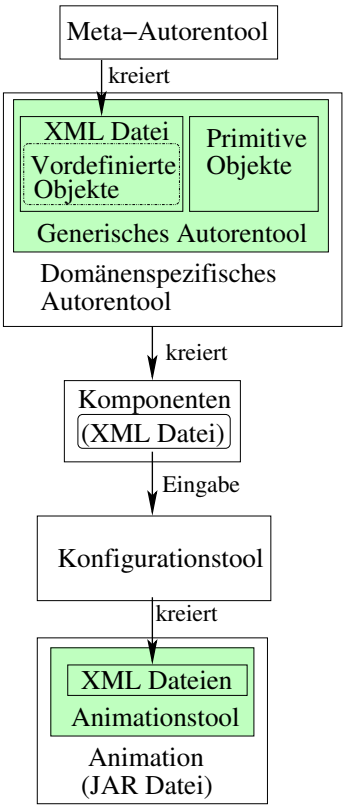

Abbildung 10: Erstellung einer Animation

Alle Anwendungen lassen sich in verschiedene logische Einheiten unterteilen, mit deren Hilfe bestimmte Daten erfasst werden können. Diese Einheiten, die

Abbildung 11: Das Startfenster

auf dem Bildschirm jeweils ein eigenes Fenster besitzen, werden im folgenden beschrieben. Ausserdem gibt es noch ein Fenster, in dem die Objekte mittels Java3D dargestellt werden, das *Canvas*. Es wird bei Bedarf geöffnet und bietet folgende Interaktionsmöglichkeiten:

- Rotation: Mit der linken Maustaste lässt sich die Arbeitsfläche rotieren, d.h. alle dargestellten Objekte.
- Translation: Mit der rechten Maustaste kann die Arbeitsfläche in der x-y-Ebene verschoben werden.
- Zoom: Mit der mittleren Maustaste kann der Arbeitsplatz in der z-Achse auf den Betrachter zu- oder wegbewegt werden.
- Selektion: Die Selektion erlaubt es, je nach Anwendung einzelne Objekte oder ganze Gruppen hervorzuheben. Dabei werden Eigenschaften der Objekte erfasst und in den Anwendungsfenstern angegeben.

5.1 Das Meta-Autorentool

Das Meta-Autorentool dient der Erstellung domänenspezifischer Autorentools. Der Vorteil eines solchen Autorentools liegt darin, dass Komponenten dieser Domäne, die öfters zum Einsatz kommen, nur einmal erfasst werden müssen. Sie können dann zur Erstellung verschiedener Komponenten und Produkte verwendet werden.

5.1.1 Das Startfenster

Im Startfenster kann der Benutzer ein neues Autorentool für eine bestimmte Domäne erstellen (**Create New Authoring Tool**). Falls schon vorher Autorentools kreiert wurden, können diese geändert werden (**Change**).

5.1.2 Die Grundeinstellungen

In diesem Fenster werden die Voraussetzungen für das Autorentool geschaffen. Hier wird der Name der Domäne festgelegt, in der später die Komponenten

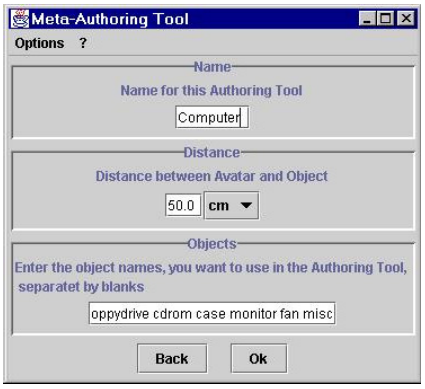

Abbildung 12: Die Grundeinstellungen

erstellt werden sollen. Desweiteren kann die Entfernung, die der Betrachter (Avatar) zu seiner Arbeitsfläche haben soll, festgelegt werden. Als Maße stehen Zentimeter und Meter zur Verfügung. Um später eine Gliederung der Komponenten zu erhalten, können im unteren Textfeld der Abbildung 11 Gruppenbezeichnungen angegeben werden.

5.1.3 Das Objektfenster

In diesem Fenster (siehe Abbildung 13) werden verschiedene Panels mit den Gruppenbezeichnungen dargestellt. In jedem dieser Panels können weitere Unterteilungen gemacht werden. Diese Unterteilungen dienen später im Autorentool dazu, dass die vordefinierten Objekte gegliedert zur Verfügung stehen. Zum Beispiel gibt es in der Computerdomäne verschiedene Slot- oder verschiedene Sockeltypen. Nach diesen Typen können die vordefinierten Objekte gegliedert werden. Wurden bereits Objekte vordefiniert, so können sie mit Hilfe einer Liste und dem **Change**-Knopf überarbeitet und mit dem **Delete**-Knopf gelöscht werden. Mit dem **Store Data**-Knopf werden alle bisher erfassten Daten in den Panels in einer XML-Datei gespeichert. Diese XML-Datei bildet zusammen mit einem allgemeinen Autorentool das domänenspezifische Autorentool. Bisher wurden nur Daten erfasst, die der Gliederung und Arbeitsplatzeinstellung dienen. Mit dem **Predefine New Object**-Knopf gelangt man in das Fenster, in dem die Objekte vordefiniert werden können, d.h. hier werden die Objekte grafisch erfasst.

5.1.4 Erstellen vordefinierter Objekte

Die Werte der vordefinierten Objekte werden im Fenster, das in Abbildung 14 zu sehen ist, erfasst. Die Visualisierung der Objekte geschieht im *Canvas*. In Abbildung 15 ist ein vordefinierter Sockel dargestellt. Zuerst muss das Objekt, das erfasst werden soll, eine Identifikation erhalten. Dies geschieht durch die Eingabe einer Bezeichnung und die Auswahl einer Gruppe, die in dem Objektfenster eingegeben wurde. Anhand dieser Identifikation kann ein Objekt später im Autorentool ausgewählt und verwendet werden. In dem **Options**-Dialog

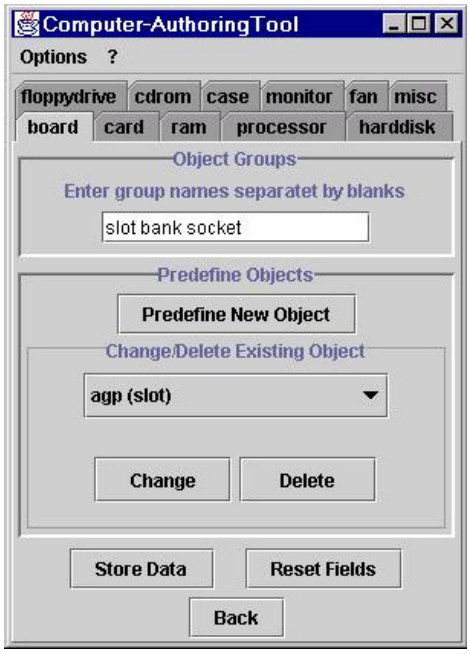

Abbildung 13: Das Objektfenster

lassen sich die Beleuchtung für das im Canvas dargestellte Objekt ein- und ausschalten, sowie die Hintergrundfarbe des Canvas ändern.
Wie werden nun die Objekte erfasst? Ein Objekt besteht aus mehreren Primitiven. Exemplarisch wird hier nur erläutert wie ein Quader erzeugt und verändert werden kann. Die anderen Primitive werden analog bearbeitet. Durch den **New Box**-Knopf wird ein weisser Quader mit Kantenlänge 1 erzeugt. Die Kantenlängen des Quaders können in den **X**-, **Y**- und **Z**-Feldern verändert werden. Die Einheit dieser Werte wird durch den Auswahl-Knopf bestimmt. Sie werden in dem Canvas entsprechend umgesetzt, d.h., da die Einheit in Java3D Meter ist, müssen alle Grössenwerte mit einem Faktor multipliziert werden, der sich aus der Wahl der Einheit ergibt. Der Faktor beträgt 0.001 für Millimeter, 0.01 für Zentimeter und 1 für Meter. Danach kann der Quader gefärbt werden. Durch den **Color**-Knopf lässt sich ein Fenster öffnen, das die Möglichkeit bietet, beliebige Farben auszuwählen. Die Farben können mittels des RGB-Farbmodells bestimmt werden. Die Oberfläche eines Primitivs kann zusätzlich zur Farbwahl durch eine Textur bestimmt werden, wobei die Textur die Farbe überlagert. Für einen Quader lassen sich für jede Seite eine Textur wählen. So öffnet sich durch den **Front**-Knopf ein Fenster, in dem durch die lokale Verzeichnisstruktur gebrowst werden kann, um eine Grafik für die Vorderseite des Quaders zu finden. Die anderen Primitive können durch eine Grafik texturiert werden. Bei einer Selektion werden die Daten des primitive Objektes erfasst und in dem Fenster angegeben. Durch den **Remove Picked Object**-Knopf wird ein selektiertes Objekt gelöscht. Da es nicht immer ganz einfach ist ein ganz bestimmtes Objekt mit der Maus zu selektieren, kann man mit Hilfe des **Previous**- und **Next**-Knopfes die Primitive nacheinander anwählen, bis das

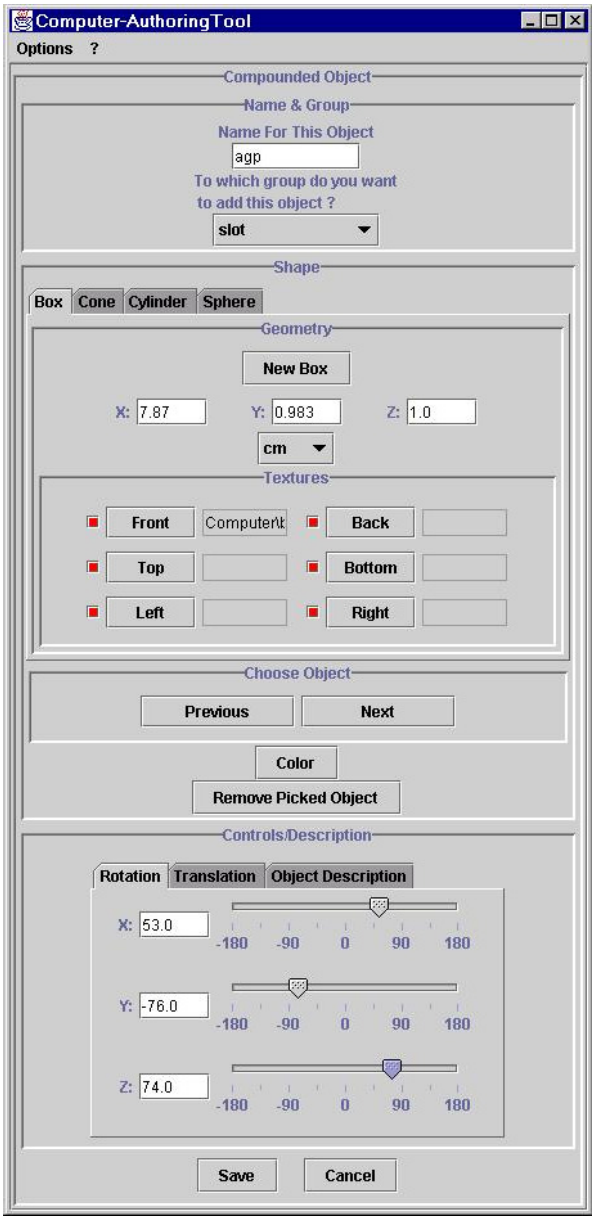

Abbildung 14: Fenster zum Vordefinieren von Objekten

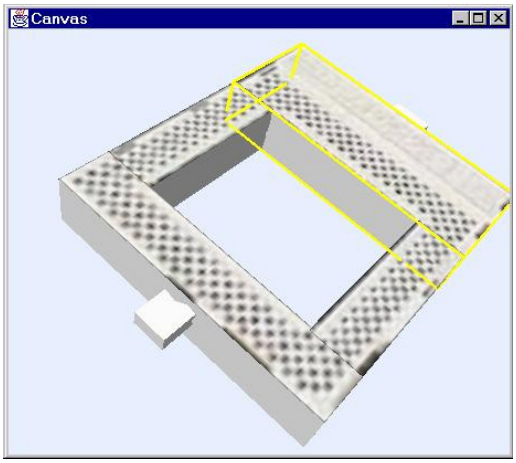

Abbildung 15: Canvas mit Prozessorsockel und einem selektierten texturierten Quader

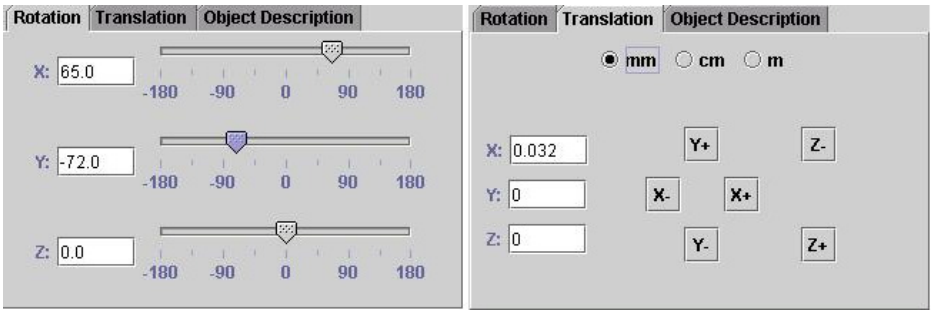

Abbildung 16: Panels zum Rotieren und Positionieren von Primitiven

Gewünschte selektiert ist.
Ein vordefiniertes Objekt kann aus mehreren Primitiven bestehen. Daher muss es möglich sein, die Primitive an eine exakte Position zu bewegen und zu rotieren. Es gibt zwar die Möglichkeit, Objekte mit Hilfe der Maus direkt im Canvas zu plazieren oder zu rotieren, jedoch ist dies zu ungenau und daher für unsere Zwecke nicht geeignet. Aus diesem Grund gibt es die in Abbildung 16 dargestellten Panels. Bei der Selektion eines Primitiv werden die Rotations- und Translationsdaten erfasst und in den Textfeldern angezeigt. Rotationen des selektierten Objektes können entweder ausgeführt werden, indem man hinter dem Achse-Kennbuchstaben einen Wert für den Winkel zwischen -180 und +180 eingibt oder den nebenstehenden Schieberegler benutzt. Die Rotation wird nicht im globalen Weltkoordinatensystem durchgeführt, sondern in dem lokalen Koordinatensystem des primitiven Objektes (siehe Kapitel 2.2).
Translationen können entweder durch die Eingabe eines Wertes oder durch die entsprechenden Richtungsknöpfe ausgeführt werden. Die Auswahlknöpfe für die Einheiten geben an, welcher Faktor für die Translation verwendet wird. Folgendes ist zu beachten: wird zuerst eine Rotation ausgeführt, so ändern sich

auch für den Betrachter die Richtung der Translation.
Der Karteireiter **Description** enthält 3 Textfelder, in denen der Autor das Datum sowie die Beschreibung für dieses vordefinierte Objekt angeben kann.

5.2 Das Autorentool

Im letzten Kapitel wurde gezeigt, wie Objekte vordefiniert werden können, um später wiederverwendet zu werden. Mit Hilfe dieser vordefinierten Objekte und des generischen Autorentools wird ein domänenspezifisches Autorentool erstellt. Dieses Autorentool erlaubt es, Produktkomponenten zu erstellen und zu verwalten.
Ausserdem können hier auch die Kontaktpunkte für die Komponenten definiert werden. Diese Kontaktpunkte machen es erst möglich, dass aus mehreren Komponenten ein Produkt entsteht.
Im folgenden wird beschrieben, wie diese Komponenten erstellt werden.
Um in das Fenster zur Konfiguration zu gelangen, wählt man in dem dropdown-Menü in Abbildung 11 (Startfenster) eine Domäne aus und bestätigt mit dem Knopf **Use**. Darauf öffnen sich drei Fenster. Das Canvas, und zwei Fenster mit den Bedienelementen.

5.2.1 Erstellen von Produktkomponenten

Das Hauptfenster ist durch Karteireiter in Gruppen untergliedert, die im Grundeinstellungsfenster eingegeben wurden. Um eine Komponente zu erstellen, wird eine Identifikation benötigt. Zu diesem Zweck werden der Hersteller und der Name der Komponente eingetragen (siehe Abbildung 17).
In dem Teil "Predefined Objects" können die Objekte ausgewählt und in das Canvas eingefügt werden. Eine Komponente wird aus vordefinierten Objekten und/oder primitiven Objekten zusammengestellt. Zum Beispiel besteht ein Mainboard aus Slots, Speicherbänken und einem Sockel. Diese Objekte wurden schon vordefiniert, da sie, unabhängig auf welchem Board, immer die gleiche Grösse und das gleiche Aussehen haben. Lediglich die Anzahl und die Anordnung auf der Platine kann sich unterscheiden. Daher wird die Platine durch ein texturiertes Primitiv erzeugt und die vordefinierten Objekte darauf plaziert (Abbildung 18). Die Primitive werden wie in Kapitel 5.1 erzeugt und bearbeitet.
Zur Verwaltung der schon kreierten Komponenten können, nach Hersteller geordnet, die Komponenten ausgewählt und im Canvas angezeigt werden. Sie werden mit all ihren Informationen geladen und können weiterverarbeitet werden.
Für den Fall, dass eine Komponente schon erstellt wurde und eines der vordefinierten Objekte im nachhinein im Meta-Autorentool wieder verändert wurde, kann mit dem **Adjust all products of the type <Gruppenname>**-Knopf diese Änderung in schon bestehende Komponenten übernommen werden.
Ein anderes Fenster enthält die Bedienelemente zur Rotation und Positionierung einzelner selektierter Objekte. Dieser Vorgang wurde in Kapitel 5.1 be-

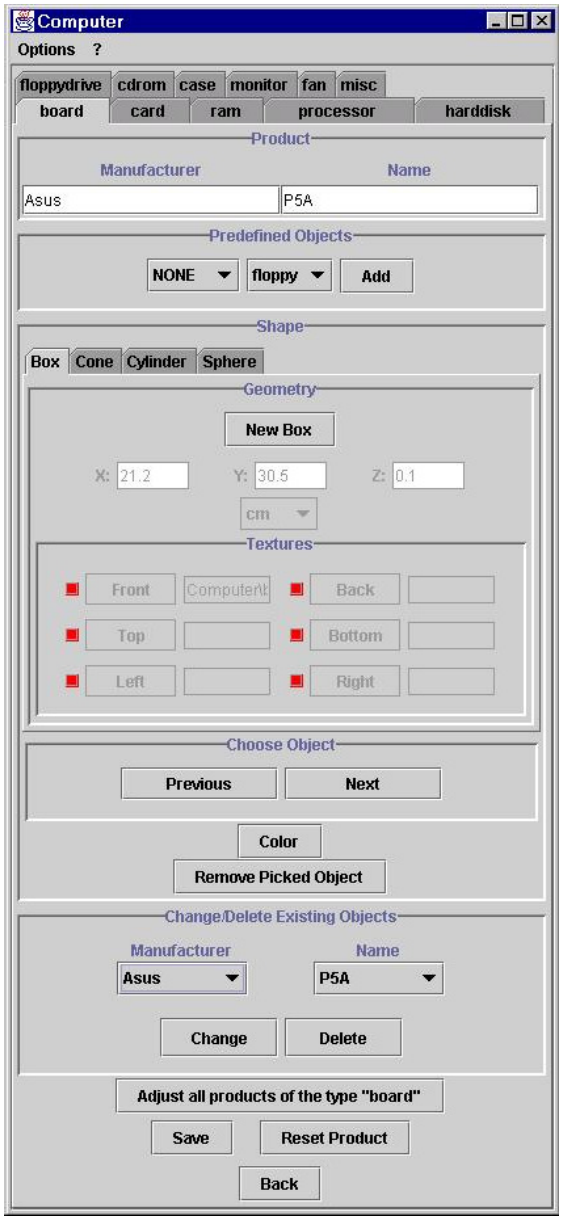

Abbildung 17: Fenster zum Erstellen von Produktkomponenten

Abbildung 18: Aus vordefinierten und primitiven Objekten erstelltes Mainboard

schrieben. Jedes Objekt kann mit einer Beschreibung, dem Namen des Autors und dem Datum versehen werden. Zusätzlich zu dieser Beschreibung kann auch die ganze Komponente auf die selbe Art und Weise beschrieben werden. Diese Beschreibungen können später in der Animation abgerufen werden.
Mittels des **Save**-Knopfes werden alle Daten in dem Objektmodell aus Kapitel 4 erfasst und in einer Datei abgespeichert.

5.2.2 Einfügen von Kontaktpunkten

In dem Dialog "Options" können Kontaktpunkte eingefügt werden. Dazu wird das Fenster aus Abbildung 19 geöffnet. Die Kontaktpunkte definieren Positions- und Rotationsdaten, mit deren Hilfe Komponenten miteinander verbunden werden können. Das heisst, die Komponenten können einerseits optisch zusammengefügt werden, andererseits werden sie auch im Szenengraph miteinander verbunden. Eine genauere Beschreibung über die Szenengraphrepräsentation gibt es in Kapitel 6. Da die Kontaktpunkte positioniert werden müssen, gibt es eine grafische Darstellung (siehe Abbildung 20). Es gibt zwei Typen von Kontaktpunkten:

- Male
- Female

Um die Positionierung der Kontaktpunkte zu vereinfachen, sind alle Objekte transparent. Wird ein Objekt selektiert, wird der dann eingefügte Kontaktpunkt im Mittelpunkt dieses Objektes positioniert. Sonst wird der Kontaktpunkt im Mittelpunkt des Weltkoordinatensystems erzeugt. Die Positionierung und die Ausrichtung eines Kontaktpunktes wird durch die selben Bedienelemente durchgeführt, wie dies auch bei den anderen Objekten geschieht. Zur Identifizierung eines Kontaktpunktes erhält dieser eine Bezeichnung und einen

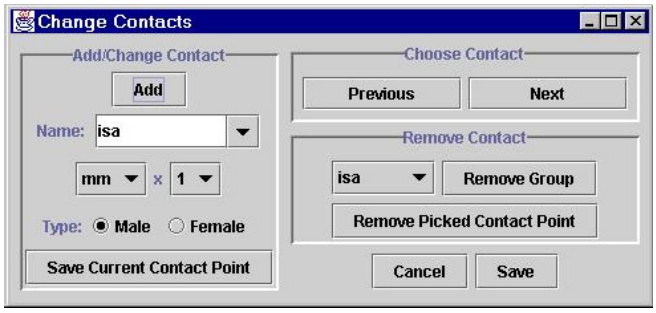

Abbildung 19: Fenster zur Erstellung von Kontaktpunkten

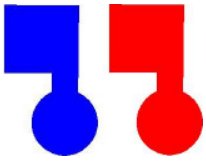

Abbildung 20: Grafische Darstellung der Kontaktpunkttypen "Male" und "Female"

Typ. Mit Hilfe dieser Angaben werden Komponenten später verbunden. Ein Kontaktpunkt kann in seiner Form nur insofern verändert werden, als dass er durch einen vorgegebenen Faktor und eine Einheit skaliert werden kann. Dadurch lässt sich die Grösse eines Kontaktpunktes der Grösse der Komponente, für die er definiert wird, anpassen.
Es lassen sich sowohl einzelne Kontaktpunkte als auch ganze Gruppen von Kontaktpunkten löschen. In Abbildung 21 ist ein Mainboard mit allen Kontaktpunkten dargestellt. Die Komponenten (Karte, Prozessor, Speicher) besitzen Kontaktpunkte vom Typ *male*.
Soll eine Komponente eingefügt werden, so wird aus allen aktuell in der Szene vorhandenen Kontaktpunkten der erste frei verfügbare Kontaktpunkt mit dem gleichen Namen und Typ *female* benutzt um die Komponente auf dem Mainboard zu plazieren.
Animationsspezifische Kontaktpunkte können im Konfigurationstool hinzugefügt oder gelöscht werden.

5.3 Das Konfigurationstool

Das Konfigurationstool, und die daraus resultierende Animation, entstand, um zu zeigen wie Komponenten weiter verarbeitet werden können. Insbesondere wird durch die Animation der Nutzen der Kontaktpunkte ersichtlich. Sie erlauben es durch wenige Handgriffe ein Produkt zusammenzustellen. Den Komponenten muss also nicht eine feste Position zugeordnet werden, stattdessen wird zur Laufzeit entschieden, wie eine Komponente positioniert wird.
Im Startfenster des Konfigurationstool kann eine vorher angelegte Domäne aus-

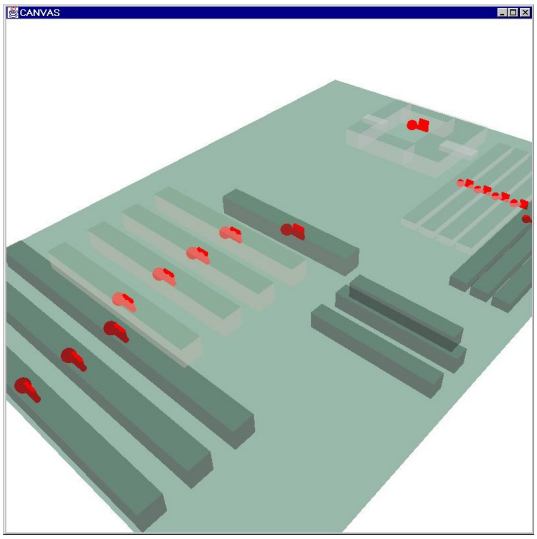

Abbildung 21: Darstellung eines Mainboards mit seinen Kontaktpunkten

gewählt werden, für die eine Animation erstellt werden soll. Wird eine Konfiguration erstellt (Abbildung 22), so wird ihr erst eine Bezeichnung gegeben. Diese Bezeichnung dient später zur Erstellung der Animationsdatei. Desweiteren muss die Entfernung, die der Betrachter zu der Animation haben soll, festgelegt werden. Mit dem **Background Image**-Knopf kann ein Bild ausgewählt werden das während der Animation den Hintergrund füllt. Um nun eine Animation zu erstellen, wird in der linken Liste ein Eintrag ausgewählt und mit dem **Add**-Knopf in die rechte Liste übernommen. Die rechte Liste legt die Reihenfolge fest, in der die Komponenten später in der Animation eingefügt werden. Jeder Eintrag auf der rechten Seite enthält vier zusätzliche Felder:

1. Rang: Der Rang bezeichnet den Punkt, an dem diese Komponente in der Animation eingefügt wird.

2. Anzahl: Diese Zahl gibt an, wie häufig eine Komponente dieses Typs eingefügt werden soll.

3. Zeit: Dieser Eintrag gibt an, wie lange die Animation einer einzelnen Komponente dauern soll. Also die Zeit von ihrem Ausgangs- zum Endpunkt.

4. Typ: Wenn die Anzahl der Komponenten, die in Punkt 2 angegeben wurde, grösser als 1 ist, gibt es zwei Möglichkeiten Komponenten einzufügen:

   - seriell: Die Komponenten werden nacheinander eingefügt.
   - parallel: Die Komponenten werden zum gleichen Zeitpunkt eingefügt.

Diese Einträge werden verändert, indem ein Eintrag selektiert wird und die gewünschten Änderungen in den Feldern unterhalb der Liste ausgefüllt werden. Mit dem **Set**-Knopf werden die Änderungen in den Eintrag übernommen.

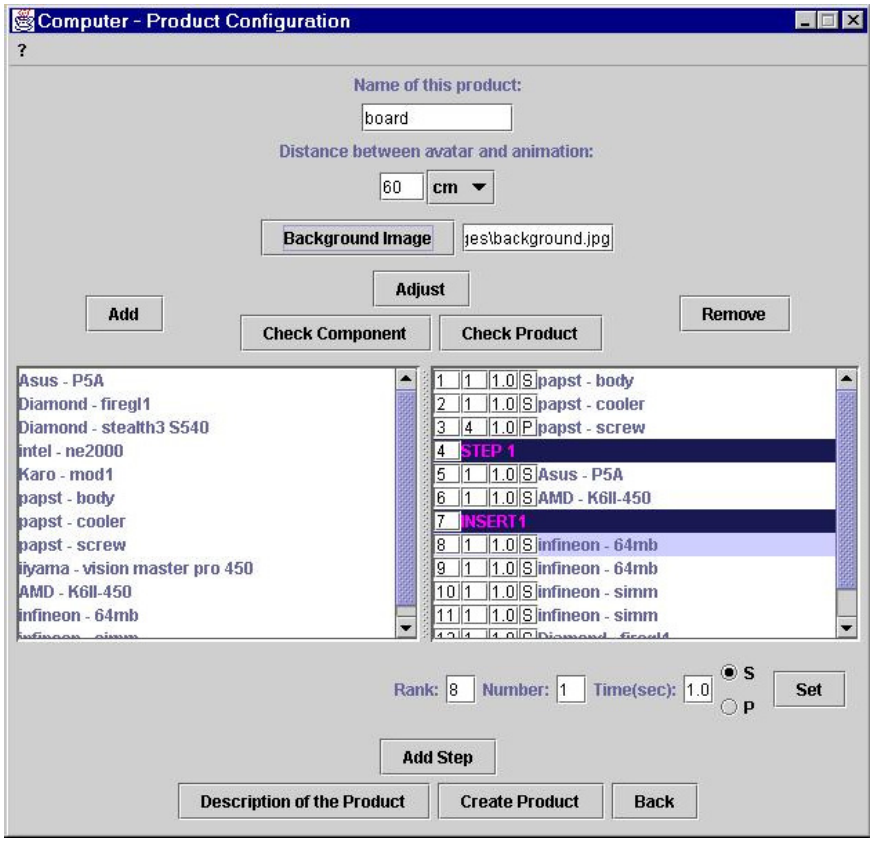

Abbildung 22: Konfigurationstool

Falls eine Komponente im nachhinein verändert wurde, kann mit dem **Adjust**-Knopf diese Änderungen in die rechte Liste übernommen werden.
Will man noch die Korrektheit einer Komponente überprüfen, geschieht dies mit dem **Check Component**-Knopf. Daraufhin öffnet sich ein Canvas, in dem die Komponente angezeigt wird und ein Fenster mit Bedienelementen, um Kontaktpunkte hinzuzufügen, zu ändern oder zu entfernen. Eine Beschreibung dieser Elemente ist in Kapitel 5.2 zu finden. Ändert man mittels dieser Funktion die Kontaktpunkte einer Komponente auf der linken Seite, so wirkt sich die Änderung auf die Komponente des Autorentools aus. Wird eine Änderung der Kontaktpunkte bei einer Komponente auf der rechten Seite vorgenommen, so gelten diese Kontaktpunkte nur in der hier erstellten Animation.
Produkte werden oft nicht an einem einzigen Stück erstellt. Stattdessen werden zuerst Teilprodukte erstellt, die dann zusammengefügt werden. Um solche Fälle in der Animation zu erfassen, können mit dem **Add Step**-Knopf Zwischenschritte generiert werden.
Wird ein Zwischenschritt erzeugt, werden immer zwei Einträge angelegt. Diese Einträge haben nur ein zusätzliches Feld in der Liste, das Rang-Feld. Dieser Eintrag bezeichnet den Rang, an dem das vorliegende Teilprodukt vom Arbeitsplatz genommen werden soll. Ein solcher Eintrag hat die Bezeichnung "STEP <Zahl>". Die Zahl gibt dabei an, der wievielte Zwischenschritt hier gemacht wird. Der mit diesem Eintrag korrespondierende Eintrag ist der, der bestimmt, wann dieses Teilprodukt wieder eingefügt wird. Dieser Eintrag hat die Form "INSERT <Zahl>". Die Zahl besagt, zu welchem *Step* dieser *Insert* gehört.
Mit dem **Check Product**-Knopf kann überprüft werden, ob die gewählten Komponenten mit ihren Kontaktpunkten das gewünschte Ergebnis liefern.
Mit dem **Description of the Product**-Knopf kann eine Beschreibung für das gesamte Produkt zusätzlich zur Beschreibung der einzelnen Komponenten gemacht werden.
Eine Animation wird durch den **Create Product**-Knopf erstellt. Daraufhin wird ein jar-File erzeugt, das alle erforderlichen XML-Dateien, Grafiken und Java-Klassen enthält.

5.4 Das Animationstool

Eine Animation wird durch folgenden Befehl gestartet:

```
java -jar <ANIMATIONSFILE>
```

Daraufhin öffnet sich ein Canvas, in dem die Animation grafisch dargestellt wird und ein Fenster mit Bedienelementen. In Abbildung 23 wird eine Animation dargestellt, in der ein Mainboard und seine Komponenten zusammengefügt werden. In dem oberen Teil des Canvas befindet sich der Teil, in dem die Komponenten abgebildet sind, die noch eingefügt werden müssen. Die linkeste Komponente wird als nächste eingefügt und die anderen rücken um eine Position auf. Eine Einfügeoperation ist eine Animation, in der die Komponente von ihrem Startpunkt zu ihrem Zielort bewegt wird. Im unteren Teil des Canvas werden die Produktteile abgebildet die durch einen Zwischenschritt vom Arbeitsplatz genommen wurden. In diesem Fall ist es ein Lüfter, der zuerst zusammenge-

Abbildung 23: Animation

setzt wurde. Der mittlere Teil des Canvas bildet den Arbeitsplatz, an dem die Komponenten zusammengefügt werden. Wie auch beim Meta-Autorentool und Autorentool kann der Benutzer mit der Maus die Arbeitsfläche drehen, bewegen, heran- und herauszoomen. Weiterhin können die einzelnen Komponenten mit der Maus selektiert werden, um eventuell vorhandene Informationen einzusehen. Während der gesamten Animation werden automatisch zu jeder Komponente vorhandene Informationen angezeigt. In Abbildung 24 befindet sich das zum Canvas zugehörige Benutzerinterface. Der obere Teil ist eine Liste, in der sich der Reihe nach alle Komponenten befinden, die eingefügt werden sollen. Falls mehrere Komponenten auf einem Rang stehen und sie nacheinander eingefügt werden, wird die verbleibende Anzahl in diesem Eintrag angezeigt.
Mit der Checkbox "Light" wird die Beleuchtung im Canvas ein oder ausgeschaltet.
Mit den Checkboxen "Product description" und "Component description" lassen sich Textfenster öffnen und schliessen, in denen vorhandene Informationen angezeigt werden.
Mit dem **Front View**-Knopf wird der Arbeitsplatz in seine Anfangsposition gebracht. Ist keine Hintergrundgrafik vorhanden kann mit dem **Background color**-Knopf eine Farbe für den Hintergrund gewählt werden.
Darunter befinden sich die Steuerelemente. Mit ihnen kann gewählt werden ob die Animation vor- oder rückwärts abgespielt werden soll. Die oberen beiden Knöpfe spielen die Animation bis zum Ende durchgehend ab. Die Animation kann zu jedem Zeitpunkt gestoppt werden. Mit den unteren beiden Knöpfen kann der Benutzer die Animation in einzelnen Schritten vor- oder rückwärts

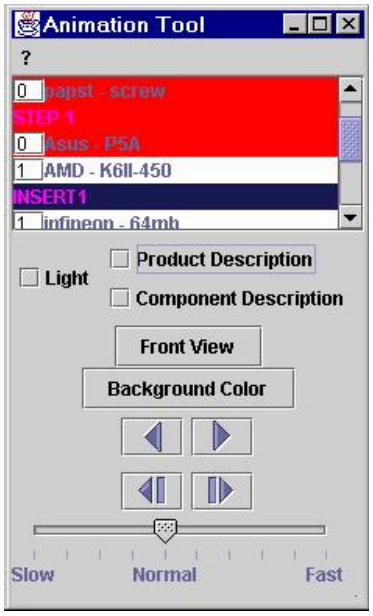

Abbildung 24: Benutzerinterface für eine Animation

durchgehen.
Im unteren Teil befindet sich noch ein Schieberegler der die Animationsgeschwindigkeit steuert.

6 Die Szenengraphverwaltung

Wir haben bis jetzt das XML-Objektmodell betrachtet, in dem die Daten der 3D-Objekte erfasst werden können und die Tools, die dem Benutzer eine grafische Oberfläche bieten, um diese Daten aufzunehmen. Es wurden verschiedene Einheiten vorgestellt, mit deren Hilfe Produkte erstellt werden können.
Für die Szenengraphverwaltung muss festgestellt werden, welche Anforderungen sich aus den Anwendungen ergeben, und wie diese in der Szenengraphverwaltung umgesetzt werden können. Folgende Anforderungen werden an den Szenengraph gestellt:

- **Objekterfassung**: Komponenten und andere Objekte müssen im Szenengraph repräsentiert werden.
- **Zusammenfügen von Komponenten**: Komponenten sollen zur Laufzeit zusammengefügt werden. Daraus ergibt sich die Frage: Wie können Komponenten im Szenengraph mittels ihrer Kontaktpunkte miteinander verbunden werden?
- **Abbildung realer Zusammenhänge**: Die Komponenten sollen im Szenengraph so verbunden sein, wie sie auch in der Realität verbunden sind. Zur Erinnerung: Bei einem Szenengraph handelt es sich um einen gerichteten azyklischen Graph, d.h. alle Transformationen, die auf einem TransformGroup-Knoten ausgeführt werden, beeinflussen damit die Darstellung der visuellen Objekte, die im Szenengraph nach ihm kommen.
- **Konsistenz**: Die Konsistenz des Szenengraph muss unter der Komponenten-Vertauschoperation (Kapitel 6.2.2) erhalten werden. Das heisst, weder die Verbindungen zwischen den einzelnen Komponenten, noch deren Darstellung für den Betrachter dürfen verändert werden.

Darüber hinaus muss es möglich sein, folgende Operationen auf den im Szenengraph repräsentierten Einheiten (Kapitel 6.1) auszuführen:

- Transformation
- Hinzufügen und Entfernen zur Laufzeit
- Eigenschaften der Objekte verändern. Zu diesen Eigenschaften zählen Farbe, Transparenz, Ausmaße etc..

In diesem Kapitel wird gezeigt, wie Objekte in einem Szenengraph repräsentiert werden können. Zugleich wird eine möglichst hohe Flexibilität erreicht. In diesem Fall bedeutet dies, dass Komponenten, die als Einheiten im Szenengraph repräsentiert werden, im Szenengraph verschoben werden können, ohne dass sich diese Manipulation auf der Darstellungsebene auswirkt.

6.1 Verwaltungseinheiten

Es werden folgende Einheiten im Szenengraph betrachtet:

- Primitives Objekt: Ein primitives Objekt bildet die kleinste Verwaltungseinheit im Szenengraph.
- Zusammengesetztes Objekt: Ein zusammengesetztes Objekt besteht aus mehrerer Primitiven.
- Komponente: Ein Komponente besteht aus primitiven und/oder zusammengesetzten Objekten, sowie den Kontaktpunkten.

Die grafischen Darstellungen wurden in Kapitel 3 geschildert. Alle Einheiten bilden Teilgraphen und besitzen bestimmte Eigenschaften und Manipulationsmöglichkeiten. Die wichtigste dieser Einheiten ist die Komponente, die nicht nur die anderen Einheiten beinhaltet, sondern auch die Kontaktpunkte, durch die die Verbindungen der Komponenten realisiert werden. Ist eine Komponente A mit einer Komponente B in der Realität verbunden, so wird diese Verbindung auch im Szenengraph widergespiegelt. Da der Szenengraph azyklisch sein muss, können im Fall der Darstellung eines Zyklus nicht alle Verbindungen im Szenengraph abgebildet werden. Die verbleibenden Verbindungen reichen jedoch zur Darstellung aus.
Im folgenden Kapitel steht $<Name>BG$ für ein BranchGroup-Knoten und $<Name>TG$ für einenTransformGroup-Knoten. Diese und andere Knotentypen des Szenengraphs sind in Kapitel 2.3 näher beschrieben.

6.1.1 Primitive Objekte

Primitive Objekte stellen sich dem Benutzer als Quader, Kegel, Kugeln oder Zylinder dar. Sie besitzen bestimmte Eigenschaften, die geändert werden können. Abbildung 25 zeigt den Aufbau eines primitiven Objektes im Szenengraph und die veränderlichen Eigenschaften einzelner Bestandteile. Zur Verwaltung eines solchen primitiven Objektes gibt es ein API, das Methoden bietet die Eigenschaften zu manipulieren. Jedes primitive Objekt kann jederzeit in den Szenengraph eingefügt oder von ihm entfernt werden. Daher ist der erste Knoten ein BranchGroup-Knoten, da nur ein BranchGroup-Knoten zur Laufzeit eingefügt oder entfernt werden kann.
Der darunterstehende TransformGroup-Knoten dient zur Positionierung und Rotation des primitiven Objektes.
Der *ObjectTG*-Knoten leitet das grafische Objekt ein. Der *BG*-Knoten und der *3D-Object*-Knoten bilden das eigentliche Primitiv. Das *3D-Object* besteht zwar wieder aus einem Teilgraph, dieser ist hier jedoch nicht weiter von Interesse.
Um ein Primitiv zu selektieren, gibt es in diesem Teilgraphen ausserdem noch den *SelBoxBG*- und den *Shape3D*-Knoten. Letzterer bildet ein Drahtgitter um das nebenstehende Primitiv und kann bei Bedarf angezeigt werden, d.h. zur Laufzeit wird der *SelBoxBG*-Knoten von dem Teilgraph entfernt oder hinzugefügt.
Das primitive Objekt hat ausserdem noch folgende Eigenschaften:

- Die Farbe des Primitiv kann geändert werden.
- Die räumliche Ausdehnung kann je nach Form des Primitiv geändert werden.

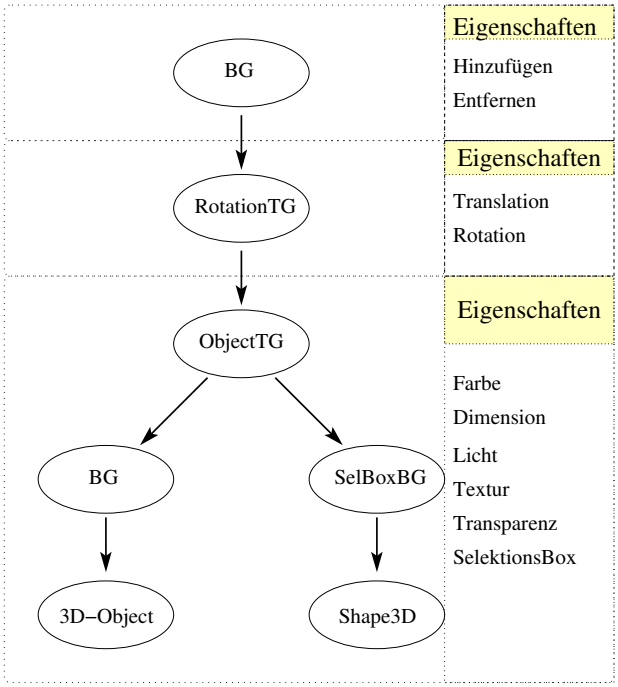

Abbildung 25: Repräsentation eines primitiven Objektes im Szenengraph

- Jedes Primitiv kann mit einer oder im Fall eines Quaders mit mehreren Texturen versehen werden. Zusätzlich können diese Texturen ein- oder ausgeschaltet werden.
- Das Primitiv kann ausserdem transparent gemacht werden. Dies und das Ausschalten der Texturen ermöglicht eine genaue Positionierung der Kontaktpunkte.

6.1.2 Zusammengesetzte Objekte

Ein zusammengesetztes Objekt ist aus mehreren primitiven Objekten aufgebaut. Das grafische Pendant sind die vordefinierten Objekte, deren Erfassung in Kapitel 5.1.4 beschrieben ist.
Ebenso wie bei den primitiven Objekten beginnt der Teilgraph mit einem BranchGroup-Knoten, um dieses Objekt zur Laufzeit zu entfernen oder hinzuzufügen, sowie mit einem TransformGroup-Knoten, um Position und Rotation bestimmen zu können. Die grafische Einheit wird durch den *ObjectTG*-Knoten eingeleitet. Dessen Kinder sind die für dieses zusammengesetzte Objekt bestimmten primitiven Objekte, deren Szenengraphdarstellung in Kapitel 6.1.1 schon beschrieben wurde. Die Eigenschaften Light, Texture, Transparency und SelectionBox werden einfach an die primitiven Objekten weitergeleitet. Das bedeutet, dass die Java-Klasse, die ein solches zusammengesetztes Objekt implementiert eine Verwaltungsstruktur für die primitiven Objekte beinhaltet.

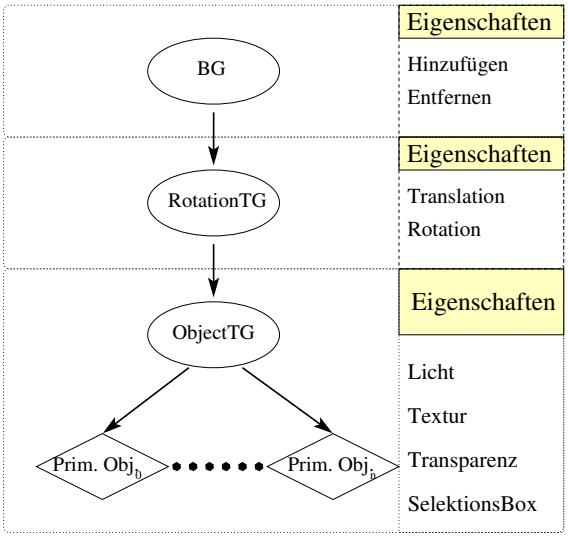

Abbildung 26: Repräsentation eines zusammengesetzten Objektes im Szenengraph

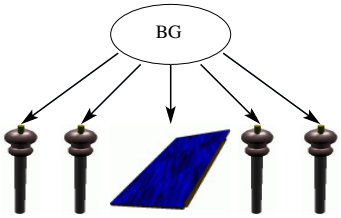

Abbildung 27: Flacher Szenengraph

6.1.3 Kontaktpunkte

Kontaktpunkte bieten die Möglichkeit, Komponenten im Szenengraph so zu verbinden, wie sie auch in der Realität miteinander verbunden sind. Warum diese Repräsentation so wichtig ist, soll durch ein Beispiel verdeutlicht werden. Hierzu soll ein Tisch bestehend aus einer Tischplatte und den vier Tischbeinen in einem Szenengraph dargestellt werden. In Abbildung 27 ist dargestellt, wie eine solcher Tisch in einem Szenengraph dargestellt werden kann. Die Tischbeine und die Tischplatte in der Abbildung können zu diesem Zeitpunkt als zusammengesetzte Objekte betrachtet werden. Im weiteren Verlauf dieses Kapitels werden sie noch zu Komponenten weiterentwickelt, um die nun geschilderten Probleme beheben zu können. In diesem Szenengraph sind die visuellen Objekte direkt mit dem Wurzelknoten verbunden. Dabei befindet sich die Tischplatte im Ursprung des Koordinatensystems. Die Tischbeine sind mittels des *RotationTG*-Knotens so verschoben, dass sie für den Betrachter unter der Tischplatte in ihrer richtigen Position erscheinen. Für den Betrachter macht es keinen Unterschied, ob die Tischbeine im Szenengraph mit der Platte verbunden sind. Aber was passiert, wenn der Tisch mit seiner Platte als Angriffspunkt

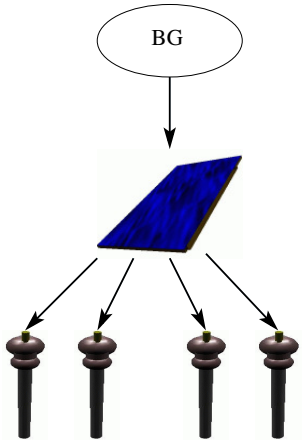

Abbildung 28: Kontaktpunktrepräsentation

bewegt werden soll? In der Realität wird der ganze Tisch bewegt. In einem Szenengraph wird die Platte dadurch verschoben, dass die Translationswerte in dem TransformGroup-Knoten gesetzt werden. Nun wird es für den Betrachter einen Unterschied machen, ob die Beine auch im Szenengraph mit der Platte verbunden sind. Im Fall des flachen Szenengraph wird sich nur die Tischplatte bewegen, nicht aber die Beine. Um hier den kompletten Tisch zu bewegen, müssen die Beine entsprechend der Bewegung der Platte auch verschoben werden, das heisst, dass die Translationswerte der TransformGroup-Knoten der Beine gesetzt werden müssen. Dies sind vier zusätzliche Operationen. In diesem Fall: $T_{trans} * M_{bein(i)}$
Also vier Matrizenmultiplikationen der vorhanden Translationswerte mit der Translationsmatrix.
Nach kurzer Überlegung kommt man dazu, dass es genügt, einen TransformGroup-Knoten zwischen dem Wurzelknoten und dem zusammengesetzten Objekten einzufügen. Für das oben geschilderte Beispiel reicht dies aus.
Behalten wir nun den modifizierten Szenengraph bei. Ein anderes Problem tritt auf, wenn der Tisch mit einem Bein als Angriffspunkt verschoben werden soll. In diesem Fall muss obige Multiplikation für das entsprechende Bein durchgeführt werden und dieser Translationswert muss zu dem Translationswert des eben hinzugefügten TransformGroup-Knotens addiert werden. Solche Operationen sind nicht sehr intuitiv und spiegeln in keinster Weise die Operationen in der Realität wieder. Dies ist nur ein kleines Beispiel. Sollen Operation wie Skalierung, Rotation oder Transformation auf einzelne Komponenten eines komplexeren Produktes angewandt werden, so verschlechtert sich die Situation zusehends.
Es wäre also sehr praktisch, wenn physikalische Verbindungen im Szenengraph erfasst werden könnten. Zu diesem Zweck werden Kontaktpunkte eingeführt. Daraus ergibt sich der in Abbildung 28 dargestellte Szenengraph. Die Kontaktpunkte alleine beheben jedoch noch nicht das Problem, dass eine Teilgraph (Tischbein) manipuliert wird, so dass diese Manipulation sich auch auf Einheiten oberhalb dieses Teilgraphen auswirkt (Tischplatte). Aber in Kapitel 6.2.2

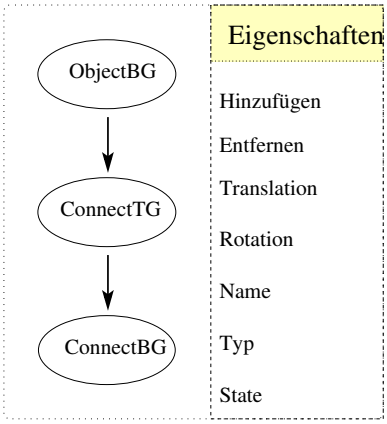

Abbildung 29: Repräsentation eines Kontaktpunktes im Szenengraph

wird auch hierfür eine Lösung gefunden, in der die Kontaktpunkte eine wichtige Rolle spielen.
Ein Kontaktpunkt selbst ist in seiner Graphrepräsentation sehr einfach aufgebaut. Die Abbildung 29 beschreibt diesen Aufbau und die Eigenschaften, die ein Kontaktpunkt besitzt. Er besteht aus einem *ObjectBG*-Knoten, der mit der Komponente verbunden wird, für die der Kontaktpunkt definiert wurde. Der *ConnectBG*-Knoten ist für die Verbindung mit einer anderen Komponente verantwortlich. Bei einer Verbindung wird die andere Komponente als Kind an den *ConnectBG*-Knoten gehängt.
Zwischen dem *ObjectBG*- und dem *ConnectBG*-Knoten befindet sich der *ConnectTG*-Knoten. Er enthält die Positions- und Rotationsdaten des Kontaktpunktes. Desweiteren besitzt er einen Namen und eine Typbezeichnung. Mit deren Hilfe wird festgestellt, mit welcher Komponente oder vielmehr mit welchem Kontaktpunkt einer Komponente dieser Kontaktpunkt verbunden werden kann. Wird ein Kontaktpunkt in einer Verbindung benutzt, so wird sein Zustand als *occupied*, sonst als *free* bezeichnet. Die exakte Prozedur, wie zwei Komponenten verbunden werden, wird in Kapitel 6.2.1 erklärt.

6.1.4 Komponenten

Komponenten sind eine erweiterte Form der zusammengesetzten Objekte. In Abbildung 30 ist die Repräsentation einer Komponente im Szenengraph dargestellt.

Wie alle anderen Einheiten beginnt auch eine Komponente mit einem *BG*-Knoten, der das Entfernen und Hinzufügen der Komponente zur Laufzeit ermöglicht. Darunter folgt eine Anzahl von TransformGroup- Knoten. Drei dieser TransformGroup-Knoten werden für die Animation der Komponente benötigt. Durch diese TransformGroup-Knoten kann die Position, die Rotation, sowie die Skalierung einer Komponente von einem Start- zu einem Endzustand interpoliert werden:

- MotionTG-Knoten: Dieser Knoten enthält die Position der Komponente, ausserdem nutzt der Positionsinterpolator diesen Knoten, um die Kom-

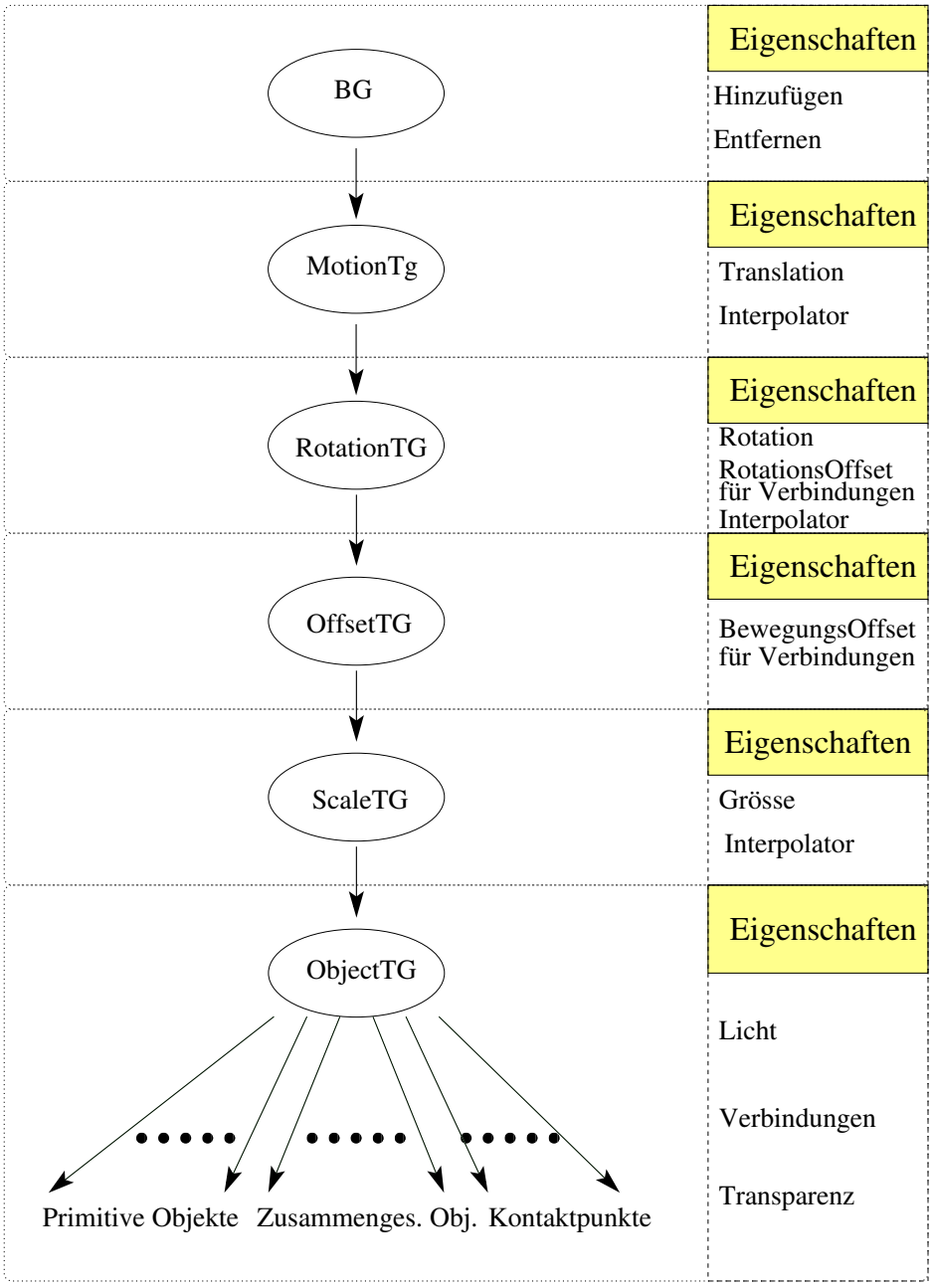

Abbildung 30: Repräsentation einer Komponente im Szenengraph

ponente an der richtigen Stelle zu plazieren.

- RotationTG-Knoten: Dieser Knoten enthält die Rotation der Komponente und wird vom Rotationsinterpolator zur Ausrichtung der Komponente benutzt. Außerdem wird er für die Komponentenverbindungen benötigt (siehe Kapitel 6.2.1).
- ScaleTG: Da eine Komponente zu Beginn einer Animation klein-skaliert im oberen Teil des Animationsfensters vorliegt, wird dieser Knoten genutzt, um sie in der Grösse zu verändern. Ausserdem wird der Knoten vom Skalierungsinterpolator genutzt.

Der OffsetTG-Knoten spielt eine wichtige Rolle bei der Komponentenverbindung. Seine Funktion wird in Kapitel 6.2.1 erläutert. Der untere Teil dieses Graphen besteht aus den primitiven und zusammengesetzten Objekten, die die grafische Darstellung einer Komponente ergeben, sowie aus den Kontaktpunkten. Es können folgende Eigenschaften geändert werden:

- Licht
- Transparenz
- Selektion

6.2 Verbindungen

Im folgenden wird der Szenengraph durch die Einheit *Komponente* aufgebaut. Die Kanten werden durch die Kontaktpunkte gebildet. Bei Verbindungen zwischen Komponenten werden zwei verschiedene Fälle unterschieden. Seien A und B Komponenten, und B soll als Kind an A angefügt werden.
Fall 1:

- A und B sind unabhängig
- B ist Wurzel eines Teilproduktes und A ist unabhängig oder Bestandteil eines Teilproduktes

Fall 2:

- A ist Bestandteil eines Teilproduktes, besitzt eine Komponente als Elter und B ist unabhängig oder Bestandteil eines Teilproduktes

Fall 1 wird in Kapitel 6.2.1 behandelt und Fall 2 in Kapitel 6.2.2.

6.2.1 Verbindungen zwischen Komponenten

Zwei Komponenten A und B können verbunden werden, wenn die folgende Bedingung **M1** gilt:

CONTACTNAME(A)== CONTACTNAME(B)
& CONTACTTYPE(A) $\neq$ CONTACTTYPE(B)
& CONTACT(A)==free & CONTACT(B)==free
und CONTACTTYPE ist entweder *male* oder *female*.

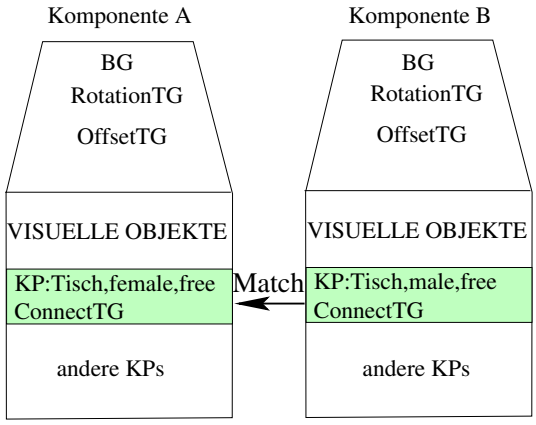

Abbildung 31: Matching zwischen zwei Komponenten

Geht ein Kontaktpunkt eine Verbindung ein, ändert sich sein Zustand. Er wird dann als *occupied* bezeichnet. Wird eine Verbindung zwischen zwei Komponenten gelöst, ändert sich der Zustand der daran beteiligten Kontaktpunkte in *free*. Wie kommt nun eine Verbindung zustande, und wie wird sie im Szenengraph umgesetzt? Um diesen Vorgang beschreiben zu können, wird zunächst eine Komponente genauer definiert: Im folgenden gilt:

- $\mathcal{M} = \mathbb{R}^{4\times 4}$ ist die Menge der Matrizen.
- $\Omega = \mathbb{R}^3$ ist die Menge aller Punkte im Koordinatensystem.
- **ro** ist die Matrix des Rotation-TransformGroup-Knoten und des Offset-TransformGroup-Knoten mit $\mathbf{ro} = M(\text{RotationTG}) * M(\text{OffsetTG})$

Eine Komponente besteht aus einer Menge von Definitionspunkte, sowie einem Identifier, durch den sie eindeutig bestimmt werden kann.

Definition 1 (Komponente) *$k = (\mathsf{P}, id)$ heisst Komponente mit:*

- $\mathsf{P} \subset \Omega$ *ist die endliche Menge von Definitionspunkten, die k im Koordinatensystem beschreibt.*
- $id \in \mathbb{N}$ *ist der Identifier der Komponente.*

Dann ist $\mathcal{K} \subset \Omega \times \mathbb{N}$ die Menge aller Komponenten.
Eine Komponentenverbindung kann dann folgendermaßen beschrieben werden:

1. **Matching**: Die Kontaktpunkte zweier Komponenten werden miteinander verglichen. Wenn sich zwei Kontaktpunkte finden, die die Bedingung aus M1 erfüllen, gibt es einen *Match*, und die beiden Komponenten können miteinander verbunden werden. Dieser Vorgang ist in Abbildung 31 zu sehen. Die Komponenten in dieser Abbildung sind kompakter dargestellt als in Abbildung 30. Es werden nur die zu einer Verbindung nötigen Bestandteile betrachtet.

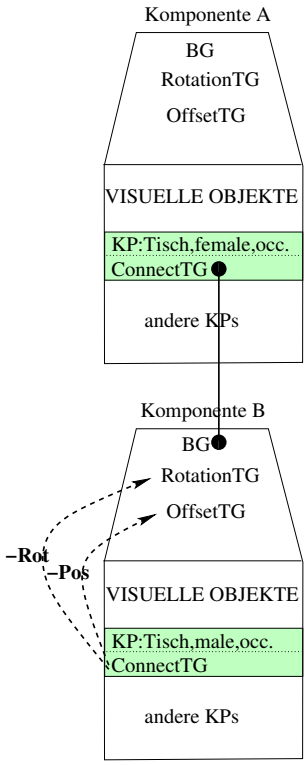

Abbildung 32: Verbindung zwischen zwei Komponenten

2. **Verbinden**: Die Komponenten müssen nun mit Hilfe der beiden Kontaktpunkte verbunden werden. Intuitiv würde man nun einfach die beiden Kontaktpunkte der Komponenten miteinander verbinden. Dies ist jedoch nicht möglich, da die Kontaktpunkte Kinder der Komponenten sind. Eine direkte Verbindung zwischen ihnen würde zu einem Zyklus im Szenengraph führen. Dies ist jedoch verboten. Um nun B an A anzufügen benutzen wir einerseits den Kontaktpunkt von A und andererseits den Wurzelknoten von B (BG-Knoten) und machen B zu einem Kind von A (Abbildung 32). Folgende Definition beschreibt die Verbindung zwischen zwei Komponenten:

   Definition 2 (Verbindung) $v = ((k, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}')$ *ist eine Verbindung mit:*

   - (k, k') *mit* $k, k' \in \mathcal{K}, k \neq k'$
     k *heisst Elter von* k'
     k' *heisst Kind von* k
   - $\mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}' \in \mathcal{M}$
   - $\mathbf{ro}' = \mathbf{kp}'^{-1}$

   Im folgenden ist $\mathcal{V} \subset (\mathcal{K} \times \mathcal{K}) \times (\mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M})$ die Menge dieser Verbindungen.

Wir schreiben für $((k,k'),\mathbf{ro},\mathbf{kp},\mathbf{ro}',\mathbf{kp}')$ auch $((k,k'),\overline{m})$, wenn die Informationen der Matrizen nicht benötigt werden.

3. **Daten umkopieren**: Durch den Prozess im letzten Schritt haben wir die Information des Kontaktpunktes von B nicht an der richtigen Position im Szenengraph. Somit stimmt die Relation zwischen A und B nicht. Um diesen Umstand zu beheben werden die Positionsdaten aus dem Kontaktpunkt von B negiert in den *OffsetTG*-Knoten und die Rotationsdaten negiert in den *RotationTG*-Knoten kopiert. Warum werden die Daten negiert kopiert? Dazu ziehen wir die Betrachtung der Transformgruppen-Knoten als lokale Koordinatensysteme heran (siehe Kapitel 2.3). Um eine Ausrichtung von B zu A so zu erreichen, dass die Kontaktpunkte in ihrer Position und Rotation übereinstimmen, muss der Kontaktpunkt von B in den Ursprung seines Koordinatensystems verschoben werden. Zusätzlich muss er die Rotation von 0° besitzen. Dieser Vorgang, eine Komponente auszurichten ist in Abbildung 33 dargestellt. Nun hat B die Lage, die durch die Kontaktpunkte von A und B vorgeschrieben wird. Mit Hilfe der folgenden Funktion lassen sich die Daten einer Verbindung umkopieren:

   Definition 3 (Daten umkopieren) $\mathsf{du}\colon \mathcal{V} \longrightarrow \mathcal{V}$
   Sei $v = ((k,k'),\mathbf{ro},\mathbf{kp},\mathbf{ro}',\mathbf{kp}') \in \mathcal{V}$
   Dann gilt:
   $\mathsf{du}(v) = v' = ((k,k'),\mathbf{ro}'',\mathbf{kp},\mathbf{ro}''',\mathbf{kp}')$ *mit:*
   $\mathbf{ro}'' = \mathbf{kp}^{-1}$
   $\mathbf{ro}''' = \mathbf{I}$

Desweiteren wird eine Funktion benötigt, die uns die Matrix einer Verbindung liefert. Dazu muss die Matrix des beteiligten Kontaktpunktes des Elter mit der RotationOffset-Matrix des Kindes multipliziert werden.

Definition 4 (Multiplikation der Verbindungsmatrizen) $\mathsf{mv} : \mathcal{V} \longrightarrow \mathcal{M}$
Sei $v = ((k,k'),\mathbf{ro},\mathbf{kp},\mathbf{ro}',\mathbf{kp}') \in \mathcal{V}$.
Dann gilt:
$\mathsf{mv}(v) = \mathbf{kp} * \mathbf{ro}'$

Dieser Vorgang, zwei Komponenten miteinander zu verbinden, ist die Grundlage für die im folgenden Kapitel geschilderte Verbindungsart.

6.2.2 Verbindungen zwischen Produktteilen

Als erstes soll hier auf das Problem eingegangen werden, das auftritt wenn in dem Szenengraph einzelne Komponenten manipuliert werden sollen. Wir betrachten wiederum die Komponenten als die Einheiten, durch die der Szenengraph aufgebaut wird. Die visuellen Bestandteile der Komponenten bilden die Knoten. Die Kanten werden durch die Gruppenknoten gebildet, die benötigt werden, um eine Verbindung zwischen zwei Komponenten herzustellen.

- Von der **Elter-Komponente** sind dies die Knoten des Kontaktpunktes, der an dieser Verbindung beteiligt ist:

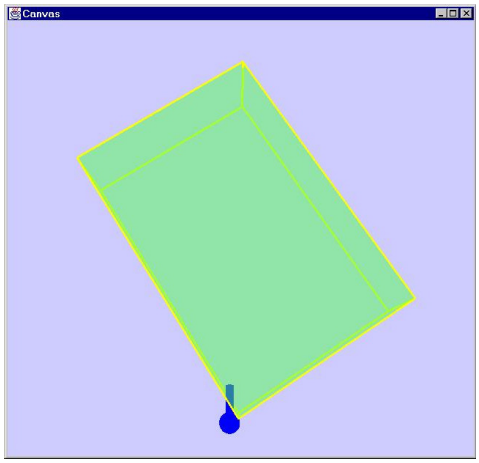

Abbildung 33: Ausrichtung einer Komponente nach ihrem Kontaktpunkt

  - ObjectBG, ConnectTG, ConnectBG

- Von der **Kind-Komponente** sind dies:

  - BG, RotationTG, OffsetTG

  Wobei der RotationTG-, sowie der OffsetTG-Knoten die negierten Werte des beteiligten Kontaktpunktes der Kind-Komponente beinhalten.

Alle Verbindungen werden wie in Kapitel 6.2.1 geschildert aufgebaut. In Kapitel 6.1.3 wurde ein Problem geschildert, in dem ein Tisch mit einem Bein als Angriffspunkt verschoben werden soll. Der Tisch liegt im Szenengraph so vor, dass wir eine Tischplatte als Wurzelelement und die Beine als Kinder der Wurzel vorliegen haben. Eine solche Verschiebung wird normalerweise im Szenengraph dadurch erreicht, dass in einem TransformGroup-Knoten zwischen Platte und Bein die Translationswerte gesetzt werden. Das Ergebnis wäre ein verschobenes Tischbein, wobei der Rest an seiner alten Position bestehen bleibt. Der erste Ansatz ist also die neue Information durch den Szenengraph zu propagieren. Die Information muss von dem geänderten Knoten über den Pfad bis zum Wurzelknoten propagiert werden.
Dabei muss berücksichtigt werden, in welcher Relation das jeweilige Kind zum Elter-Knoten steht. Betrachten wir nun die TransformGroup-Knoten als Matrizen und fassen die TransformGroup-Knoten $OffsetTG$ und $RotationTG$ zu einer Matrix zusammen. So erhalten wir eine vereinfachte Darstellung derart, dass zwischen zwei verbundenen Komponenten zwei TransformGroup-Knoten (Matrizen) liegen. Wenn wir von Matrizen reden, müssen Komponenten entsprechend als Punktmengen dargestellt werden. Für zwei Komponenten A und B, die miteinander verbunden sind, ergibt sich dann folgende Schreibweise, um ihre Lage L im Weltkoordinatensystem zu bestimmen:
$L(A) = M_A * A$
$L(B) = M_A * M_{AB} * M_{BA} * B$

Unter dem Begriff "Lage" verstehen wir hier die Translation und Rotation einer Komponente im Weltkoordinatensystem. M_A ist die Matrix, die die Lage von A beschreibt, M_{AB} ist die Matrix die die Lage des Kontaktpunktes von A beschreibt und M_{BA} ist die Matrix die die Lage des Kontaktpunktes von B beschreibt.
Soll B nun eine neue Lage gegeben werden so muss diese Information bis zur Wurzel propagiert werden. Sei M_I diese neue Information. So ergibt sich folgende Berechnung um M_A neu zu bestimmen:
$M_A = M_A^{-1} * M_{AB}^{-1} * M_{BA}^{-1} * M_I$
Dies ergibt 3 Matrizenmultiplikationen. Muss eine Information also über n Kanten propagiert werden, so erhalten wir $n+1$ Multiplikationen. Die Multiplikationen der Matrizen von der zu manipulierenden Komponente bis zur Wurzel können vorberechnet werden. Somit bleibt noch eine Multiplikation übrig. Dieser Ansatz ist nicht schlecht, hat jedoch den Nachteil, dass er nicht wirklich die Operation wiederspiegelt, die in der Realität ausgeführt wird. Viel wichtiger ist jedoch, dass dieser Ansatz nicht für einen dynamisch wachsenden Szenengraph verwendet werden kann, da hierdurch die Konsistenz nicht erhalten werden kann. In diesem Fall könnten die realen Verbindungen nicht auf den Szenengraph übertragen werden
Wenn nun zwei Produktteile PT_1 und PT_2 miteinander verbunden werden sollen, kann genau das oben geschilderte Problem auftauchen. Es tritt dann auf, wenn zwei Komponenten A und B so miteinander verbunden werden sollen, dass B zum Kind von A gemacht werden soll. A befindet sich in PT_1 und B in PT_2 und B besitzt einen Elter-Knoten. Die angestrebte Lösung ist es B zur Wurzel des Teilproduktes zu machen ohne die visuellen Eigenschaften dieses Teilproduktes zu verändern. Und genau diese Lösung wird durch die Kontaktpunkte ermöglicht. Im folgenden wird gezeigt, wie dies zu bewerkstelligen ist. Desweiteren wird bewiesen, dass diese Umformungen weder das Aussehen noch die Verbindungen, die die Komponenten untereinander haben, beeinträchtigen.
Um die Lage einer Komponente zu ändern, wollen wir einfach nur die Information dieser einen Komponente verändern. Für eine beliebige Komponente A im Szenengraph bedeutet das, dass sie zur Wurzel dieses Graphen gemacht werden muss.
Zunächst soll aber der Szenengraph definiert werden, der eine Baumstruktur bildet.

Definition 5 (Szenengraph) *$s = (\mathsf{K}, \mathsf{V}, k_0)$ ist ein Szenengraph mit:*

- $\mathsf{K} \subset \mathcal{K}$ *ist endliche Menge von Komponenten,* $\mathsf{K} = \{k_0, k_1, ..., k_n\}$ *und* $\forall k_i, k_j$ *mit* $k_i = (\mathsf{P}_i, id_i), k_j = (\mathsf{P}_j, id_j) : id_i \neq id_j$ *für* $0 \leq i, j \leq n, i \neq j$

- $\mathsf{V} \subset \mathcal{V}$ *ist endliche Menge von Verbindungen.*
  $\forall k' \in \mathsf{K}, k' \neq k_0 : \exists! k \in \mathsf{K} : ((k, k'), \overline{m}) \in \mathsf{V}$
  k heisst Elter von k'.
  k' heisst Kind von k.

- $k_0 \in \mathsf{K}$ *ist die Wurzel von s,* $\nexists k \in \mathsf{K} : (k, k_0) \in \mathsf{V}$

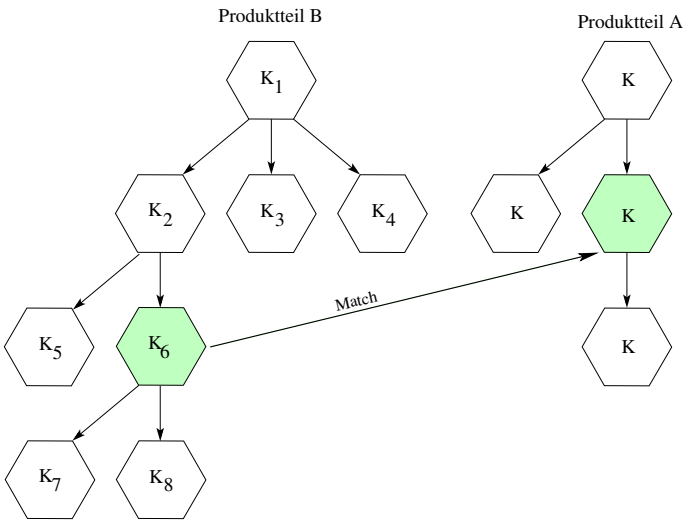

Abbildung 34: Matching

In einem solchen Szenengraph sind die ids aller Komponenten verschieden. Die Menge der Szenengraphen wird dann als $\mathcal{S} \subset \mathcal{P}(\mathcal{K}) \times \mathcal{P}(\mathcal{V}) \times \mathcal{K}$ bezeichnet.

Um nun eine Komponente zur Wurzelkomponente zu machen, werden folgende Schritte ausgeführt:

1. **Pfadbestimmung**: Der Pfad von B zum Wurzelknoten wird bestimmt. Hierbei werden die Kanten von B in Richtung Wurzel verfolgt und die besuchten Knoten gemerkt. Ein Pfad kann dann folgendermaßen beschrieben werden:

   Definition 6 (Pfad) *Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ ein Szenengraph. Dann heisst P der Komponentenpfad von k_0 nach k in s.*
   P: $\mathcal{S} \times K \longrightarrow K^{n+1}$
   $P(s,k) = (k_0, k_1, ..., k_n)$ mit $k_i \in \mathsf{K}$ für $0 \leq i \leq n$, $((k_i, k_{i+1}), \overline{m_i}) \in \mathsf{V}$ für $0 \leq i < n$ und $k_n = k$.

2. **Neuanordnung**: Um den Szenengraph neu anzuordnen, wird die folgende Operation auf die Komponenten im oben bestimmten Pfad angewandt:

   Sei X Elter von Y auf dem bestimmten Pfad und X ist aktuelle Wurzel. Dann werden folgende Schritte ausgeführt, um X und Y zu vertauschen.

   - Löse Y von X.
   - Lösche vorhandene Werte aus dem *RotationTG*- und *OffsetTG*-Knoten von Y.
   - Kopiere die Werte des für diese Verbindung benötigten Kontaktpunktes von X negiert in die Knoten *RotationTG* und *OffsetTG* von X.

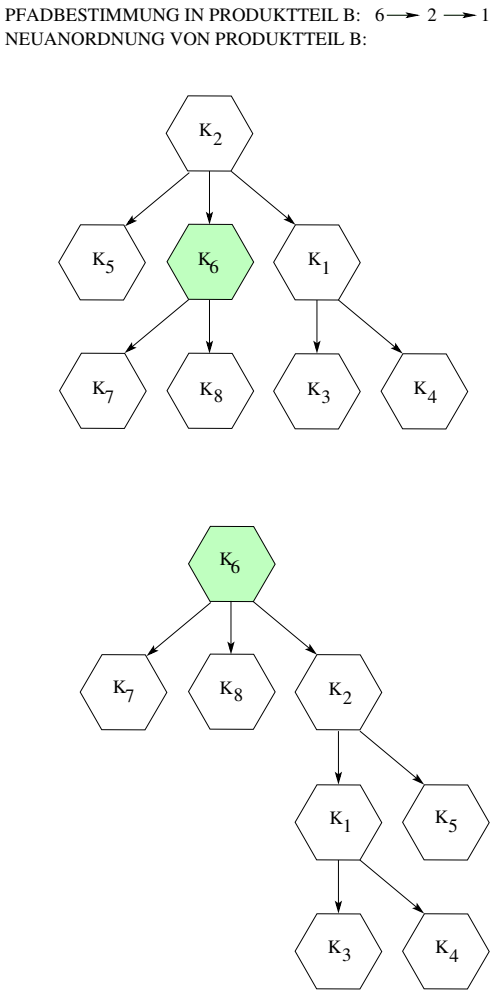

Abbildung 35: Neuanordnung

- Füge X mit seinem BG-Knoten als Kind von Y an dessen beteiligten Kontaktpunkt an.

Hiermit wird die Elter-Kind-Relation umgekehrt, unter Berücksichtigung der vorhandenen Rotations- und Translationswerte. So besteht auch nach dem Vertauschen eine Verbindung, wie sie in Kapitel 6.2.1 beschrieben wurde.

Um diese Vertauschoperation für einen Szenengraph formal definieren zu können, benötigen wir noch eine Zusatzfunktion, die nur die Elter-Kind-Relation der Wurzel und eines Kindes dieser Wurzel umkehrt.

Definition 7 (Vertausche Komponenten) vk*:* $\mathcal{S} \times \mathcal{V} \longrightarrow \mathcal{S}$
Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ *ein Szenengraph,* $k, k' \in \mathsf{K}$,
$v = ((k, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro'}, \mathbf{kp'}) \in \mathsf{V}, k = einek_0$
Dann gilt:
$\mathsf{vk}(s, v) = (\mathsf{K}, \mathsf{V}', k_0')$ *mit*
$\mathsf{V}' = \{\mathsf{V} \setminus \{((k, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro'}, \mathbf{kp'})\}\} \cup \{((k', k), \mathbf{ro'}, \mathbf{kp'}, \mathbf{ro}, \mathbf{kp})\}$
$k_0' = k'$

Mit Hilfe dieser Komponentenvertauschung und der eben definierten Funktion du lässt sich jetzt die Vertauschoperation folgendermaßen darstellen:

Definition 8 (Vertauschoperation) vo*:* $\mathcal{S} \times \mathcal{V} \longrightarrow \mathcal{S}$
Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ *ein Szenengraph,* $k_0, k' \in \mathsf{K}$, $v = ((k_0, k'), \overline{m}) \in \mathsf{V}$.
Dann ist vo *die Vertauschoperation:*
$\mathsf{vo}(s, v) = \mathsf{vk}(s, \mathsf{du}(v))$

Für den ganzen Pfad ist der Vorgang in Grafik 35 dargestellt und kann folgendermaßen beschrieben werden:
Die Knoten des Pfades liegen in dem Feld `pathComponents`. Das erste Element ist der Wurzelknoten.

```
.
.
i=0;
while(i+1 < pathComponents.length) {
  /*referenziere Elter, der in der
    folgenden Vertauschoperation zum Kind gemacht wird*/
  newChild = pathComponents[i];
  /*referenziere Kind, das in der
    folgenden Vertauschoperation zum Elter gemacht wird*/
  newParent = pathComponents[i+1];
  /*Löse das Kind von seinem Elter
  newParent.detach();
  /*Füge den alten Elter an das alte Kind, wie oben beschrieben, an*/
  newParent.addChild(newChild);
  i++;
```

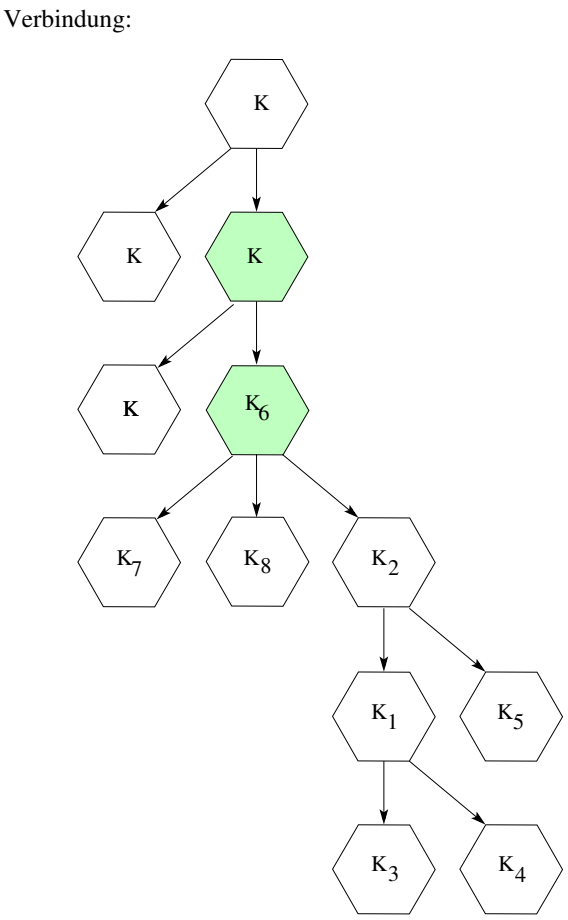

Abbildung 36: Verbindung zwischen Produktteilen

```
}
.
.
```

Wenn nun die gewünscht Komponente in PT_2 zur Wurzel dieses Graphen gemacht wurde, kann der gesamte Graph (PT_2) mit B als Wurzel an A in PT_1 angehängt werden. Der gesamte Vorgang der Pfadbestimmung, Vertauschung und Verknüpfung ist in den Abbildungen 34, 35 ,36 dargestellt.

Um die Skalierung, Rotation und Translation einer Komponente im Weltkoordinatensystem zu bestimmen, müssen alle Matrizen auf dem Pfad von der Wurzel bis zu dieser Komponente aufmultipliziert werden. Daraus ergibt sich die folgende Definition einer Transformationsmatrix für eine Komponente.

Definition 9 (Transformationsmatrix einer Komponente) *Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ ein Szenengraph. Die Transformationsmatrix einer Komponente k in s wird dann folgendermaßen bestimmt:*

tm: $\mathcal{S} \times \mathsf{K} \longrightarrow \mathcal{M}$
$\mathsf{tm}(s,k) = \prod_{i=0}^{n-1} \mathsf{mv}((k_i, k_{i+1}), \overline{m_i})$ *mit* $(k_0, ..., k_n) = P(s,k)$

Nun soll der Beweis erbracht werden, dass die Vertauschoperation die Verbindungen, die aus der Realität in den Szenengraph übertragen wurden, nicht beeinträchtigt werden. Dazu wird zunächst bewiesen, dass durch eine erweiterte Vertauschoperation weder Rotation, noch Skalierung, noch Translation der Komponenten im Szenengraph verändert werden. Hierfür müssen noch drei, der schon vorgestellten Funktionen, erweitert werden.
Im folgenden soll **ro** der aktuellen Wurzelkomponente k_0 dazu benutzt werden, Matrizen aufzumultiplizieren. Das hat den Effekt, dass sich die Transformationsmatrizen der Komponenten nach einer Vertauschoperation nicht ändern.

Definition 10 (Erweitertes Umkopieren der Daten) du_E: $\mathcal{V} \longrightarrow \mathcal{V}$
Sei $v = ((k,k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}') \in \mathcal{V}$ *mit* $k, k' \in \mathcal{K}, k \neq k'$
Dann gilt:
$\mathsf{du}_E(v) = v' = ((k,k'), \mathbf{ro}'', \mathbf{kp}, \mathbf{ro}''', \mathbf{kp}')$ *mit:*
$\mathbf{ro}'' = \mathbf{kp}^{-1}$
$\mathbf{ro}''' = \mathbf{ro} * \mathbf{kp} * \mathbf{ro}'$

Dadurch erhalten wir die erweiterte Vertauschoperation:

Definition 11 (Erweiterte Vertauschoperation) vo_E: $\mathcal{S} \times \mathcal{V} \longrightarrow \mathcal{S}$
Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ *ein Szenengraph,* $v = ((k_0, k'), \overline{m}) \in \mathsf{V}$. *Dann ist* vo_E *die erweiterte Vertauschoperation:*
$\mathsf{vo}_E(s,k) = \mathsf{vk}(s, \mathsf{du}_E(v))$

Eine solche erweiterte Vertauschoperation ist in Abbildung 37 dargestellt.
Nun muss noch die Transformationsmatrix einer Komponente an die neuen Begebenheiten angepasst werden. In diesem Fall heisst das, dass zusätzlich zur ursprünglichen Transformationsmatrix, noch **ro** der aktuellen Wurzelkomponente berücksichtigt werden muss. Die erweiterte Transformationsmatrix wird dann folgendermaßen definiert:

Definition 12 (Erweiterte Transformationsmatrix einer Komponente)
Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ *ein Szenengraph. Die erweiterte Transformationsmatrix wird dann folgendermaßen bestimmt:*
tm_E: $\mathcal{S} \times \mathcal{K} \longrightarrow \mathcal{M}$

- *Für* $n = 1$: *Sei* $k' \in \mathsf{K}$, $v = ((k_0, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}') \in \mathsf{V}$.
  Dann gilt:
  $\mathsf{tm}_E(s, k_0) = \mathbf{ro}$

- *Für* $n \geq 2$: $v = ((k_0, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}')$.
  Dann gilt:
  $\mathsf{tm}_E(s,k) = \mathbf{ro} * \prod_{i=0}^{n-1} \mathsf{mv}((k_i, k_{i+1}), \overline{m_i})$, *mit* $(k_0, ..., k_n) = P(s,k)$

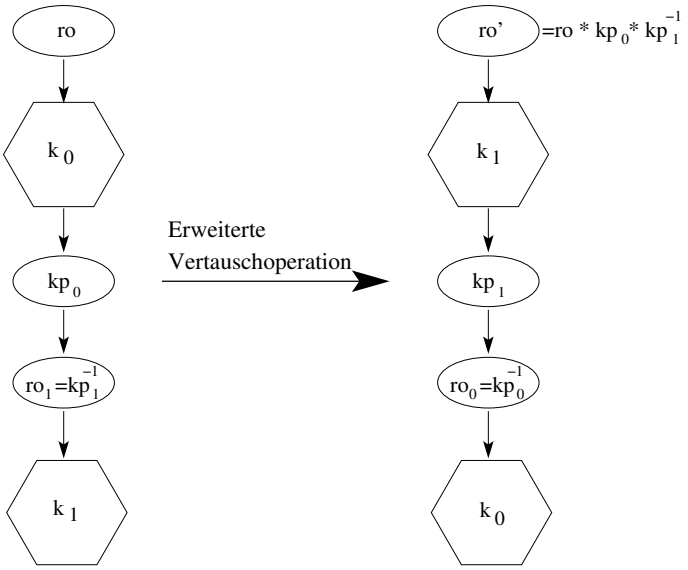

Abbildung 37: Eine erweiterte Vertauschoperation

Nun kann gezeigt werden, dass die erweiterte Vertauschoperation keine Auswirkung auf die erweiterten Transformationsmatrizen der Komponenten hat. Für den Betrachter bedeutet das, dass sich trotz erweiterter Vertauschoperation die Lage der Komponenten in der virtuellen Welt nicht ändert.

Satz 1 Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ ein Szenengraph, $v = ((k, k'), \overline{m}) \in \mathsf{V}, k = k_0$.
Dann gilt:
$\forall k'' \in \mathsf{K}$: $\mathsf{tm}_E(s, k'') = \mathsf{tm}_E(\mathsf{vo}_E(s, v), k'')$.

Die erweiterte Vertauschoperation ändert die erweiterte Transformationsmatrix der Komponenten im Szenengraph nicht. Es reicht zu zeigen, dass die beiden, an der Operation beteiligten Komponenten, ihre erweiterte Transformationsmatrix beibehalten. Dies genügt, da eine dieser vertauschten Komponenten, oder beide, immer den Anfang eines Pfades einer beliebigen anderen Komponente im Szenengraph bildet.

Beweis 1 Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ ein Szenengraph und $v = ((k, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}') \in \mathsf{V}, k = k_0$
Zu zeigen:

1. $\mathsf{tm}_E(s, k) = \mathsf{tm}_E(\mathsf{vo}_E(s, v), k)$
2. $\mathsf{tm}_E(s, k') = \mathsf{tm}_E(\mathsf{vo}_E(s, v), k')$

Zu 1:

- Linke Seite der Gleichung:
  $\mathsf{tm}_E(s, k) = \mathsf{tm}_E(s, k_0) = \mathbf{ro}$

- Rechte Seite der Gleichung:
  $\mathsf{tm}_E(\mathsf{vo}_E(s, v), k)$
  $= \mathsf{tm}_E(\mathsf{vk}(s, \mathsf{du}_E(v)), k)$
  $= \mathsf{tm}_E(\mathsf{vk}(s', v'), k)$ mit
  $s' = (\mathsf{K}, \mathsf{V}', k_0)$,
  $\mathsf{V}' = \{\mathsf{V} \setminus \{v\}\} \cup \{v'\}$,
  $v' = ((k, k'), \mathbf{ro}'', \mathbf{kp}, \mathbf{ro}''', \mathbf{kp}')$ mit
  $\mathbf{ro}'' = \mathbf{kp}^{-1}$,
  $\mathbf{ro}''' = \mathbf{ro} * \mathbf{kp} * \mathbf{ro}' = \mathbf{ro} * \mathbf{kp} * \mathbf{kp}'^{-1}$
  $= \mathsf{tm}_E(s'', k)$ mit
  $s'' = (\mathsf{K}, \mathsf{V}'', k_0')$,
  $\mathsf{V}'' = \{\mathsf{V}' \setminus \{v'\}\} \cup \{v''\}$,
  $v'' = ((k', k), \mathbf{ro}''', \mathbf{kp}', \mathbf{ro}'', \mathbf{kp})$,
  $k_0' = k'$
  $= \mathbf{ro}''' * \mathbf{kp}' * \mathbf{ro}'' = \mathbf{ro} * \mathbf{kp} * \mathbf{kp}'^{-1} * \mathbf{kp}' * \mathbf{kp}^{-1} = \mathbf{ro}$

Zu 2: Dieser Teil des Beweises läuft analog zu Teil 1.

□

Somit wurde gezeigt, dass die erweiterte Transformationsmatrix, die die Rotation, Skalierung und Translation einer Komponente im Weltkoordinatensystem beschreibt, durch die Vertauschoperation nicht geändert wird. Alle anderen Komponenten des Graphen stehen in Relation zu einer oder beiden Komponenten, die an der erweiterten Vertauschoperation beteiligt sind. Daher ändert sich deren erweiterte Transformationsmatrix auch nicht.
Jetzt ist leicht nachzuvollziehen, dass die normale Vertauschoperation die Anordnung der Komponenten untereinander erhält. Für den Betrachter bedeutet dies, dass die Komponenten in der virtuellen Welt mit ihren Kontaktpunkten untereinander verbunden bleiben. Der Unterschied zwischen den beiden Operationen besteht lediglich darin, dass nach der normalen Vertauschoperation die Wurzelkomponente immer im Ursprung des Weltkoordinatensystems liegt.

Satz 2 Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ ein Szenengraph, $((k, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}') \in \mathsf{V}$ mit $k = k_0$ und $\mathsf{tm}_E(\mathsf{vo}_E(s, v), k) = \mathsf{tm}_E(s'', k)$ mit $s'' = (\mathsf{K}, \mathsf{V}'', k_0')$ und $v'' = ((k', k), \mathbf{ro}''', \mathbf{kp}', \mathbf{ro}'', \mathbf{kp}) \in \mathsf{V}''$
Dann gilt:
$\forall k'' \in \mathsf{K}$: $\mathsf{tm}(\mathsf{vo}(s, v), k'') = \mathbf{ro}'''^{-1} * \mathsf{tm}_E(\mathsf{vo}_E(s, v), k'')$.

Der Beweis wird wieder über die beiden, an der Vertauschoperation beteiligten Komponenten geführt.

Beweis 2 Sei $s = (\mathsf{K}, \mathsf{V}, k_0)$ ein Szenengraph und $((k, k'), \mathbf{ro}, \mathbf{kp}, \mathbf{ro}', \mathbf{kp}') \in \mathsf{V}$ mit $k = k_0$ und $\mathsf{tm}_E(\mathsf{vo}_E(s, v), k) = \mathsf{tm}_E(s'', k)$ mit $s'' = (\mathsf{K}, \mathsf{V}'', k_0')$ und $v'' = ((k', k), \mathbf{ro}''', \mathbf{kp}', \mathbf{ro}'', \mathbf{kp}) \in \mathsf{V}''$
Zu zeigen:

1. $\mathsf{tm}(\mathsf{vo}(s,v),k) = \mathbf{ro}'''^{-1} * \mathsf{tm}_E(\mathsf{vo}_E(s,v),k)$
2. $\mathsf{tm}(\mathsf{vo}(s,v),k') = \mathbf{ro}'''^{-1} * \mathsf{tm}_E(\mathsf{vo}_E(s,v),k')$

Zu 1:

- Linke Seite der Gleichung:
  $\mathsf{tm}(vo(s,v),k)$
  $= \mathsf{tm}(\mathsf{vk}(s,\mathsf{du}(v)),k)$
  $= \mathsf{tm}(\mathsf{vk}(s',v'),k)$ mit
  $s' = (\mathsf{K},\mathsf{V}',k_0)$,
  $\mathsf{V}' = \{\mathsf{V}\setminus\{v\}\}\cup\{v'\}$,
  $v' = ((k,k'),\mathbf{ro}'',\mathbf{kp},\mathbf{ro}''',\mathbf{kp}')$ mit
  $\mathbf{ro}'' = \mathbf{kp}^{-1}$,
  $\mathbf{ro}''' = \mathbf{I}$
  $= \mathsf{tm}(s'',k)$ mit
  $s'' = (\mathsf{K},\mathsf{V}'',k_0')$,
  $\mathsf{V}'' = \{\mathsf{V}\setminus\{v'\}\}\cup\{v''\}$,
  $v'' = ((k',k),\mathbf{ro}''',\mathbf{kp}',\mathbf{ro}'',\mathbf{kp})$,
  $k_0' = k'$
  $= \mathbf{kp}' * \mathbf{ro}'' = \mathbf{kp}' * \mathbf{kp}^{-1}$

- Rechte Seite der Gleichung:
  $\mathbf{ro}'''^{-1} * \mathsf{tm}_E(\mathsf{vo}_E(s,v),k)$
  $= \mathbf{ro}'''^{-1} * \mathsf{tm}_E(\mathsf{vk}(s,\mathsf{du}_E(v)),k)$
  $= \mathbf{ro}'''^{-1} * \mathsf{tm}_E(\mathsf{vk}(s',v'),k)$ mit
  $s' = (\mathsf{K},\mathsf{V}',k_0)$,
  $\mathsf{V}' = \{\mathsf{V}\setminus\{v\}\}\cup\{v'\}$,
  $v' = ((k,k'),\mathbf{ro}'',\mathbf{kp},\mathbf{ro}''',\mathbf{kp}')$ mit
  $\mathbf{ro}'' = \mathbf{kp}^{-1}$,
  $\mathbf{ro}''' = \mathbf{ro} * \mathbf{kp} * \mathbf{ro}' = \mathbf{ro} * \mathbf{kp} * \mathbf{kp}'^{-1}$
  $= \mathbf{ro}'''^{-1} * \mathsf{tm}_E(s'',k)$ mit
  $s'' = (\mathsf{K},\mathsf{V}'',k_0')$,
  $\mathsf{V}'' = \{\mathsf{V}\setminus\{v'\}\}\cup\{v''\}$,
  $v'' = ((k',k),\mathbf{ro}''',\mathbf{kp}',\mathbf{ro}'',\mathbf{kp})$,
  $k_0' = k'$
  $= \mathbf{ro}'''^{-1} * \mathbf{ro}''' * \mathbf{kp}' * \mathbf{ro}''$
  $= \mathbf{kp}' * \mathbf{kp}^{-1} * \mathbf{ro}^{-1} * \mathbf{ro} * \mathbf{kp} * \mathbf{kp}'^{-1} * \mathbf{kp}' * \mathbf{kp}^{-1} = \mathbf{kp}' * \mathbf{kp}^{-1}$

Zu 2: Dieser Teil des Beweises läuft analog zu Teil 1.

□

Die Transformationsmatrizen aller Komponenten im Szenengraph unterscheiden sich nach einer normalen Vertauschoperation in der Matrix **ro** der Wurzelkomponente k_0. Für den Betrachter bedeutet dies, dass die aktuelle Wurzelkomponente immer im Ursprung des Weltkoordinatensystems liegt und die Anordnung der Komponenten untereinander durch die Vertauschoperation nicht geändert wird.

7 Zusammenfassung

In diesem Kapitel soll noch einmal kurz der Inhalt der vorliegenden Arbeit dargestellt werden.
Das Ziel war es, eine Möglichkeit zu bieten, komponentenbasierte Objekte aus der realen Welt in der virtuellen Welt darzustellen. Dies soll aber im Gegensatz zu CAD-Programmen mit einfacheren Mitteln geschehen. Zudem soll die Datenmenge nicht zu stark anwachsen.
Es wurden drei Ebenen betrachtet. Jede dieser Ebenen besitzt eine spezielle Sichtweise auf die Komponenten.

- **Das Benutzerinterface *GenAu-3D***: Diese Ebene erlaubt dem Benutzer Objekte mit Hilfe von primitiven Objekten approximativ zu visualisieren. Primitive Objekte sind Quader, Zylinder, Kugeln und Kegel. GenAu-3D besteht aus:

  - Meta-Autorentool: Mit dem Meta-Autorentool lassen sich Objekte vordefinieren. Aus diesen Objekten und einem generischen Autorentool wird das domänenspezifische Autorentool kreiert.
  - Autorentool: In einem domänenspezifischen Autorentool können Komponenten mit Hilfe der vordefinierten sowie der primitiven Objekte erstellt werden. Ausserdem werden hier Kontaktpunkte eingefügt, die die Lage der Komponenten zueinander festlegen.
  - Konfigurationstool: Mit dem Konfigurationstool werden Produkte konfiguriert. Die dafür benötigten Komponenten werden im Autorentool erstellt.
  - Animationstool: Das Animationstool soll zeigen, was mit dem Objektmodell und der Art wie der Szenengraph verwaltet wird, möglich ist. Die Produktkonfigurationen werden als Animation angezeigt, in der die Komponenten Schritt für Schritt zusammengefügt werden. Genauso können auch fertige Produkte erstellt werden, die intern aus Komponenten bestehen.

- **XML-Objektmodell**: In dem Objektmodell, das durch eine DTD beschrieben ist, werden die Daten erfasst und können zur Wiederverwertung abgespeichert werden. Diese Dateien werden mittels eines Parsers ausgelesen, um die grafischen Objekte, sowie Eigenschaften und Beschreibungen dieser Objekte wieder herzustellen.

- **Szenengraphverwaltung**: Die Szenengraphverwaltung ist ein wichtiger Bestandteil. Die Komponenten werden als Einheiten in dem Szenengraph aufgefasst und besitzen eine Schnittstelle über die ihre Eigenschaften verändert werden können. Über die Kontaktpunkte der Komponenten werden Verbindungen zwischen ihnen hergestellt, wobei diese Kontaktpunkte eine Abbildung der realen Verbindungen auf den Szenengraph zulassen. Es gibt Einschränkungen, da der Szenengraph keine Zyklen beinhalten darf. Ausserdem ist es möglich, jede Komponente zur Wurzel

des Szenengraph zu machen, ohne die Darstellung zu beeinflussen. Dies lässt realitätsnahe Operationen auf dem Szenengraph zu.

In der vorliegenden Arbeit wurden oft die Begriffe *Produkt* und *Komponente* gebraucht. Dieses Gebiet wurde wegen seinem aktuellen Bezug gewählt. Genausogut können Konstrukte erfasst werden, die sich auf irgendeine Art und Weise aus mehreren Bestandteilen zusammensetzen, z.B.: Molekülmodelle, deren verschiedene Atome und Bindungen farblich unterschiedlich dargestellt werden können.
Diese Arbeit bietet eine gute Grundlage um komponentenbasierte Applikation und Applets zu erstellen, insbesondere in den Bereichen des e-Commerce, in denen die Plastizität eine grosse Rolle spielt.

8 Ausblick

Da bei der Entwicklung der Werkzeuge auf Modularität und Objektorientiertheit gesetzt wurde, können mit relativ wenig Aufwand Erweiterungen hinzugefügt werden. Zur Zeit wird Java3D noch unzureichend unterstützt. Wünschenswert wäre eine Aufnahme der Java-3D-Klassen in die core-Klassen von Java. Somit wären Applets in Webbrowsern möglich, ohne vorher eine Laufzeitumgebung installieren zu müssen. Gleiches gilt für eine Applikation. Eine Java-Laufzeitumgebung würde für die Ausführung der Applikationen genügen. Da aber das Java3D-API noch nicht ganz ausgereift ist, ist die abwartende Haltung der Hersteller verständlich. Eventuell kann man auf die nächsten Browsergeneration hoffen.
Desweiteren ist eine Koexistenz zwischen der bestehenden Beschreibungssprache *VRML*, der in der Entstehung begriffenen Sprache *X3D* und Java3D möglich, da sie verschiedene Ansätze verfolgen.
Für die Zukunft wäre es durchaus denkbar aus den bestehenden Klassen dieser Arbeit ein *API* zu entwickeln, mit dessen Hilfe man auf komfortable Art und Weise Objekte kreieren und verwalten kann.
Betrachtet man GenAu-3D selbst, so kann man immer Verbesserungen anbringen. Vorrangig wäre ein ausgefeiltes Design des Benutzerinterfaces. Desweiteren könnte ein Editor eingefügt werden, um freiformbare Objekte zu erstellen, mit deren Hilfe Objekte aus der Realität besser nachempfunden werden können. Es könnten aber auch Objekte, die in anderen Formaten (z.B.: VRML) beschrieben wurden, importiert werden. Für das bestehende Prinzip der Szenengraphverwaltung würde dies keinen Unterschied machen. Ausserdem könnten weitere Komponenteneigenschaften mit eingebettet werden. Dies könnten zum Beispiel eventgesteuerte Bewegungen der Komponenten oder veränderliche Materialeigenschaften sein. Last but not least besteht die Realität nicht nur aus Farben und Formen. Daher sollten auch Klänge in die virtuelle Welt aufgenommen werden.

Literatur

[BD01a] Peter Blanchebarbe, Stephan Diehl, *A Framework for Component Based Model Acquisition and Presentation Using Java3D, Web3D 2001, Fifth Symposium, Paderborn, Germany, 2001, pp. 117-125*, ACM, 2001

[BD01b] Peter Blanchebarbe, Stephan Diehl, *GenAu-3D- A Tool to Acquire and Present 3D-Components*, Shaker, erscheint noch

[Bou99] Dennis J. Bouvier, *Java 3D API Tutorial Updated*, Sun Microsystems, 1999

[BP98] Kirk Brown, Daniel Petersen, *Ready-to-Run Java 3D*, John Wiley & Sons, Inc, 1998

[BPS00] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, *Extensible Markup Language (XML) 1.0, Second Edition (http://www.w3.org/TR/REC-xml)*, 2000

[FD94] James D. Foley, Andries van Dam et al., *Grundlagen der Computergraphik, Einführung, Konzepte, Methoden*, Addison-Wesley, 1994

[GM96] James Gosling, Henry McGilton, *The Java Language Environment, A White Paper (http://java.sun.com/docs/white/langenv/index.html)*, 1996

[GP99] Charles F. Goldfarb, Paul Prescod, *XML Handbuch*, Prentice Hall, 1999

[Har99] Elliotte Rusty Harold, *XML Bible*, IDG Books Worldwide, 1999

[KRS98] Jörg Kloss, Robert Rockwell, Kornel Szabo et al., *VRML 97, Der neue standard für interaktive 3D-Welten im World Wide Web*, Addison-Wesley, 1998

[SRD98] Henry Sowizral, Kevin Rushforth, Michael Deering, *The Java 3D API Specification*, Addison-Wesley, 1998

[Wat93] Alan Watt, *3D Computer Graphics*, Addison-Wesley, 1993