3D-Grafik und -Animation: Java3D

Hauptseminar „Virtuelle Präsenz“
Wintersemester 2003/2004

Marco Jahn

Inhaltsverzeichnis

1 Einleitung

Dass Menschen über das Internet miteinander kommunizieren stellt kein neues Phänomen dar. Das Verschicken von Emails oder der Besuch von Chats sind alltäglich geworden, die Menschen sind mehr und mehr in der Virtualität präsent. Mittlerweile sind es nicht nur textbasierte Chaträume in denen sich, über real existierende räumliche Grenzen hinaus, Menschen treffen und miteinander unterhalten. Dreidimensionale Welten, teils der realen Welt nachgebildet, teils reine Phantasiekonstrukte, erlauben es dem Benutzer, sich frei in dieser zu bewegen. Zusätzlich zu einem Nickname, wie in Textchats üblich, wählt der Besucher eines 3D-Chats eine Figur, einen so genannten Avatar als seinen virtuellen Repräsentanten. Diesen steuert er durch die virtuelle Welt, erkundet diese, baut Häuser und kommuniziert mit anderen Avataren.
Doch wie entstehen solche dreidimensionalen Welten, wie sieht die technische Seite hinter den Avataren und ihrer virtuellen Umgebung aus? Diese Arbeit beschäftigt sich mit dem Thema 3D-Grafik für virtuelle Welten und Avatare. Mit Java3D soll ein Einblick in eine Erweiterung der Programmiersprache Java gegeben werden, die sich sehr gut für die Programmierung von dreidimensionalen Welten eignet.

2 Java3D Grundlagen

Eine erste Spezifikation von Java3D wurde erstmals auf der SIGGRAPH97 vorgestellt. Seitdem wurde die Technologie bis zur aktuellen Version 1.3.1 weiterentwickelt und ist mit dieser keineswegs abgeschlossen. Die Java3D API ist als optionales Paket zum J2SDK erhältlich. Im Folgenden sollen die grundlegenden Eigenschaften der Java3D API erläutert werden.

2.1 Die Java3D API

Die Java3D API stellt eine Menge von Klassen zur Verfügung, die es ermöglichen, dreidimensionale Konstrukte zu erzeugen, zu manipulieren und rendern zu lassen und in Applikationen oder Applets einzufügen. Man spricht von einer High-Level-API, da die Programmierung von 3D-Welten auf einem relativ hohen Abstraktionsgrad stattfindet (s. Abbildung 1)

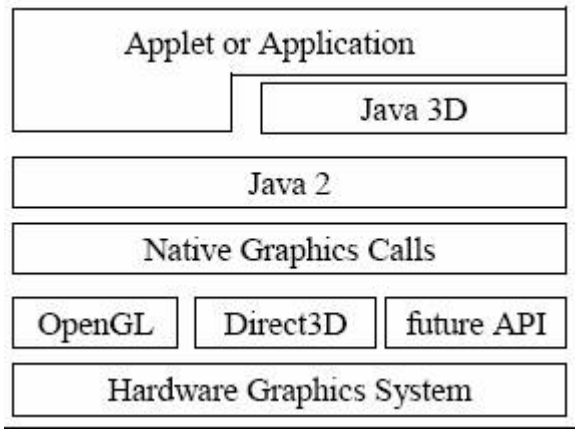

Abbildung 1: Schichtendarstellung

Diese Grafik soll das Abstraktionslevel von Java3D verdeutlichen. Wie zu erkennen ist, bietet die Java3D API keinen direkten Zugriff auf die Grafik-Hardware sondern ermöglicht das Erstellen von komplexen dreidimensionalen Inhalten auf einer höheren Ebene. Der Anwender muss sich somit nicht um Details des Renderingablaufs kümmern, hat aber auch dementsprechend wenig Einfluss auf diesen.
Die Java3D Bibliothek lässt sich im Wesentlichen in zwei große Packages gliedern. Zum einen die Kernklassen, die durch das *javax.media.j3d* Package bereitgestellt werden. Dieses Paket liefert bereits alle benötigten Klassen und Methoden, um anspruchsvolle dreidimensionale Inhalte zu erstellen.
Der zweite große Teil sind die *com.sun.j3d.utils.** Packages, welche die Kernklassen um weitere nützliche Funktionen erweitern. Die Utility Klassen können in vier Kategorien unterteilt werden: Content loaders, scene graph construction aids, geometry classes und convenience classes. Die Aufgabe der Utility Klassen besteht also darin, dem Programmierer die Arbeit zu erleichtern, indem Funktionen angeboten werden, die den Programmieraufwand deutlich reduzieren.
Zusätzlich zu den Kern- und Utility-Klassen greift jedes Java3D-Programm auch auf Klassen der Packages *java.a*wt und *javax.vecmath* zurück. Das Abstract Windowing Toolkit (awt) wird benötigt, um das Fenster, in welches das Ergebnis gerendert wird, zu erzeugen. J*avax.vecmath* stellt Klassen für den Umgang mit mathematischen Objekten, wie Vektoren und Matrizen zur Verfügung.

2.2 Aufbau eines Java3D Universums

Eine Java3D Applikation wird als `VirtualUniverse` bezeichnet. Ein solches `VirtualUniverse` ist eine, aus dreidimensionalen Objekten erzeugte, virtuelle Welt. Um ein `VirtualUniverse` zu erschaffen, muss ein Szenengraph aufgestellt werden, welcher Geometrie, Positionierung, Orientierung, Belichtung etc. von grafischen und auditiven Objekten definiert. Jede Java3D Applikation lässt sich als ein solcher Szenengraph darstellen. Folgende Grafik veranschaulicht das Konzept des Szenengraphen und zeigt seine wesentlichen Bestandteile, die in den folgenden Kapiteln näher erläutert werden.

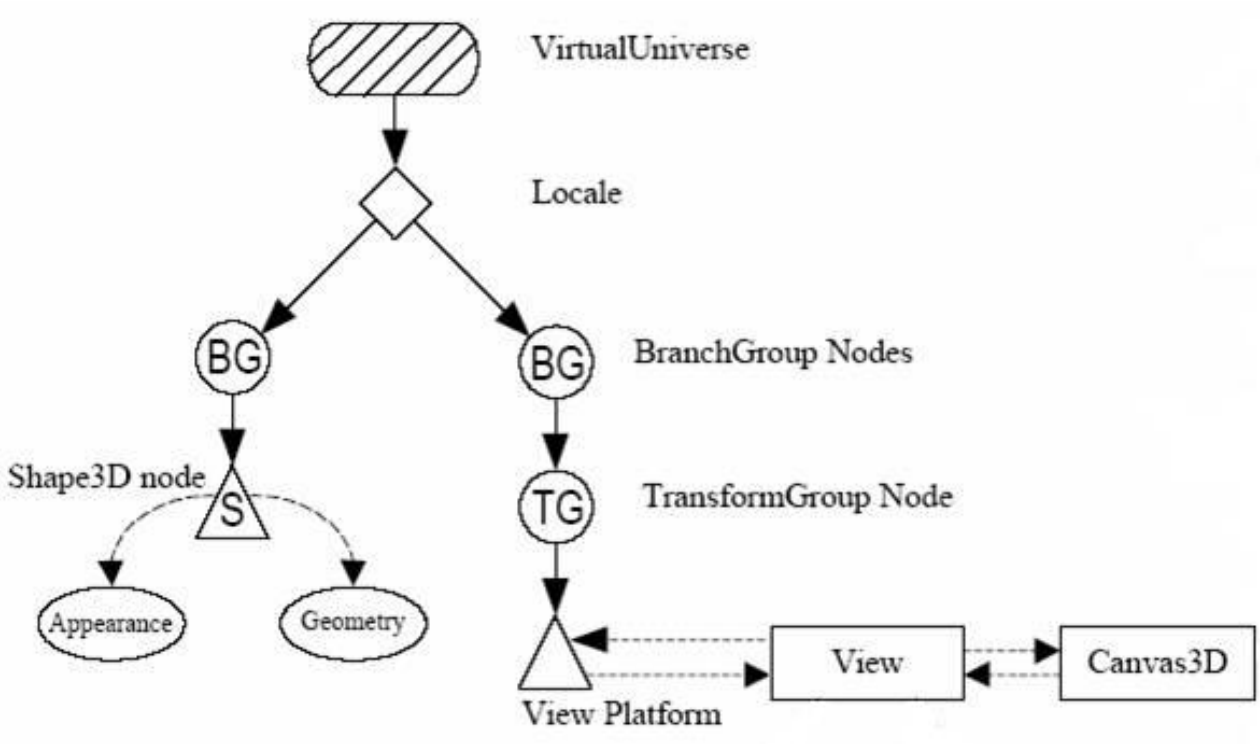

Abbildung 2: Der Szenengraph (SceneGraph)

Die Komponenten des Szenengraphen:

`VirtualUniverse`:

Ein `VirtualUniverse` Objekt definiert immer die Wurzel eines Szenengraphen.

`Locale`:

Der `Locale` Knoten definiert einen Orientierungspunkt in einem virtuellen Universum und positioniert seine Unterbäume in diesem Universum. Der Kind-Knoten eines `VirtualUniverse`-Knotens ist immer eine `Locale`.

`GroupNode`:

Ein Group Knoten definiert die Wurzel eines Telbaumes. Dies bedeutet, dass ein Group Knoten beliebig viele Kind-Knoten besitzen kann, jedoch nur einen Eltern Knoten. Dieser kann entweder ein `Locale`- oder wieder ein `Group`-Knoten sein. In Abbildung 2 als Kreise dargestellt.

`LeafNode`:

Blattknoten. Diese Knoten haben keine Kinder, können aber `NodeComponent`-Knoten referenzieren. In der obigen Grafik als Dreiecke dargestellt.

`NodeComponent`:

`NodeComponents` definieren Geometrie, Erscheinung, Textur und Materialeigenschaften von dreidimensionalen Objekten (`Shape3D` Leaf Nodes) und sind in Abbildung 2 als Ovale gekennzeichnet.

Wie in Abbildung 2 zu sehen ist, teilt sich der Szenengraph unterhalb des `Locale`-Knotens in zwei Teilgraphen. Der linke Teilbaum wird `ContentBranch` genannt und ist für die Definition der dreidimensionalen Inhalte zuständig. Der rechte Teilbaum ist die `ViewBranch` und bestimmt die Sicht auf das Universum. Diese beiden Bäume werden in den folgenden Kapiteln noch näher beschrieben.

Aufbau des Szenengraphen:

Ein Graph lässt sich definieren als Struktur von Knoten und Kanten. Knoten stellen Datenelemente dar, Kanten sind Beziehungen zwischen Knoten. Die Knoten in einem Java3D Szenengraphen sind Instanzen von Objekten. Die Kanten zwischen diesen Knoten stellen zwei Arten von Beziehungen zwischen Knoten dar. Zum einen die Eltern-Kind-Beziehung zwischen Knoten. Ein `GroupNode` kann eine beliebige Anzahl von Kind-Knoten haben, darf jedoch nur einen einzigen Eltern-Knoten besitzen. Ein `LeafNode` hat einen Eltern-Knoten und keine Kind-Knoten.

Die zweite mögliche Beziehung zwischen Instanzen ist die Referenz. Eine Referenz setzt einen Knoten mit einer `NodeComponent` in Beziehung. `NodeComponent` Objekte definieren die Geometrie- und Erscheinungsattribute eines darstellbaren Objektes.

Durch die Vorgabe, dass jeder Knoten eines Java3D-Szenengraphen nur einen Eltern-Knoten haben darf, wird gewährleistet, dass eine Baumstruktur entsteht, in der keine Schleifen vorkommen, d.h. es existiert nur ein Weg vom Wurzelobjekt bis zu jedem Knoten. Dieser Weg von der Wurzel bis zu einem bestimmten Blattknoten wird scene-graph-path genannt. Jeder scene-graph-path spezifiert genaue Informationen (Position, Orientierung, Größe etc.)

über den entsprechenden Blattknoten. Mit Hilfe der durch die scene-graph-paths bereitgestellten Informationen, ist es dem Renderer möglich, eine optimale Renderingreihenfolge zu ermitteln.
Des Weiteren kann die Repräsentation des `VirtualUniverse` als Szenengraph den Programmierer beim Entwurf des Universums als Designtool unterstützen. Das komplette Universum kann zunächst als Szenengraph spezifiziert und auf dieser Grundlage implementiert werden.

3 Die Content Branch

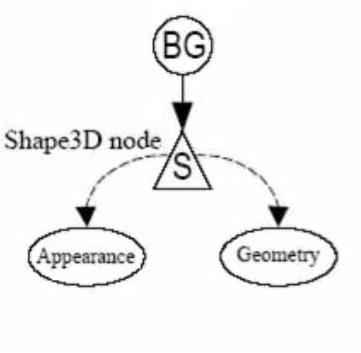

Abbildung 3: Beispiel einer ContentBranch

Der Teil des Szenengraph, der für die Definition und Darstellung der dreidimensionalen Inhalte des Universums zuständig ist, wird `ContentBranch` gennant. In Abblidung 2 entspricht der linke Teilbaum unterhalb des `Locale` Objekts der `ContentBranch`.
Zunächst ein kleines Beispiel, das zeigt, wie mit einfachsten mitteln ein kleines `VirtualUniverse` erschaffen werden kann:

3.1 Einführendes Beispiel

Es soll ein einfacher bunter Würfel dargestellt werden, ein kleines Programm, welches als das „HelloWorld“ von Java3D bezeichnet werden kann und deshalb auch oft als „HelloCube“ bezeichnet wird.

```
1. import javax.media.j3d.*;
2. import javax.vecmath.*;
3. import java.awt.*;
4. import com.sun.j3d.utils.geometry.*;
5. import com.sun.j3d.utils.universe.*;
6.
```

```
7. public class HelloCube {
8.
9.    public static void main(String[] args) {
10.         Frame frame = new Frame();
11.         frame.setSize(640, 480);
12.         frame.setLayout(new BorderLayout());
13.
14.         Canvas3D canvas = new Canvas3D(null);
15.         frame.add("Center", canvas);
16.
17.         SimpleUniverse univ = new SimpleUniverse(canvas);
18.         univ.getViewingPlatform().setNominalViewingTransform();
19.
20.         BranchGroup scene = createSceneGraph();
21.         univ.addBranchGraph(scene);
22.
23.         frame.show();
24.   }
25.
26.   public static BranchGroup createSceneGraph() {
27.         // BranchGroup erzeugen
28.         BranchGroup branch = new BranchGroup();
29.
30.         // ColorCube erzeugen
31.         ColorCube my_Cube = new ColorCube(0.4);
32.
33.         // ColorCube als Kind der BranchGroup hinzufügen
34.         branch.addChild(my_Cube);
35.
36.         return branch;
37.   }
38. }
```

Zunächst werden die benötigten Packages importiert (Zeile 1-5). Dann wird in den Zeilen 10 - 12 der Frame definiert, der als Fenster für das `VirtualUniverse` dienen wird. Diesem Fenster wird die Größe 640*480 und ein `BoderLayout` zugewiesen.
`Canvas3D` ist eine Klasse, die einen Rahmen definiert, in welchem 3D Objekte gerendert und dargestellt werden können. Dieser `Canvas3D` wird dem Frame an die Position `Center` des `BorderLayouts` übergeben (Zeile 14 - 15).

In Zeile 17 wird ein Objekt vom Typ `SimpleUniverse` instanziiert. Diese Klasse wird durch das Paket *com.sun.j3d.utils.universe* bereitgestellt. Die Besonderheit eines `SimpleUniverse` besteht darin, dass die komplette `ViewBranch` bereits definiert ist. Der Programmierer muss sich beim Verwenden eines `SimpleUniverse` also keine Gedanken über die `ViewBranch` machen sondern bekommt diese bereits als Default mitgeliefert.
Durch den Aufruf der Methode `setNominalViewingTransform` in Zeile 18 wird die `ViewPlatform` in Richtung der z-Achse verschoben, so dass der Benutzer auf den Ursprung der Szene blickt (Dazu mehr im Kapitel 4: `ViewBranch`).
Als nächstes wird ein `BranchGroup` Objekt definiert, welches in der Methode `createSceneGraph()` erzeugt wird. Dieses `BranchGroup` Objekt definiert einen Teilgraphen zur Darstellung des Würfels. Mittels der Methode `addBranchGraph(BranchGroup)` wird dieser Graph als Kind dem `SimpleUniverse` übergeben (Zeilen 20 - 21).
Durch den Aufruf der Methode `show()` in Zeile 23 wird gewährleistet, dass der Frame die im Szenengraph definierten Objekte anzeigt.
In den Zeilen 26 - 38 wird nun die Methode `createSceneGraph()` definiert, welche einen einfachen Teilgraphen erzeugt. Die Wurzel eines solchen Teilgraphen ist ein Knoten vom Typ `Group`, in diesem Fall ein `BranchGroup` Knoten (Auf die verschiedenen `Group`-Typen wird im weiteren Verlauf noch eingegangen). Der eigentliche Inhalt unseres Universums, also das 3D-Objekt, welches im Frame dargestellt wird, ist der `ColorCube`. Dieser wird in Zeile 31 mit der Größe 0.4 instanziiert. Der letzte Schritt in der Erstellung des Teilgraphen besteht darin, eine Eltern-Kind-Beziehung zwischen der zuvor erzeugten `BranchGroup` und dem `ColorCube` herzustellen. Dies geschieht mittels des Aufrufes der Methode `Group.addChild(Shape3D)` in Zeile 34.
Abbildung 4 zeigt das Ergebnis dieses einfachen Java3D Programms.

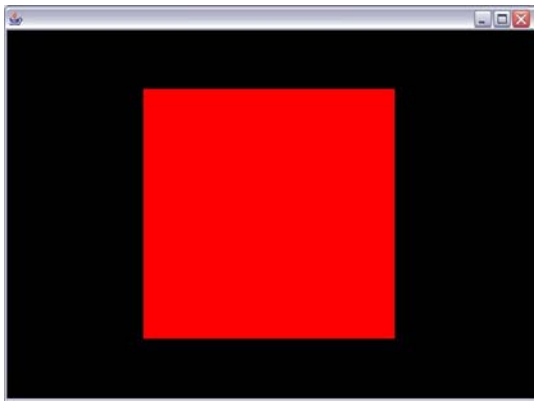

Abbildung 4: "HelloWorldCube"

Dass dieser Würfel hier als Quadrate erscheint, liegt daran, dass der Betrachter entlang der z-Achse auf den Ursprung der Szene schaut.

3.2 Group Knoten

Wie zuvor gesehen wird die Hierarchie eines Szenengraphen mithilfe von `GroupNodes` erstellt. Im Beispielprogramm „HelloCube“ wurde bereits ein Typ einer `Group` vorgestellt: Die `BranchGroup`. Java3D stellt eine Reihe von `Group`-Knoten zur Verfügung. Diese Knoten dienen zum einen der Gliederung des Szenegraphen in Teilgraphen, zum anderen stellen sie aber auch weitere Funktionalitäten zur Verfügung. Die verschiedenen Arten von Group-Knoten werden im Folgenden vorgestellt.

Die Superklasse für alle Gruppen ist die `Group` Klasse. Diese stellt Methoden zum hinzufügen und entfernen von Kindern (welche natürlich auch wieder `Group`-Knoten sein können) zur Verfügung.

`BranchGroup`:

Eine `BranchGroup` definiert einen Teilbaum im Szenegraphen, wie bereits im Beispiel „HelloCube“ gesehen. Zusätzlich zur `Group` Klasse verfügt die `BranchGroup` über Methoden um den Renderingprozess zu beschleunigen.

`OrderedGroup`:

Ein `OrderedGroup` Knoten garantiert, dass alle Kinder dieser Gruppe in der, ihrer Hierarchie entsprechenden Reihenfolge, gerendert werden, was beispielsweise bei einer `BranchGroup` nicht der Fall sein muss.

`DecalGroup`:

Die `DecalGroup` ist abgeleitet von `OrderedGroup` und wird hauptsächlich zum rendern von Text verwendet.

`TransformGroup`:

Um Objekte im Raum zu positionieren gibt es die `TransformGroup`-Knoten. Diese `Group` erzeugt ein neues Koordinatensystem relativ zu dem, welches durch ihren Elternknoten gegeben ist. Somit ist es möglich Objekte zu positionieren, rotieren und skalieren. Alle Kind-Knoten einer `TransformGroup` befinden sich in diesem Koordinatensystem. Modifizierungen, die an einer solchen Gruppe vorgenommen werden, wirken sich somit auf ihren gesamten Unterbaum aus. Natürlich können sich in der Hierarchie unterhalb einer `TransformGroup` weitere `TransformGroups` befinden. Welche Transformation eine solche Gruppe ausführen soll wird durch ein Objekt der Klasse `Transform3D` definiert. Diese Klasse stellt eine Vielzahl von Hilfsfunktionen zur Verfügung, welche den Umgang mit den möglichen Transformationsarten erheblich erleichtern. Somit ist es auch möglich, ohne tiefgreifende Kenntnisse der mathematischen Hintergründe, Transformationen durchzuführen. Selbstverständlich können aber auch eigene Transformationsmatrizen oder Quaternionen verarbeitet werden. Um die gewünschte Transformation wirksam werden zu lassen ist es dann nur noch erforderlich, das `Transform3D` Objekt in Beziehung zu einer `TransformGroup` zu stellen. Folgender kleiner Codeausschnitt zeigt die Anwendung einer `TransformGroup`.

```
Transform3D rotate = new Transform3D( );
rotate.rotY( Math.PI / 4.0d );
TransformGroup trans = new TansformGroup(rotate);
```

Um die hier definierte Rotation (Drehung um die y-Achse um 45°) wirksam werden zu lassen, muss die `TransformGroup` nur noch in den Szenengraph eingefügt werden.

3.3 Shape3D Knoten

Am Beispiel des „HelloCube“ wurde gezeigt, wie man mit wenigen Zeilen Code bereits ein primitives 3D-Objekt erstellt. In Abbildung 2 ist als Blattknoten in der `ContentBranch` ein `Shape3D` Knoten zu sehen. Der in „HelloCube“ erzeugte `ColorCube` ist nur ein Spezialfall eines `Shape3D` Objektes. Möchte man komplexere Inhalte als beispielsweise diesen Würfel erzeugen, bedient man sich der `Shape3D` Klasse. An dieser Stelle sei noch einmal darauf hingewiesen, dass `Shape3D` Knoten immer Blattknoten sein müssen um einen validen Szenengraphen zu gewährleisten.

Ebenfalls ist in Abbildung 2 zu sehen, dass der `Shape3D` Knoten zwei Referenzen auf `NodeComponent` Objekte hält. Die `Geometry`, welche Form und Struktur des Objektes

beschreibt und `Appearance`, welche für die Kolorierung und das Shading zuständig ist. Die Klasse `Shape3D` stellt zwei Methoden zur Verfügung, um die beiden Eigenschaften festzulegen:

```
void setGeometry(Geometry geometry)
void setAppearance(Appearance appearance)
```

`Geometry`:

Die Geometrie eines `Shape3D` wird definiert durch verschiedene, von `Geometry` abgeleitete Klassen. Diese ermöglichen das Beschreiben von Objekten durch GeometryArray, Text3D, Raster und CompressedGeometry.

Durch Geometrie Arrays ist es möglich, Objekte als Folge von Punkten (Vertices) im dreidimensionalen Koordinatensystem zu beschreiben. Ein Dreieck beispielsweise, kann so als Verbindung, dreier, aufeinanderfolgender Vertices verstanden werden

Neben Dreiecken können so auch (unverbundene) Vertices, Linien, Vierecke und Polygone beschrieben werden.

`Appearance`:

Die Klasse `Appearance` legt fest wie Java3D die Objektgeometrie rendert. Es können Attribute wie Farbe, Alpha, Shading, Linienstärke, Texturierung etc. definiert werden.

Um z.B. die Farbe eines `Shape3D` Objektes festzulegen sind folgende Anweisungen notwendig:

```
Appearance app = new Appearance( );
ColoringAttributes ca = new ColoringAttributes( );
ca.setColor( 1.0f, 1.0f, 0.0f );
```

Das `Appearance` Objekt muss nun noch mittels der Methode `Shape3D.setAppearance` in Beziehung zu einem Objekt gesetzt werden.

3.4 Content Loader

Schaut man sich aktuelle dreidimensionale Welten an, so wird man schnell zu der Überzeugung gelangen, dass das Erzeugen komplexer Objekte, wie z.B. Avatare, mittels der hier vorgestellten Geometriebeschreibungen einen erheblichen Zeitaufwand beanspruchen würde. Als Lösung für dieses Problem stellt Java so genannte Loader Classes zur Verfügung. Diese Klassen ermöglichen das Importieren von bereits fertigen Modellen, Szenen und Animationen. Java3D bietet Unterstützung für gängige 3D-Modeling-Tools wie Maya oder 3D-Studio. Aber auch VRML Objekte können in den Java3D Szenengraph importiert werden.

4 Behaviors

Mit den bisher vorgestellten Konzepten von Java3D ist es möglich komplexe dreidimensionale Inhalte zu erzeugen. Um ein virtuelle Universum mit Leben zu füllen ist es aber unabdingbar, dass sich Objekte bewegen und verändern und dass der Benutzer mit der virtuellen Welt interagieren kann. Java3D stellt die Klasse `Behavior` zur Verfügung, die es ermöglicht, Knoten des virtuellen Universums während der Laufzeit zu verändern um Animationen und Interaktionen ausführen zu können. In der Java3D Klassenhierarchie ist `Behavior` abgeleitet von `LeafNode`, d.h. ein `Behavior` darf keine Kind Knoten besitzen, sondern wird in eine Refernz-Beziehung zu einem Knoten gesetzt. Wird ein `Behavior` aktiv, kann es jede mögliche Funktion ausführen, von Berechnungen bis hin zu Veränderungen im Szenengraph. Hier stellt sich natürlich die Frage, wann ein `Behavior` aktiv werden soll. Als Lösung dienen `SchedulingBounds` und `WakeUpCriteria` welche vom Programmierer definiert werden.

`SchedulingBounds`:

Die `SchedulingBounds` beschreiben den Aktivierungsradius eines `Behaviors` welcher definiert ist durch ein beliebiges polygonales Objekt. Überschneidet sich nun der Sichtweiteradius des Betrachters mit dem Aktivierungsradius eines `Behaviors` wird es aktiviert. Somit wird gewährleistet, dass z.B. eine rechenintensive Animation erst dann ausgeführt wird, wenn sie sich in Sichtweite des Benutzers befindet, was die Performance einer Applikation erheblich steigert.

`WakupCriteria`:

Zusätzlich zu den `SchedulingBounds` lassen sich so genannte `WakupCriteria` festlegen. Ein `Behavior` wird dann erst aktiv, wenn diese Kriterien erfüllt sind, auch wenn sich der Betrachter bereits innerhalb der `SchedulingBounds` befindet. Beispiele für diese Kriterien sind: Eine bestimmte Anzahl Frames oder Millisekunden sind verstrichen, ein AWT Event wurde ausgelöst, zwei Shapes sind miteinander kollidiert oder eine Transformation innerhalb einer `TransformGroup` hat sich verändert. Diese Kriterien können beliebig mit AND/OR Operatoren verknüpft werden.

`InterpolatorBehaviors`:

Viele einfachere `Behaviors` lassen sich mithilfe der Klasse Interpolator darstellen indem ein Paramter wie z.B. Transformationskoordinaten oder Farbwerte über die Zeit verändert werden. Der Ablauf (Start, Ende, Geschwindigkeit) wird über ein Objekt der Klasse `Alpha` gesteuert. Dieses Objekt liefert einen Wert zwischen 0.0 und 1.0 abhängig von der

verstrichenen Zeit und den übergebenen Parametern. Zeichnet man den Alpha-Wert über die Zeit, so ergibt sich die Form der Alpha-Welle.

Beispiel für ein `InterpolatorBehavior`:

Um den „HelloCube“ um seine Achse rotieren zu lassen sind nur einige kleine Erweiterungen in der Methode `createSceneGraph` nötig. Die `main`-Methode kann unverändert bleiben. Die entscheidenden Veränderungen am Code sind blau dargestellt.

```
01.   private static BranchGroup createSceneGraph() {
02.         // BranchGroup erzeugen
03.         BranchGroup branch = new BranchGroup();
05.
06.         // TransformGroup erzeugen und Veränderbar machen
07.         TransformGroup trans = new TransformGroup();
08.         trans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
09.
10.         // ColorCube erzeugen
11.         ColorCube my_Cube = new ColorCube(0.4);
12.
13.
14.         // Behavior erzeugen, um Würfel drehen zu lassen
15.         Alpha spinAlpha = new Alpha(-1, 4000);
16.         RotationInterpolator spinner =
17.               new RotationInterpolator(spinAlpha, trans);
18.         spinner.setSchedulingBounds(
19.               new BoundingSphere(new Point3d(), 1000.0));
20.
21.         // Behavior als Kind der TransformGroup setzen
22.         trans.addChild(spinner);
23.         // ColorCube als Kind der TransformGroup setzen
24.         trans.addChild(my_Cube);
25.         // TransformGroup als Kind der BranchGroup setzen
26.         branch.addChild(trans);
27.
28.         return branch;
29.   }
```

Um die Rotation des Würfels zu ermöglichen, müssen die Rotationsattribute der vorgeschalteten `TransformGroup` zur Laufzeit veränderbar sein. Diese Eigenschaft der `TransformGroup` muss mittels der Methode

`setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE)` manuell gesetzt werden (Zeile 08).

Das eigentliche `Behavior` wird in den Zeilen 14 - 19 erzeugt. Zunächst wird der zugehörige `Alpha`-Wert gesetzt (Zeile 15). In diesem Fall -1, da sich der Würfel unendlich lange drehen soll. Der übergebene Wert 4000 bedeutet, dass sich der Würfel in 4000ms einmal um seine Achse dreht. Als nächstes wird der `RotationInterpolator` instanziiert. Wie in Zeile 17 zu sehen ist, erhält dieser als Übergabeparameter das `Alpha` Objekt und die `TransformGroup` auf die das `Behavior` angewendet werden soll. Die `SchedulingBounds` des Behaviors werden in den Zeilen 18 und 19 gesetzt. Wenn dann noch die Knoten in der richtigen Reihenfolge zueinander in Beziehung gesetzt werden (Zeilen 21 – 26) erhält man einen rotierenden, farbigen Würfel.

5 Die View Branch

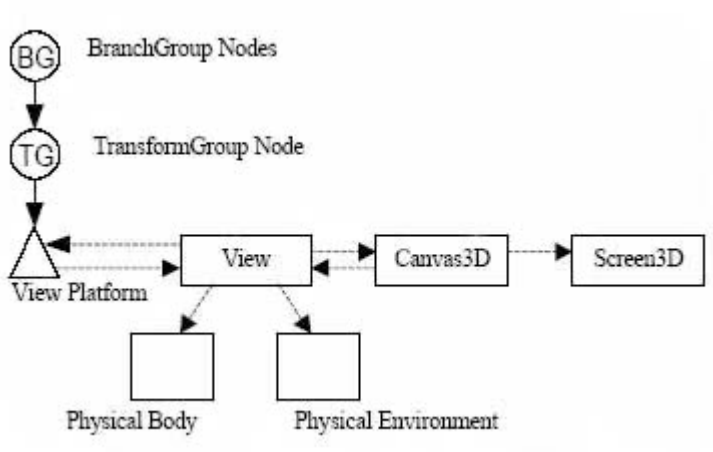

Abbildung 5: Die ViewBranch

Während sich die vorigen Kapitel mit dem Erzeugen von Objekten und deren Manipulation beschäftigten, widmet sich dieses Kapitel dem zweiten großen Teilbaum des Szenengraphen, der `ViewBranch`. Abbildung 2 zeigt eine schematische Darstellung der `ViewBranch` im rechten Teilbaum unterhalb des `Local` Knotens. Dieser Teil einer Java3D Applikation definiert die Sicht auf das virtuelle Universum, kontrolliert also die Position und Orientierung des Benutzers innerhalb der virtuellen Welt. Die Besonderheit des Java3D View Modells besteht vor allem darin, dass es Unterstützung für spezielle Bildschirmkonfigurationen, wie Head-Mounted-Displays oder Multi-Screen-Portale, und Head-Tracking bietet. Hierbei unterscheidet Java3D zwischen der physischen und der virtuellen Welt des Benutzers. Die physische Welt beschreibt die Position und Orientierung des Benutzers zu seinem Bildschirm

oder Bildschirmen. Die Aufgabe des View Modells besteht nun darin diese Daten aus der realen Welt auf die virtuelle Sicht zu übertragen.
Diese Daten der physischen Welt werden in der `ViewBranch` repräsentiert durch die Klassen `PhysicalBody` und `PhysicalEnvironment`. Für Applikationen die sich keiner speziellen Hardware bedienen, also nur einen einzelnen Monitor als Ausgabegerät benutzen, spielen diese Unterscheidungen keine Rolle.
Die eigentliche Sicht auf das Universum wird durch das `View` Objekt definiert. Diese `View` kann man sich vorstellen als den Benutzer der in die Welt hineinschaut und auf einer `ViewPlatform` steht. Der Benutzer steht also auf einer Art fliegendem Teppich, welcher ihn durch das virtuelle Universum trägt. Um nun eine Bewegung zu ermöglichen müssen Position und Ortientierung der `ViewPlatform` verändert werden. Wie Abbildung 5 zeigt, ist die `ViewPlatform` ein Kind Knoten einer `TransformGroup`. Wie im Kapitel `ContentBranch` gesehen dienen `TransformGroups` dazu Objekte im Raum zu positionieren; um den Benutzer zu bewegen wird also nicht die `ViewPlatform` direkt manipuliert, sondern die ihr vorgeschaltete `TransformGroup`. Zur Navigation durch das Universum, müsste man diese `TransformGroup` nun in Beziehung zu einem `Behavior` setzen, welches auf `MouseMotionEvents` reagiert, und die `TransformGroup` entsprechend dieser Daten aktualisiert.
Die Klasse `Canvas3D`, welche bereits aus dem Beispielprogramm „HelloCube“ bekannt ist, ist ein Fenster, in welchem dreidimensionale Objekte dargestellt werden können. Das `Screen3D` Objekt beschreibt die physikalischen Bildschirmeigenschaften.

6 Weitere Möglichkeiten von Java3D

6.1 Texturierung von Objekten

Um optisch anspruchsvolle Welten zu erzeugen, ist es nicht zufrieden stellend nur die Farbattribute von Objekten zu beeinflussen. Texture Mapping bietet die Möglichkeit Bitmap Bilder auf Objekte abzubilden. Eine Mauer z.B. lässt sich so einfach darstellen ohne jeden einzelnen Stein modellieren zu müssen. Texturen lassen sich mithilfe von Texture Loadern einfach aus beliebigen Bildern erzeugen und der `Appearance` eines Objektes zuweisen.

6.2 Beleuchtung

Zur Beleuchtung einer Szene bietet Java3D verschiedene Lichtquellen an: Ambient, Directional, Point und Spot. Alle diese Lichtquellen sind von der Superklasse `Light` abgeleitet und teilen sich die Attribute on/off Status, Farbe und Begrenzungsvolumen (`BoundingVolume`).

6.3 Hintergründe

Mithilfe eines `Background`-Knoten kann ein Hintergrund in eine Szene eingefügt werden. Möglich sind hier einfache farbige Hintergründe, Bilder und Geometrie. Ähnlich wie `Behaviors` sind auch `Backgrounds` mit einem `BoundingVolume` versehen, so dass der Hintergrund erst dann sichtbar wird wenn der Programmierer es erwünscht.

Des Weiteren bietet Java3D noch die Möglichkeiten Nebel, Text bzw. 3D-Text und Sound in ein Universum zu integrieren, auf deren Behandlung aber in dieser Arbeit nicht näher eingegangen werden soll.

7 Zusammenfassung

Wie die in den vorangegangenen Kapiteln vorgestellten Konzepte zeigen, bietet Java3D vielfältigste Möglichkeiten zur Erstellung virtueller Welten. Auch komplexe dreidimensionale Objekte lassen sich mithilfe der `ContentLoader` leicht in einen Szenengraphen einfügen. Da eine Vielzahl von Hilfsklassen angeboten werden, benötigt der Programmierer keine herausragenden Kenntnisse auf dem Gebiet der Computergrafik und der mathematischen Hintergründe. Für erfahrene Java Programmierer dürfte der Einstieg in Java3D nicht sehr beschwerlich ausfallen, da sich Java3D konsequent an die Java Programmierkonventionen hält. Einzig die Darstellung der virtuellen Welt als Szenengraph stellt ein neues Konzept dar. Trotzdem sei angemerkt, dass Java3D, bzw. Java, eine höhere Programmiersprache ist, und somit viel komplexer ist als beispielsweise Beschreibungssprachen wie VRML. Somit stellt sich die Frage, für welche Art von Projekten Java3D eingesetzt werden könnte. Aufgrund des hohen Maßes an Funktionalität bietet sich Java3D wohl vor allem für größere Applikationen an. Dem Programmierer sind z.B. durch das Konzept der `Behaviors` keine Grenzen im Umgang mit Benutzerinteraktionen gesetzt. Und auch den Anforderungen, die durch Hardware wie Head-Tracker oder 3D-Brillen gestellt werden, ist Java3D gewachsen, da es den kompletten Bereich der Sichtdefinition in die `ViewBranch` kapselt und Unterstützung für solche Hardware bietet.

Da sich die gesamte Java-Technologie in ständiger Weiterentwicklung befindet, ist davon auszugehen, dass es auch im Bereich Java3D noch viele Neuerungen und Verbesserungen geben wird.
Ein Beispiel für ein virtuelles Universum, ein 3D-Chat, in dem sich die Benutzer mit Avataren in einer dreidimensionalen Welt bewegen ist www.3d-chat.com

8 Referenzen

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

http://www.sdsc.edu/~nadeau/nadeau_courses.html

http://www.java3d.org/

http://www.j3d.org/