

Michael Pfeiffer

Java 3D 1.3

**für Einsteiger und Fortgeschrittene
inclusive erster Ausblicke auf Java 3D 1.3.2**

Version 1.1

Dieses Dokument ist unter den Bedingungen der Creative Commons, Namensnennung-Nicht-kommerziell-Weitergabe unter gleichen Bedingungen 3.0 Deutschland (CC BY-NC-SA 3.0) lizenziert. Es darf somit von jedem frei benutzt werden, so lange damit kein Profit erzielt oder in sonstiger Weise Gegenleistungen dafür verlangt werden. Veränderungen am Dokument sind zulässig. In diesem Fall würde ich mich über Feedback freuen.

Alle Informationen in diesem Dokument werden so angeboten, wie sie sind. Es werden keinerlei Garantien oder Gewährleistungen für deren Richtigkeit und Tauglichkeit übernommen. Eine Benutzung erfolgt also in jedem Fall vollständig auf eigene Gefahr.

Die zugehörigen Beispielsourcen sind freie Software und unterliegen den Bedingungen der GNU General Public License so wie sie von der Free Software Foundation veröffentlicht wurde. Das bezieht sich auf Version 2 dieser Lizenz oder optional auf jede weitere, spätere Version.

Die vorliegende Version ist hiermit zusammen mit den zugehörigen Sourcen zum Review freigegeben. Kommentare, Hinweise, Verbesserungsvorschläge, Bugreports etc. sind erwünscht und möglichst unter Angabe der betreffenden Zeilennummer an virtual_worlds@gmx.de zu senden. Auch sollten die Rechtschreibfehler jetzt so selten sein, dass es sich lohnt, auf die noch Verbliebenen hinzuweisen – bitte wieder unter Angabe der Zeilennummer.

Inhaltsverzeichnis

1 Vorwort.....	6
2 Einführung.....	8
2.1 Die Entwicklungsumgebung.....	8
2.1.1 Besonderheiten der Windows-Version.....	8
2.1.2 Installation und Konfiguration der Umgebung.....	9
2.1.2.1 Installation des Java 3D SDK unter Linux.....	9
2.1.2.2 Installation des Java 3D SDK unter Windows.....	10
2.1.2.3 Die Konfiguration des Borland JBuilder.....	10
2.1.2.4 Die Konfiguration von Eclipse.....	11
2.1.3 Java-3D-Applikationen und Java WebStart.....	12
3 Der Einstieg.....	13
3.1 Hallo Universum.....	13
3.2 Das erste 3D-Objekt.....	15
3.3 SceneGraph und Transformation.....	17
3.3.1 BranchGroup und TransformGroup.....	19
3.3.1.1 BranchGroup.....	19
3.3.1.2 TransformGroup und Transform3D.....	20
3.3.1.2.1 TransformGroup.....	20
3.3.1.2.2 Transform3D.....	21
3.3.1.3 Funktioniert es?.....	22
3.4 Daten-Sharing.....	22
3.4.1 Link.....	24
3.4.2 SharedGroup.....	25
4 Objekteigenschaften.....	27
4.1 Farben.....	27
4.1.1 AmbientLight und DirectionalLight.....	28
4.1.1.1 AmbientLight.....	29
4.1.1.2 DirectionalLight.....	29
4.1.2 Appearance.....	29
4.1.3 Material.....	30
4.2 Texturen.....	33
4.2.1 TextureLoader.....	34
4.2.2 TexCoordGeneration.....	36
4.2.3 TextureAttributes.....	39
4.2.4 Texture.....	41
4.3 Durchsichtiges.....	42
4.3.1 PolygonAttributes.....	44
4.3.2 Transparency.....	47
4.3.2.1 Transparenz und Z-Order.....	49
4.3.2.2 Die OrderedGroup.....	50
5 Licht nach Maß.....	53
5.1 BoundingSphere.....	55
5.2 BoundingBox.....	58
5.3 BoundingPolytope.....	59
5.4 PointLight.....	60
5.5 SpotLight.....	62
5.6 Die Basisklasse Light.....	64
6 Bewegungen.....	68

6.1 Capability.....	69
6.1.1 Capability Bits der TransformGroup.....	71
6.2 Der elegante Weg.....	73
6.2.1 Alpha.....	76
6.2.2 Interpolator.....	80
6.2.2.1 RotationInterpolator.....	81
6.2.2.2 Andere Interpolatoren.....	83
7 Beobachtungen.....	87
7.1 Die ViewPlatform.....	88
7.2 Navigation.....	89
7.3 Behavior.....	90
7.3.1 WakeupOnAWTEvent.....	92
7.3.2 Eine Übersicht über die WakeupCriteria.....	93
7.3.2.1 WakeupOnActivation.....	93
7.3.2.2 WakeupOnBehaviorPost.....	94
7.3.2.3 WakeupOnCollisionEntry, WakeupOnCollisionExit und WakeupOnCollisionMovement.....	94
7.3.2.4 WakeupOnElapsedFrames.....	95
7.3.2.5 WakeupOnElapsedTime.....	96
7.3.3 Kombinieren von WakeupCriteria.....	96
7.3.3.1 WakeupAnd.....	97
7.3.3.2 WakeupOr.....	97
7.3.3.3 WakeupAndOfOrs.....	97
7.3.3.4 WakeupOrOfAnds.....	98
7.3.4 MouseRotation Behavior.....	98
7.3.5 Weitere Mouse Behaviors.....	100
7.3.6 KeyNavigatorBehavior.....	100
8 Geometry und komplexe 3D-Objekte.....	102
8.1 Vertex und Polygon.....	102
8.2 Texturkoordinaten.....	103
8.3 Farben und Reflektionsverhalten.....	104
8.4 Shape3D – das Universalgenie.....	104
8.4.1 IndexedTriangleArray.....	108
8.4.2 Das Shape3D-Beispielprogramm.....	114
8.4.3 TriangleArray.....	117
8.4.4 IndexedTriangleStripArray und TriangleStripArray.....	118
8.4.5 IndexedTriangleFanArray und TriangleFanArray.....	119
8.4.6 IndexedQuadArray und QuadArray.....	119
8.4.7 Sonstige GeometryArrays.....	120
8.4.8 OrientedShape3D.....	121
8.4.8.1 Billboard.....	124
8.4.9 Normals.....	126
8.4.9.1 GeometryInfo.....	127
8.4.9.2 NormalGenerator.....	129
8.4.9.3 Stripifier.....	131
8.4.9.3.1 StripifierStats.....	132
9 Picking.....	134
9.1 Dreidimensionaler Text.....	134
9.1.1 FontExtrusion.....	135
9.1.2 Font3D.....	136
9.1.3 Text3D.....	138
9.1.4 PickCanvas.....	140

9.1.5 PickTool.....	143
9.1.5.1 PickResult.....	146
9.1.5.1.1 PickIntersection.....	148
9.1.5.2 PickShape.....	150
9.1.5.2.1 PickRay.....	151
9.1.5.2.2 PickSegment.....	152
9.1.5.2.3 PickCylinder.....	152
9.1.5.2.4 PickCone.....	154
9.1.5.2.5 PickPoint.....	156
9.1.5.3 Picking-Funktionalitäten der BranchGroup.....	156
9.1.5.4 Das Picking-Ergebnis SceneGraphPath.....	157
9.1.5.5 Picking-Funktionalitäten des Shape3D.....	160
9.2 Terrain Following und Collision Prevention.....	161
10 Umweltbedingungen.....	163
10.1 Hintergrund.....	163
10.1.1 Der Background – Farbe und Geometrie.....	164
10.1.2 Ein Sternenhimmel.....	166
10.1.2.1 PointAttributes.....	170
10.2 Nebel.....	171
10.2.1 ExponentialFog.....	174
10.2.2 LinearFog.....	176
11 Klangvolles – Sound in der virtuellen Welt.....	178
11.1 Die Verwendung der Basisklasse Sound.....	178
11.1.1 MediaContainer.....	178
11.1.2 Sound.....	181
11.1.3 BackgroundSound.....	186
11.1.4 PointSound.....	189
11.1.5 ConeSound.....	194
11.2 Klänge und Schall – mehr als nur Soundsamples.....	199
11.2.1 AuralAttributes.....	200
11.2.2 Soundscape.....	208
11.3 Sound unter Java 3D 1.3.2.....	209
12 Fortgeschrittene Techniken.....	210
12.1 Die Basisklasse SceneGraphObject.....	210
12.2 Die Basisklasse Node.....	213
12.3 Die Klasse Group.....	216
12.4 Die Switch-Group.....	218
12.5 AlternateAppearance.....	220
12.6 Front- und Backclipping.....	223
12.7 Das Canvas3D.....	225
12.7.1 Off-Screen Rendering.....	225
12.7.2 Stereoskopie.....	227
12.7.3 Sonstige Canvas3D-Methoden.....	228
12.7.4 Java 3D und Swing.....	229
12.8 LOD.....	231
12.8.1 DistanceLOD.....	233
12.9 Sky-Objekte für Panoramahintergründe.....	235
12.9.1 RenderingAttributes.....	236
12.9.2 Die Skyplane.....	240
12.9.3 Skydome und Skysphere.....	240
12.9.4 Die Skybox.....	241
13 Anhang A – Überblick über die mitgelieferten Sourcen.....	243

Abbildungsverzeichnis

Abbildung 1	Konfiguration der Klassenpfade beim JBuilder	11
Abbildung 2	Klassendefinitionen bei den Eclipse-Projekteinstellungen	12
Abbildung 3	Ein erster SceneGraph	20
Abbildung 4	Anordnung von Link-Nodes und SharedGroup	24
Abbildung 5	Darstellung eines Polygons von vorne und hinten	47
Abbildung 6	Erzeugung einer BoundingBox aus zwei Punkten	61
Abbildung 7	Der Öffnungswinkel beim SpotLight	65
Abbildung 8	Die Verwendung eines Interpolators	77
Abbildung 9	Vertices und Polygone	104
Abbildung 10	Der Aufbau eines Triangle-Strips	121
Abbildung 11	Der Aufbau eines Triangle-Fans	122
Abbildung 12	Terrain Following	165
Abbildung 13	Aufbau des SceneGraphen im PointSound-Beispielprogramm	195
Abbildung 14	Öffnungswinkel und AngularAttenuation beim ConeSound	200
Abbildung 15	Die AuralAttributes-Parameter	209
Abbildung 16	Front und Back Clipping	229

Tabellenverzeichnis

Tabelle 1	Download-URLs für die Java 3D API	11
-----------	-----------------------------------	----

1 Vorwort

Vor einiger Zeit bin ich in die Verlegenheit gekommen, Software zu evaluieren oder genauer eine 3D-Engine zu suchen, an die nicht eben geringe Anforderungen gestellt wurden. Sie sollte

- plattformunabhängig, aber mindestens für Windows und Linux in vergleichbarer Qualität verfügbar sein
- frei verfügbar aber zumindest für den Endanwender kostenlos sein
- dynamische Szenen unterstützen, es also erlauben, in einer aktuell angezeigten Szenerie Veränderungen vorzunehmen
- den Ansprüchen an eine moderne 3D-Engine genügen, also neben akzeptablen Frameraten bei komplexen Szenen auch die üblichen Effekte wie verschiedene Lichttypen, Sounds, Nebel, komplexe Texturen und vieles mehr bieten

Gerade der Punkt „Geschwindigkeit“ legte es nahe, Software zu verwenden, die native für die jeweilige Plattform kompiliert wird, was heißen würde, dass die Engine in C oder C++ geschrieben sein würde. Interessanterweise erwiesen sich gerade diese nativen Engines als ausgesprochen untauglich. Eines dieser plattformübergreifend verfügbaren Exemplare, die allen Anforderungen noch am nächsten kam, war die Crystal Space 3D Engine, die zum damaligen Zeitpunkt zwar äußerst ansehnliche Ergebnisse lieferte, das allerdings nach einer unakzeptabel hohen Initialisierungszeit. Bei nur mittelmäßig komplexen Szenen musste man hier schon minutenlang warten, bevor überhaupt etwas dargestellt wurde, von der anschließend verfügbare Framerate und Verarbeitungsgeschwindigkeit ganz zu schweigen.

So kam irgendwann die Idee auf, in einer Ecke zu suchen, die zwar für ihre Plattformunabhängigkeit bekannt war, mit der man aber „3D“ nicht unbedingt in Verbindung bringen würde: bei Java. Tatsächlich bot Sun eine 3D-API in damals der Version 1.2 an, die alle Anforderungen zu erfüllen schien – und das in einer unerwartet hohen Qualität und Geschwindigkeit.

Zurückblickend kann ich – jetzt da sich das Projekt in einem weit fortgeschrittenen Stadium befindet – nur feststellen, dass die damalige Entscheidung für eine Lösung die komplett auf Java und der Java 3D API beruht eine Gute war. Trotz einiger Fehler, Macken und Bugs – die wohl jedes Projekt dieser Komplexität mitbringt – lässt sich mit Java 3D gut arbeiten. Ich möchte sogar so weit gehen zu behaupten, dass diese API mit dem Kenntnisstand des Entwicklers skaliert. Man kann hier einen schnellen und leichten Einstieg finden ohne sich mit den Details der darunterliegenden Grafiksicht auskennen zu müssen. Die komplexeren und vielseitigen Möglichkeiten von Java 3D lassen sich später mehr und mehr verwenden und einsetzen, der Entwickler dringt mit wachsendem Wissensstand also schrittweise immer tiefer auch in komplexe Themen ein.

Leider war die Weiterentwicklung des oben angesprochenen Projektes nicht wirklich frei von bösen Überraschungen und unnötigen Adrenalin-schüben. Das lag aber in erster Linie

an diversen, offensichtlich gezielt gestreuten Desinformationen, die besagten, dass Sun alle Java-3D-Entwickler gefeuert und die Weiterentwicklung mit der Version 1.3.1 eingestellt hätte. Dass diese Gerüchte haltlos sind, zeigten jedoch Meldungen, dass Java 3D erstmalig für den Macintosh umgesetzt wurde und auch wieder Portierungen der
50 Version 1.3.x für IRIX, HP-UX und andere Betriebssysteme verfügbar wurden. Für Linux steht mittlerweile sogar eine Version zur Verfügung, die die AMD64 Plattform unterstützt.

Anfang diesen Jahres wurden die Negativgerüchte um J3D dann endgültig als Falschmeldungen und offensichtlichen FUD interessierter Kreise entlarvt: Mit einer
55 Meldung am 17. 03. 2004 gab Sun offiziell bekannt, dass Java 3D Open Source wird. Welche Bedeutung diese Nachricht für die Java-3D-Community hatte, zeigten nicht zuletzt die vielen Reaktionen innerhalb der Mailingliste „Java 3D Interest Group“, welche am 17. März und den folgenden Tagen wie eine Tsunami-Welle durch diese schwappte.

60 Die Zukunft sieht für Java 3D also mehr als positiv aus: Mit J3D 1.3 liegt eine Version vor, die technisch auf dem Stand der Dinge ist und mit ihrer Veröffentlichung als Open Source ist es den Entwicklern und Nutzern möglich, ihre eigenen Ideen einzubringen und die weitere Entwicklung selbst mit voranzutreiben. Und in Bezug auf Industrietauglichkeit und professionelle Einsetzbarkeit ist Java 3D über jeden Negativverdacht erhaben – und das
65 nicht erst, seit die NASA es für ihre Raumfahrtprojekte einsetzt.

Des weiteren ist der Schritt hin zu Open Source inzwischen erfolgreich vollzogen und die Java3D-Quellen sind – erweitert um einige interessante, neue Teilprojekte - auf
<https://dev.java.net> veröffentlicht worden.

70

Im folgenden soll nun diese API ausführlich dargestellt und beschrieben werden, es soll ein Einstieg geboten werden , der es auch ohne detailliertere Vorkenntnisse möglich macht, eigene 3D-Applikationen zu erstellen. Was für die Lektüre dieses Dokuments vorausgesetzt wird, sind lediglich Grundkenntnisse in Sachen Java und objektorientierter
75 Programmierung sowie ein wenig mathematisches Know-How. Es wurde versucht, in diesem Dokument alles so klar, eindeutig und verständlich wie möglich darzustellen und zu erklären. Sollte das einmal nicht gelungen sein, so freut sich der Autor über Feedback. Auch sind jede andere Art von Hinweisen, Verbesserungsvorschlägen und Kommentaren per Mail an virtual_worlds@gmx.de jederzeit willkommen.

80

Auf eines sei jedoch hingewiesen: Fragen zu Sachthemen werden per Mail prinzipiell nicht (kostenlos) beantwortet, hierfür stehen verschiedene Mailinglisten und Foren wie z.B. das unter <http://www.javacore.de> zur Verfügung.

2 Einführung

85

2.1 Die Entwicklungsumgebung

90

Damit Java 3D in eigenen Applikationen genutzt werden kann, muss die API zuvor installiert werden. Die Pakete sind für verschiedene Betriebssysteme verfügbar, die an unterschiedlichen Stellen im Internet heruntergeladen werden können. Hier werden jeweils die reine Runtime Environment (RT) angeboten, als auch das Software Development Kit (SDK). Um Java 3D Applikationen entwickeln zu können, wird das SDK benötigt.

Betriebssystem	URL
HP-UX	http://www.hp.com/products1/unix/java/java2/java3d/downloads/
Linux	http://www.blackdown.org oder https://j3d-core.dev.java.net/
Mac OS X	http://developer.apple.com/java/
Sparc Solaris	https://java3d.dev.java.net/
SGI Irix	http://www.sgi.com/products/evaluation/6.5_java3d_1.3.1
Windows	https://java3d.dev.java.net/

Tabelle 1 Download-URLs für die Java 3D API

95

Die in Tabelle 1 aufgeführten URLs beziehen sich dabei nur auf Betriebssysteme, für die die Java 3D API in der aktuellen Version 1.3.x verfügbar ist. Eine vollständige Auflistung aller unterstützten Betriebssysteme und damit auch der älteren Java 3D Versionen ist unter <http://java3d.virtualworlds.de/download.php> zu finden.

100

2.1.1 Besonderheiten der Windows-Version

105

Java 3D für Windows weist eine Besonderheit auf: Es ist in zwei unterschiedlichen Paketen für OpenGL und DirectX verfügbar. Diese Angabe bezieht sich auf die darunterliegende Grafikschnittstelle. Ob OpenGL verwendet werden kann, hängt im wesentlichen von der Unterstützung durch die vorhandene Grafikhardware und ihrer Treiber ab.

110

Ist die nötige Unterstützung für OpenGL nicht oder nicht vollständig vorhanden, was leider relativ häufig der Fall ist, sollte die Version für DirectX verwendet werden. Diese Variante müsste unter Windows im Allgemeinen funktionieren, sie setzt allerdings das Vorhandensein von mindestens DirectX 8.1 voraus. Das DirectX-Paket kann bei Microsoft kostenlos heruntergeladen werden, wo hingegen OpenGL – so es denn vernünftig unterstützt wird – bereits zusammen mit den Treibern der Grafikhardware installiert wird.

115

2.1.2 Installation und Konfiguration der Umgebung

120 Das Java 3D SDK wird in der Regel im Verzeichnis des verwendeten JDKs installiert. Wie die Installation für die jeweiligen Betriebssysteme im Detail von statten geht, ist den Installationsanleitungen der verschiedenen Pakete zu entnehmen. Diese finden sich ebenfalls unter den in Tabelle 1 aufgeführten Adressen. Hier soll nur kurz auf die Installation der SDKs für Linux und Windows eingegangen werden.

125 Zum Java 3D SDK gehören (wie auch zur RT) die folgenden Komponenten, die im Classpath liegen müssen bzw. von der JRE aus erreichbar sein sollten:

- jre/lib/ext/j3daudio.jar
- jre/lib/ext/j3dcore.jar
- jre/lib/ext/j3dutils.jar
- jre/lib/ext/vecmath.jar
- 130 - jre/bin/J3D.dll bzw. jre/lib/i386/libJ3D.so
- jre/bin/J3DUtils.dll bzw. jre/lib/i386/libJ3DUtils.so
- jre/bin/j3daudio.dll bzw. jre/lib/i386/libj3daudio.so

2.1.2.1 Installation des Java 3D SDK unter Linux

135 Das SDK der Version 1.3.1 für Linux kann auf einem der unter <http://www.blackdown.org> gelisteten Mirrors heruntergeladen werden. Bei diesem handelt es sich um ein selbstextrahierendes Archiv mit dem Namen java3d-sdk-1.3.1-linux-i386.bin. Um das Paket nach dem Herunterladen installieren zu können, muß es unter Umständen erst executeable gemacht werden. Das entsprechende Bit kann mit dem Aufruf von

140

```
# chmod 755 java3d-sdk-1.3.1-linux-i386.bin
```

145 in der Konsole gesetzt werden. Um das SDK an die richtige Stelle zu installieren, ist es notwendig, in das Verzeichnis des Ziel-JDKs zu wechseln. Für Java 1.4.2 ist das beispielsweise das Directory /usr/java/j2sdk1.4.2/. Von dort aus wird dann das heruntergeladene .bin-File ausgeführt. Das kann in der Konsole beispielsweise so aussehen:

```
150 # cd /usr/java/j2sdk1.4.2/  
# /home/User/java3d-sdk-1.3.1-linux-i386.bin
```

Es werden dann die Lizenzbestimmungen angezeigt, die mit „yes“ anzunehmen sind. Anschließend werden die Files entpackt und an die richtige Position kopiert. Damit sind
155 alle nötigen Schritte getan und die nötigen Files befinden sich jetzt im Verzeichnis des gewünschten JDK.

2.1.2.2 Installation des Java 3D SDK unter Windows

- 160 Die Installation des SDKs unter Windows geschieht mittels eines Setup-Programmes. Ist nur ein JDK auf dem ganzen System installiert, so ist dieses Setup lediglich auszuführen und das SDK somit ohne irgend welche Änderungen an den vorgegebenen Einstellungen von diesem Setup-Programm installieren zu lassen.
- 165 Sind mehrere verschiedene JDKs auf dem Zielsystem vorhanden, so muß unter Umständen das Richtige ausgewählt werden. Das ist während des Setup-Vorganges kurz vor dem Kopieren der Daten möglich, wenn angezeigt wird, wo das Java 3D SDK hin installiert werden soll. Gegebenenfalls ist an dieser Stelle der Pfad zu ändern und das gewünschte JDK anzugeben.

170

2.1.2.3 Die Konfiguration des Borland JBuilder

- Kommt der JBuilder als Entwicklungsumgebung zum Einsatz, so sind diesem die neuen Files des Java 3D SDK nach dessen Installation unbedingt bekannt zu machen. Wie das geschieht, hängt ein wenig von der Version ab. Im wesentlichen sind die nötigen Schritte aber identisch.

175

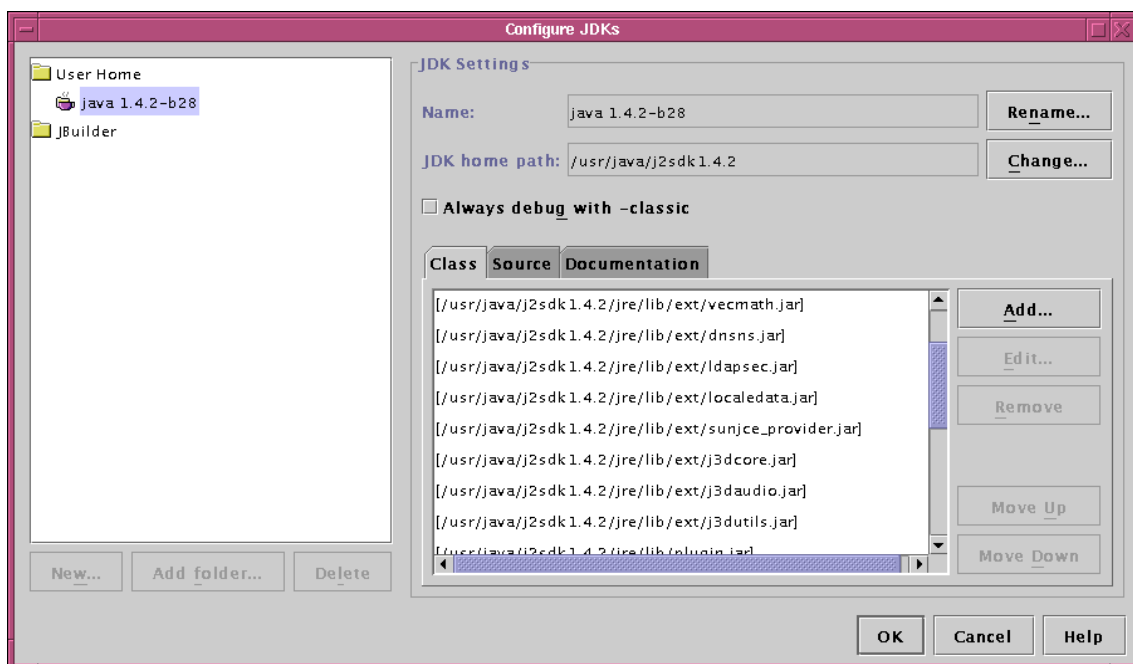


Abbildung 1 Konfiguration der Klassenpfade beim JBuilder

- 180 Im Menü „Tools“ des JBuilder findet sich ein Menüpunkt „Configure JDKs...“. Wird dieser ausgewählt, öffnet sich ein Fenster, das dazu dient, die Pfade zu den nötigen .jar-Files und Bibliotheken festzulegen. Um nun die veränderte Konfiguration des um Java 3D erweiterten JDKs bekannt zu machen, ist der Button „Change...“ hinter dem Feld „JDK home path“ zu betätigen und anschließend das Verzeichnis des JDKs, in dem Java 3D

eben installiert wurde, auszuwählen. Die Verzeichnisstruktur wird daraufhin untersucht und die gefundenen Klassen und Libraries werden dem Classpath automatisch hinzugefügt.

Anschließend sollte Java 3D innerhalb des JBuilder in vollem Umfang zur Verfügung stehen.

2.1.2.4 Die Konfiguration von Eclipse

Wenn die freie IDE Eclipse verwendet wird, sind ebenfalls ein paar kleine Schritte notwendig, um das neu hinzugekommene Java 3D SDK dieser bekannt zu machen.

Da hier die Klassenpfade für jedes Projekt einzeln definiert werden können, ist im Wesentlichen beim Anlegen eines neuen Projektes darauf zu achten, dass für dieses das richtige JDK gewählt wird. Das kann gegebenenfalls aber auch später noch in den Projekteinstellungen geändert werden.

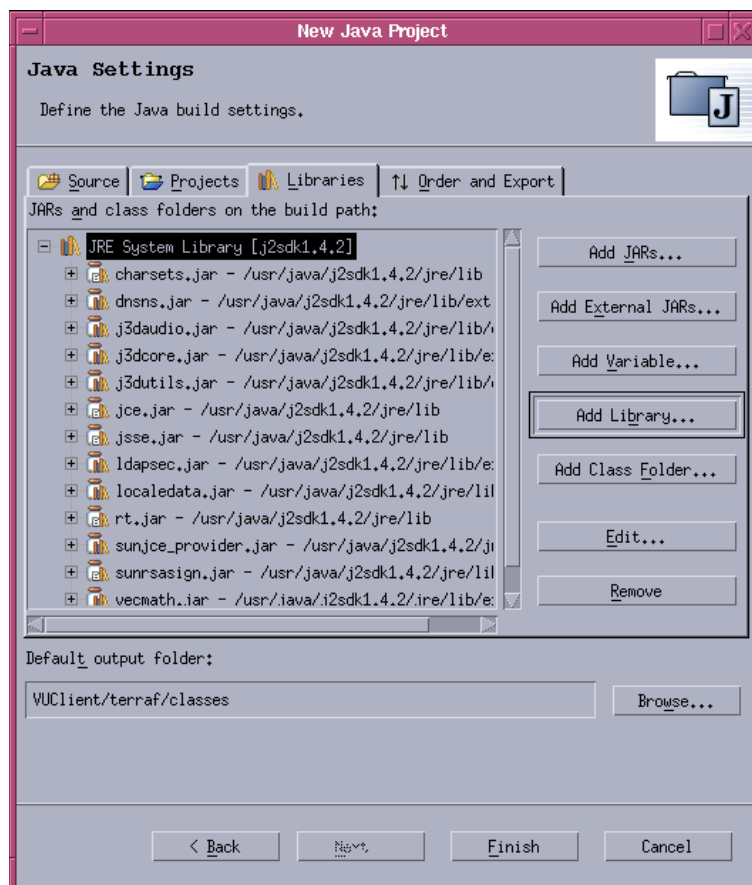


Abbildung 2 Klassendefinitionen bei den Eclipse-Projekteinstellungen

So fern es erforderlich ist, müssen hier die -jar-Files und Libraries der Java 3D API manuell mit Hilfe des Buttons „Add JARs...“ hinzugefügt werden.

205 **2.1.3 Java-3D-Applikationen und Java WebStart**

Die doch etwas aufwändige Installation des Java-3D-Paketes ist zum Glück nur für die Entwicklungsumgebung nötig. Für die spätere Auslieferung bzw. Bereitstellung des fertigen Programmes an den Endanwender bietet sich die Verwendung von Java
210 WebStart an. Auch dies ist ein Teilprojekt, das im Rahmen der Veröffentlichung als Open Source entstanden ist und das es ermöglicht, J3D-Applikationen per simplem Mausklick auf einer Webseite zu starten. Alle dazu für das jeweils vorliegende Betriebssystem notwendigen Daten und zusätzlichen Pakete werden dann automatisch heruntergeladen und installiert.

215 Die dazu nötigen Informationen und .jnlp-Files sind ebenfalls auf dev.java.net unter <https://j3d-webstart.dev.java.net/> zu finden und seien zumindest den Entwicklern wärmstens empfohlen, die planen, ihre J3D-Applikation per Webseite anzubieten und zu veröffentlichen.

220 Aber auch hier geht es leider nicht ohne einen Wermutstropfen ab: Die Möglichkeit J3D-Applikationen einfach und elegant per WebStart auszuführen steht Macintosh-Usern nicht zur Verfügung. Da Apple leider eine geradezu paranoide Lizenzpolitik verfolgt, war es den Entwicklern von Sun bisher nicht möglich, die entsprechenden plattformabhängigen Files
225 mit in dieses WebStart-Projekt zu integrieren. Vielleicht ändert sich daran ja etwas, wenn nur genügend Mac-User Apple per Mail auf die Füße treten...

3 Der Einstieg

3.1 Hallo Universum

230

Der Klassiker unter den Anfängerprogrammen schlechthin, das „Hallo Welt“-Programm, soll hier in einer passenden Abwandlung als Grundlage für den Einstieg in die Java-3D-Programmierung dienen. Dieser Einstieg stellt allerdings kein so sinnloses Programm dar wie das berühmte Vorbild, sondern wird im folgenden als Grundlage für alle weiteren Programme verwendet werden. Die grundlegenden, hier gezeigten Codeteile werden Ihnen also immer wieder begegnen:

235

```
(1)import java.applet.Applet;
(2)import java.awt.BorderLayout;
(3)import java.awt.GraphicsConfiguration;
(4)import com.sun.j3d.utils.applet.MainFrame;
(5)import com.sun.j3d.utils.universe.*;
(6)import javax.media.j3d.*;
(7)import javax.vecmath.*;
(8)
(9)public class Universe extends Applet
(10) {
(11)     private SimpleUniverse u=null;
(12)
(13)     public Universe()
(14)     {
(15)     }
(16)
(17)     public void init()
(18)     {
(19)         setLayout(new BorderLayout());
(20)         GraphicsConfiguration config=
            SimpleUniverse.getPreferredConfiguration();
(21)         Canvas3D c=new Canvas3D(config);
(22)         add("Center",c);
(23)         u=new SimpleUniverse(c);
(24)     }
(25)
(26)     public void destroy()
(27)     {
(28)         u.cleanup();
```

```

(29)      }
(30)
(31)  public static void main(String[] args)
(32)      {
(33)      new MainFrame(new Universe(), 400, 400);
(34)      }
(35) }

```

240 Die Grundstruktur sollte klar sein: es wird zur Darstellung ein Applet verwendet. Die Methoden `init()` und `destroy()` werden hier entsprechend der Spezifikation eingesetzt. Die Methode `main()` wurde hinzugefügt, um das Ausführen als Applikation ebenfalls zu ermöglichen.

245 Der erste interessante Konstrukt findet sich in Zeile 20. Hier wird eine so genannte `GraphicsConfiguration` erzeugt, die für das Erzeugen eines `Universe` zwingend notwendig ist. Diese `GraphicsConfiguration` entstammt dem Paket `java.awt` und beschreibt die Charakteristika eines Grafikausgabegerätes (bei welchem es sich hier im allgemeinen um den Monitor handeln dürfte).

250 Diese `GraphicsConfiguration` wird in Zeile 21 verwendet, um einen `Canvas3D` zu erzeugen. Dieses Objekt bildet gewissermaßen eine Schnittstelle zwischen der zweidimensionalen Welt (der Programmoberfläche) und der zu erzeugenden 3D-Szene. Mit Hilfe eines `Canvas3D` wird es möglich, eine 3D-Darstellung in ein GUI zu integrieren. Dementsprechend wird der hier erzeugte `Canvas3D` auch dem Applet hinzugefügt, so
 255 dass er sichtbar wird. Der Vollständigkeit halber sei erwähnt, dass `Canvas3D` zu den Heavyweight-Objekten gehört. Die Konsequenzen, die sich daraus ergeben können, werden später noch detailliert besprochen.

260 Unabhängig davon kann ein `Canvas3D` auch „unsichtbar“ verwendet werden, um z.B. Bilder im Hintergrund zu rendern und diese dann abzuspeichern oder anderweitig weiterzuverwenden. Dieses Verfahren wird „off-screen rendering“ genannt und soll an dieser Stelle vorerst nicht weiter beschrieben werden.

265 In Zeile 23 schließlich wird ein `Universum` erzeugt. Hier kommt bereits eines der Utilities des Java 3D Paketes zum Einsatz – das `SimpleUniverse` aus dem Paket `com.sun.j3d.utils.universe`.

270 Ein `Universe` ist – fast wie im richtigen Leben – die oberste Organisationsebene einer virtuellen 3D-Welt, innerhalb der sich alle Objekte befinden müssen. Ein solches `Universe` besitzt viele unterschiedliche Parameter und Konfigurationsmöglichkeiten. Diese sind für die meisten Anwendungen schlichtweg viel zu komplex bzw. es werden häufig immer wiederkehrende, gleiche Grundeinstellungen verwendet. Hier kann nun das `SimpleUniverse` zum Einsatz kommen: es stellt ein komplett konfiguriertes Basis-`Universum` zur Verfügung, das für die hier gezeigte Beispielanwendung geradezu perfekt

275 geeignet ist. Der Aufruf in Zeile 23 verbindet nun schließlich dieses Universe mit dem Canvas3D, der für die Darstellung in unserem Applet zuständig ist.

280 Wird der Code nun kompiliert und das Applet ausgeführt, ist nichts als ein schwarzer, quadratischer Bereich zu sehen. Das scheint auf einen Fehler im Programm hinzudeuten, ist jedoch vollkommen korrekt. Denn es befindet sich in diesem Universum bisher schlichtweg gar nichts, keine Objekte und kein Licht, was sollte also zu sehen sein?

3.2 Das erste 3D-Objekt

285 Um das optisch wenig eindrucksvolle Universum aus dem vorhergehenden Abschnitt nun etwas zu erweitern, soll der doch recht einfachen Szene jetzt ein erstes 3D-Objekt hinzugefügt werden.

290 Da 3D-Objekte und damit deren Erzeugung fast beliebig komplex werden können, soll zur Vereinfachung des Verfahrens eine Klasse aus dem Paket `com.sun.j3d.utils.geometry` verwendet werden, der `ColorCube`:

```
(1)import java.applet.Applet;
(2)import java.awt.BorderLayout;
295 (3)import java.awt.GraphicsConfiguration;
(4)import com.sun.j3d.utils.applet.MainFrame;
(5)import com.sun.j3d.utils.geometry.*;
(6)import com.sun.j3d.utils.universe.*;
(7)import javax.media.j3d.*;
300 (8)import javax.vecmath.*;
(9)
(10)public class Universe extends Applet
(11) {
(12)     private SimpleUniverse u = null;
305 (13)
(14)     public Universe()
(15)     {
(16)     }
(17)
310 (18)     public BranchGroup createSceneGraph()
(19)     {
(20)         BranchGroup RootBG=new BranchGroup();
(21)         RootBG.addChild(new ColorCube(0.4));
(22)         RootBG.compile();
```

```

315 (23)      return RootBG;
      (24)      }
      (25)
      (26)      public void init()
      (27)      {
320 (28)          setLayout(new BorderLayout());
      (29)          GraphicsConfiguration
          config=SimpleUniverse.getPreferredConfiguration();
      (30)          Canvas3D c = new Canvas3D(config);
      (31)          add("Center", c);
325 (32)          u = new SimpleUniverse(c);
      (33)
      (34)          u.getViewingPlatform().setNominalViewingTransform();
      (35)          u.addBranchGraph(createSceneGraph());
      (36)      }
330 (37)
      (38)      public void destroy()
      (39)      {
      (40)          u.cleanup();
      (41)      }
335 (42)
      (43)      public static void main(String[] args)
      (44)      {
      (45)          new MainFrame(new Universe(), 400, 400);
      (46)      }
340 (47) }

```

Erste Neuerungen finden sich in den Zeilen 34 und 35. Der Aufruf von `u.getViewingPlatform().setNominalViewingTransform()` modifiziert die oben bereits angesprochene Konfiguration des Universe. Hier wird die Position der ViewingPlatform, also die Position des Beobachters, verändert. Diese Veränderung bewirkt hier lediglich, dass sich der Beobachter etwas abseits des Nullpunktes des Universe befindet. Das ist nötig, weil der ColorCube exakt an den Koordinaten 0,0,0 hinzugefügt werden wird und nicht in der gewünschten Weise sichtbar wäre, wenn sich der Beobachter an exakt der gleichen Position befindet.

350

In Zeile 35 wird schließlich ein erstes Objekt dem Universe hinzugefügt – frei nach der Methode „Und der Autor sprach: es werde ein ColorCube. Und siehe da, es ward ein ColorCube“. Dass das Erscheinen dieses ColorCube jedoch kein Wunder ist, ist in der Methode `createSceneGraph()` zu sehen.

355

Als erstes wird hier (in Zeile 20) eine so genannte BranchGroup erzeugt. Mit Hilfe eines

solchen Objektes können mehrere Unterobjekte zu einer Szene hinzugefügt werden. Wie so eine Szene aufgebaut ist, wird in den folgenden Abschnitten detaillierter beschrieben.

360 In Zeile 21 ist zu sehen, wie ein ColorCube erzeugt wird. Der einzige Parameter, der hier verwendet wird, legt die Kantenlänge des ColorCube in Metern fest. Gleichzeitig wird dieses Objekt der bereits angesprochenen BranchGroup hinzugefügt.

365 Zeile 22 beinhaltet einen für Java 3D elementar wichtigen Aufruf. Hier wird die BranchGroup mitsamt allen untergeordneten Objekten (in diesem Fall nur der ColorCube) compiliert. Dieses Compilieren wandelt unseren so genannten SceneGraph in eine für die 3D-Engine optimierte interne Darstellung um. Klarer Vorteil dieses Vorgehens: die Performance kann dadurch deutlich gesteigert werden. Natürlich gibt es auch gewisse Nachteile. So benötigt dieser Compile-Vorgang zum einen natürlich etwas Zeit. Zum
370 anderen unterliegt ein kompilierter SceneGraph verschiedenen Einschränkungen, d.h. es kann an den kompilierten Elementen nichts mehr verändert werden. Aber auch dieser Nachteil läßt sich umgehen. Wie später noch ausführlich gezeigt wird, läßt sich detailliert festlegen, welche Eigenschaften nach dem Aufruf von `compile()` trotzdem veränderbar sein sollen.

375

Doch zurück zum aktuellen Code: Der neu erzeugte SceneGraph – bestehend aus einer BranchGroup und einem untergeordneten ColorCube – wird zurückgegeben, so dass er dem Universe hinzugefügt werden kann.

380 Nach dem Übersetzen und Ausführendes Programmes ist dieses mal etwas mehr zu sehen als beim ersten Entwurf. In der Mitte des Canvas3D findet sich ein rotes Quadrat. Dieses Quadrat ist der oben erzeugte ColorCube. Und auch hier ist die Ursache für diese wenig dreidimensionale Erscheinung sicher klar: der ColorCube wird ganz exakt von vorne betrachtet, so dass nur eine einzige Seite des Würfels zu sehen ist. Das soll nun im
385 folgenden Abschnitt geändert werden.

3.3 SceneGraph und Transformation

390 Nun soll das bisher verwendete Programm so erweitert werden, dass erstmalig zu sehen ist, dass Java 3D den Zusatz „3D“ wirklich nicht umsonst trägt. Zu diesem Zweck wird der ColorCube um je 45° um die X- und die Y-Achse rotiert. Um das zu ermöglichen, muß der bisher verwendete, sehr einfache SceneGraph ein wenig erweitert werden. Diese Änderungen werden ausschließlich in der Methode `createSceneGraph()` vorgenommen:

395

```
(1) public BranchGroup createSceneGraph()  
(2) {  
(3)     BranchGroup RootBG=new BranchGroup();  
(4)     TransformGroup CubeTG=new TransformGroup();  
400 (5)     Transform3D      CubeT3D=new Transform3D();
```

```

(6)
(7)  CubeT3D.setRotation(new AxisAngle4f(1f,1f,0f,(float)Math.toRadians(45)));
(8)  CubeTG.setTransform(CubeT3D);
(9)  CubeTG.addChild(new ColorCube(0.4));
405 (10)  RootBG.addChild(CubeTG);
(11)
(12)  RootBG.compile();
(13)  return RootBG;
(14)  }

```

410

In den Zeilen 3 bis 5 werden einige nötige Variablen deklariert und die jeweiligen Objekte auch gleich erzeugt. Neu findet sich hier die TransformGroup und das Transform3D Objekt. Diese werden benötigt, um den ColorCube in der gewünschten Weise zu rotieren.

415 Diese Rotation wird in Zeile 7 festgelegt. Das geschieht mit Hilfe der Methode `setTransform()`. Hier wird ein `AxisAngle4f`-Objekt übergeben. Die Parameter des Konstruktors dieser Klasse sind folgendermaßen festgelegt:

```
AxisAngle4f(float x, float y,float z,float angle)
```

420

Der eigentliche Winkel, um den rotiert werden soll, wird erst mit dem vierten Parameter übergeben. Um welche Achsen rotiert werden soll, wird mit den ersten dreien festgelegt. Hier sind die Parameter, die die gewünschten Achsen festlegen, auf eins zusetzen. Der Beispielcode oben zeigt, dass hier 45° um X- und Y-Achse rotiert werden soll. Es ist auch zu sehen, dass der Winkel nicht in Grad, sondern in Radians übergeben werden muss, was sich aber mit der passenden Math-Methode schnell umrechnen lässt.

425

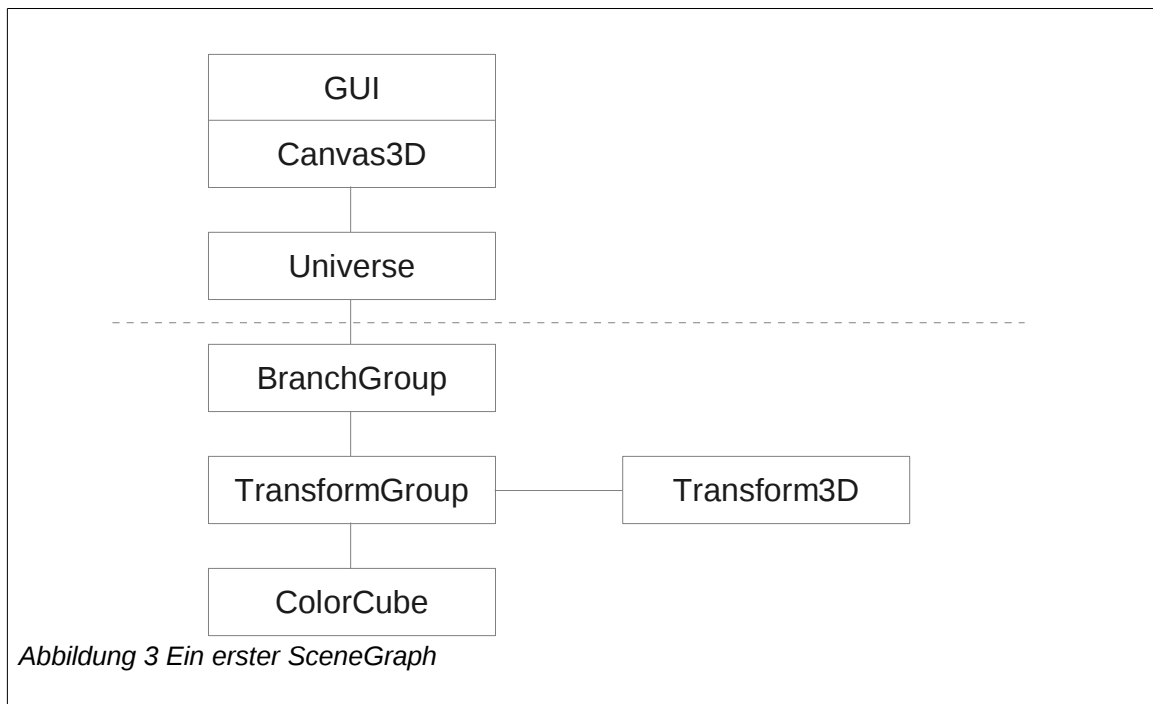
Das Transform3D-Objekt wird anschließend dem TransformGroup-Objekt zugeordnet. Da die Transformation auf alle untergeordneten Objekte wirkt, wird anschließend (in Zeile 9) der ColorCube der TransformGroup zugeordnet. Der Rest des Codes ist schon bekannt und nach dem Compilieren und Ausführen des Applets ist erstmalig zu sehen, dass dieser ColorCube wirklich ein dreidimensionales Objekt ist.

430

So kurz wie das Programm bisher auch ist, so zeigt es doch schon ein grundlegendes Prinzip von Java 3D, das SceneGraph-Konzept. Vereinfacht gesagt beinhaltet dieses, dass Objekte (so genannte Nodes) hierarchisch angeordnet werden. Nodes wie die TransformGroup, die die Darstellung beeinflussen, wirken dabei immer nur auf untergeordnete Nodes. Schaut man sich die Struktur der SceneGraphs in diesem Beispielprogramm einmal an, so ist die hierarchische Anordnung bereits zu erkennen:

435

440



3.3.1 BranchGroup und TransformGroup

Die Struktur in Bild 3 ist zweigeteilt. Der obere Teil ist relativ statisch, dieser Aufbau findet sich praktisch überall in der gleichen Form wieder. Der untere, durch die gestrichelte Linie abgetrennte Teil ist variabel. Je nach dem, was wie dargestellt werden soll, kann dieser Teil einer 3D Szene völlig anders aussehen.

In diesem ersten SceneGraph finden sich bereits mehrere wichtige Elemente, die hier kurz vorgestellt werden sollen.

3.3.1.1 BranchGroup

Eine BranchGroup ist ein recht einfaches Element, dass es jedoch erlaubt, mehrere weitere Unterelemente in den SceneGraph einzufügen. Das ist nicht selbstverständlich, da die meisten Nodes eben genau einen Childnode haben dürfen. Wichtig ist hier auch, dass eine BranchGroup nichts an der Darstellung einer Szene ändert – egal wie viele BranchGroups vorhanden sind und in welcher Reihenfolge sie angeordnet sind, das sichtbare Ergebnis wird dadurch nicht beeinflusst (ausser vielleicht die Performance, wenn zu viele BranchGroups verwendet werden).

Die wichtigsten Methoden einer BranchGroup sollen hier kurz vorgestellt werden:

```
void compile()
```

Es werden die BranchGroup und alle ihr untergeordneten Elemente compiliert, um sie für die Darstellung zu optimieren, Details hierzu wurden bereits weiter oben beschrieben. Diese Methode darf nicht aufgerufen werden, wenn das zugehörige BranchGroup-Objekt bereits live ist. Andernfalls würde eine RestrictedAccessException geworfen.

```
void addChild(Node)
```

Diese Methode entstammt eigentlich der Klasse Group, von der BranchGroup direkt abgeleitet ist. Sie fügt dem SceneGraphen einen neuen Node als Child dieser BranchGroup hinzu. Ist diese BranchGroup selbst bereits Teil eines compilierten SceneGraphen oder ist dieser SceneGraph live, so muß das hinzuzufügende Objekt ebenfalls vom Typ BranchGroup sein. Andernfalls wird eine RestrictedAccessException geworfen.

```
void detach()
```

465 Diese Methode entfernt die BranchGroup aus dem SceneGraph, in der Art, das sie aus demjenigen Node entfernt wird, dessen Child sie selbst ist. Anders gesagt, die BranchGroup entfernt sich selbst aus dem übergeordneten Eltern-Objekt.

Die BranchGroup erbt von folgenden Klassen:

```
470 java.lang.Object
    javax.media.j3d.SceneGraphObject
        javax.media.j3d.Node
            javax.media.j3d.Group
                javax.media.j3d.BranchGroup
```

3.3.1.2 TransformGroup und Transform3D

475 Eine TransformGroup kann – wie die BranchGroup – mehrere Children haben. Im Unterschied dazu werden die der TransformGroup untergeordneten Nodes transformiert, d.h. Sie können eine geänderte Position haben, um eine oder mehrere Achsen rotiert oder sogar in der Größe verändert werden. Wie diese Transformation genau aussieht, 480 beschreibt ein zu dieser Group gehörendes Transform3D-Objekt. Doch zuerst zur TransformGroup:

3.3.1.2.1 TransformGroup

485 Die Methoden, die eine TransformGroup zur Verfügung stellt, weichen von denen der BranchGroup ab, insbesondere fehlen `compile()` und `detach()`.

```
void addChild(Node child)
```

Diese Methode fügt dem SceneGraph einen neuen Node als Child dieser

TransformGroup hinzu. Ist diese TransformGroup bereits live oder compiliert, so muß das child vom Typ BranchGroup sein, da sonst eine RestrictedAccessException geworfen wird. Diese Methode stammt ebenfalls von der übergeordneten Klasse Group.

```
setTransform3D( )  
getTransform3D( )
```

Mit diesen Methoden ist es möglich, ein neues Transform3D-Objekt zu setzen bzw. das aktuelle Transform3D-Objekt zu holen, das die Transformation beschreibt, die für alle Nodes unterhalb des zugehörigen TransformGroup-Objektes gelten.

```
java.lang.Object  
    javax.media.j3d.SceneGraphObject  
490         javax.media.j3d.Node  
            javax.media.j3d.Group  
                javax.media.j3d.TransformGroup
```

3.3.1.2.2 Transform3D

495 Ein Transform3D-Objekt beschreibt, wie ein Teil-SceneGraph transformiert werden soll. Das geschieht mit Hilfe eines mathematischen Konstruktes, das es ermöglicht, diese für ein dreidimensionales Koordinatensystem festzulegen: einer 4x4 Matrix. „Transformation“ heißt für eine solche Matrix, dass sie die Verschiebung (also die Position) in X-, Y- und Z-Richtung, die Skalierung (ebenfalls für die X-, Y- und Z-Achse) sowie die Rotation um X-,
500 Y- und Z-Achse festlegt. Diese Matrix kann mit Hilfe verschiedener `get...()`- und `set...()`-Methoden geholt bzw. gesetzt werden. Verschiebungen, Rotationen usw. lassen sich natürlich auch einfacher über spezielle Methoden festlegen.

Die wichtigsten aber bei weitem nicht einzigsten Methoden der Transform3D Klasse
505 wieder in Kürze:

```
void setScale(double)  
double getScale()
```

510 Diese Methode setzt den Skalierungsfaktor der aktuell verwendeten Matrix bzw. gibt ihn zurück. Dieser Faktor gibt an, um wie viel größer oder kleiner die beeinflussten 3D-Objekte in der Szene erscheinen sollen.

```
void mul(Transform3D)
```

515 Hier handelt es sich um eine Methode, die eine mathematische Operation ausführt: Sie multipliziert eine andere Matrix (die in dem übergebenen Transform3D-Objekt enthalten ist) mit der Matrix des eigenen Transform3D-Objekts (also mit `this`).

```
void rotX(double)  
void rotY(double)  
520 void rotZ(double)
```

Diese Methodenaufrufe setzen je eine Rotation um X-, Y- bzw. Z-Achse. Hier ist zu beachten, dass alle anderen Anteile der aktuellen Matrix komplett überschrieben werden (um das zu umgehen, wäre es bei der Verwendung dieser Methoden nötig, je eine Rotation auf jeweils ein eigenes Transform3D-Objekt anzuwenden und diese

525 anschließend miteinander zu multiplizieren)

```
void setRotation(AxisAngle4f)
```

530 Diese Methode setzt eine Rotation mit Hilfe eines – wie im Beispielprogramm verwendeten – AxisAngle4f-Objektes. Bei der Verwendung dieser Methode werden alle anderen Rotationsanteile der Matrix überschrieben, im Gegensatz zu den `rot...()`-Methoden bleiben Translations- und Skalierungsanteile jedoch erhalten.

```
void setTranslation(Vector3d)
```

```
void setTranslation(Vector4f)
```

535 Mit dieser Methode ist es möglich, eine Verschiebung (also eine Positionsänderung relativ zur aktuellen Position) mit Hilfe eines Vektors zu setzen, der die gewünschte Verschiebung für X-, Y- und Z-Achse spezifiziert.

540 Natürlich können auch mehrere Transformationen mit Hilfe mehrerer TransformGroup / Transform3D Kombinationen hinter- oder in Anlehnung an die obige Darstellung des SceneGraph-Aufbaus besser untereinander geschaltet werden. Das Resultat ist eine Position, die aus der Summe der relativen Positionen zur jeweils vorhergehenden Transformation besteht. Für die Gesamtrotation und -größe gilt ebenfalls, dass sie die Summe aller Rotationen und Skalierungsfaktoren darstellt. Damit wird klar, dass sich bei 545 Transformationen innerhalb komplexer SceneGraphs unter Umständen einiges an Optimierungsmöglichkeiten ergibt, wenn es möglich ist, diese zusammenzufassen.

Abschließend wieder ein Blick auf die diesmal recht kurze „Erbfolge“:

```
550 java.lang.Object  
    javax.media.j3d.Transform3D
```

3.3.1.3 Funktioniert es?

555 Um die Funktionsweise von BranchGroup und TransformGroup sowie das Prinzip, dass Veränderungen durch eine Transformation immer nur auf den untergeordneten Teil-SceneGraph wirken, zu überprüfen, erhält das Beispielprogramm eine neue Zeile 11:

```
RootBG.addChild(new ColorCube(0.4));
```

560 Jetzt hat die BranchGroup zwei Child-Nodes. Des weiteren zeigt sich nach dem Compilieren und Ausführen des Programmes, dass die Transformation auf nur einen Teil-SceneGraphen wirkt: es sind jetzt zwei ColorCubes zu sehen, die auf Grund der identischen Position ineinander stecken, von denen aber nur ein einziger rotiert wurde.

565

3.4 Daten-Sharing

570 Car-Sharing ist eine feine Sache: Mehrere Leute nutzen ein Auto und teilen sich somit die fixen und nicht unerheblichen Kosten, die in Form von Versicherung, KFZ-Steuer und den regelmäßig wiederkehrenden Inspektionen entstehen. Umweltfreundlicher ist es auch noch, wenn dann nur noch aus einem Auto Öl tropft, statt aus mehreren (was jetzt nicht heißen soll, dass Car-Sharing eine empfehlenswerte Methode ist, um die Unmengen an Öl, die aus Autos jährlich in die Meere gelangen, zu reduzieren. Eine neue Dichtung wäre


```

605  (13)  RootBG.addChild(CubeTG);
      (14)  RootBG.addChild(Lnk2);
      (15)  RootBG.compile();
      (16)  return RootBG;
      (17)  }

```

610

Die Verbindungen von den Link-Nodes hin zur SharedGroup werden schon bei der Erzeugung der Objekte Lnk1 und Lnk2 angelegt, was einer der möglichen Link-Node-Konstruktoren ermöglicht. In den Zeilen 12 und 14 werden jetzt diese Link-Nodes an Stelle der beiden ColorCubes als Children eingehängt. Der ColorCube ist jetzt Child der SharedGroup. Somit ist die in Abbildung 4 dargestellte Struktur erstellt worden, was bedeutet, dass jetzt tatsächlich nur noch ein einziger ColorCube verwendet wird.

Nach dem Übersetzen und Starten des geänderten Programmes zeigt sich das bekante Bild: zwei ColorCubes, die ineinander stecken und von denen einer rotiert wurde. Ein Blick in den Code zeigt aber deutlich, dass wirklich nur ein einziger ColorCube erzeugt wurde. Das bedeutet, dieses 3D-Objekt wurde tatsächlich mehrfach verwendet.

Um das zu verdeutlichen könnten Sie hier nach Zeile 9 weitere Manipulationen am Transform3D-Objekt vornehmen, so beispielsweise eine zusätzliche Skalierung einfügen. Auch in diesem Fall wird immer nur der eine, betroffene Zweig des SceneGraphen verändert, das andere 3D-Objekt, das jedoch die gleiche Datenbasis nutzt bleibt auch davon unverändert. Es geht sogar noch weiter. Statt wie hier nur ein einzelnes Primitive zu sharen ist es natürlich auch möglich, ganze Teil-SceneGraphen mittels dieses Verfahrens mehrfach zu nutzen.

630

3.4.1 Link

Wie oben bereits beschrieben dient ein Link-Node als Verbindung zur SharedGroup, während er auf der anderen Seite an Stelle des zu sharenden Objektes in den SceneGraph eingehängt wird.

```

void setSharedGroup(SharedGroup)

```

Diese Methode der SharedGroup macht das, was auch schon mit Hilfe des Konstruktors möglich ist: es wird die dem Link-Node zugehörige SharedGroup zugewiesen

```

SharedGroup getSharedGroup()

```

Das Gegenstück zur vorhergehenden Methode findet sich mit dieser hier: Sie gibt die aktuell zugeordnete SharedGroup zurück

645

Und abschließend wiederum ein Blick darauf, wie sich diese Klasse ableitet:


```
java.lang.Object
    javax.media.j3d.SceneGraphObject
650         javax.media.j3d.Node
                javax.media.j3d.Leaf
                        javax.media.j3d.Link
```

3.4.2 SharedGroup

655

Wie ebenfalls schon beschrieben führt die SharedGroup die Verbindungen von den verschiedenen Links zusammen, um damit auf das zu sharende 3D-Objekt bzw. auf den zu sharenden Teil-SceneGraphen zu zeigen. Es ist allerdings zu beachten, dass der darunterliegende Teil-SceneGraph nicht jede Art von Node beinhalten kann. Folgende bisher noch nicht beschriebene Node-Typen sind nicht erlaubt und würden eine `IllegalSharingException` auslösen:

- AlternateAppearance (für Definitionen eines alternativen Aussehens von 3D-Objekten bezogen auf einen Einflußbereich statt auf ein Shape3D)
- Background (für die Definition des Hintergrundes)
- 665 - Behavior (ein Node-Typ für verschiedene User-definierte Operationen)
- BoundingLeaf (ein Node für Begrenzungen)
- Clip (definiert regionales Clipping)
- Fog (Node für in der Szene sichtbaren Nebel)
- ModelClip (spezifisches regionales Clipping)
- 670 - Soundscape (für den Klangeindruck von Sounds innerhalb eines bestimmten Bereiches)
- ViewPlatform (die aktuelle Position des Betrachters bzw. der Kamera)

Weitere interessantere Methoden sind für diese Klasse neben den schon bekannten:

```
void compile()
```

Diese Methode compiliert den untergeordneten Teil-SceneGraph (wie bereits von der BranchGroup her bekannt). Dazu darf das SharedGroup-Objekt jedoch nicht bereits live sein, da das Resultat sonst wieder eine `RestrictedAccessException` wäre.

```
675 Link[] getLinks()
```

Um in Erfahrung zu bringen, welche Links bereits zu dem zugehörigen Objekt führen, ist diese Methode nützlich: sie gibt ein Array mit allen Links, die mit dieser SharedGroup verbunden sind, zurück

680 Wie es der Name bereits andeutet, leitet sich auch die SharedGroup von der Klasse Group ab:

```
685  java.lang.Object
      javax.media.j3d.SceneGraphObject
      javax.media.j3d.Node
      javax.media.j3d.Group
      javax.media.j3d.SharedGroup
```

4 Objekteigenschaften

690

Der bisher verwendete ColorCube sieht für den Anfang sicher ganz nett aus. Allerdings sind bunte Würfel nun nicht alles, was man in der realen Welt findet. Selbst für die Darstellung eines Polo Harlekin wäre dieser etwas wenig. Aus diesem Grund soll es nun daran gehen, die 3D-Objekt- und Materialeigenschaften zu verändern, um diesen nicht nur verschiedene Farben zuzuweisen, sondern auch verschiedene Materialien darzustellen.

695

700

Gleichzeitig soll im folgenden Beispielcode auch ein neues Basisobjekt (das sich wie der ColorCube von der Klasse Primitive ableitet) verwendet werden. Aus diesem Grund wird im jetzt verwendeten Programm die Stelle, in der der ColorCube erzeugt wird, durch folgenden Konstrukt ersetzt:

```
new Sphere(1.0f, Sphere.GENERATE_NORMALS, 40, new Appearance())
```

705

Kurz zu den Parametern des Sphere-Konstruktors: hier werden der Radius der zu erzeugenden Kugel angegeben, einige Flags (das hier verwendete bewirkt unter anderem, dass die Kugel Licht an ihrer Aussenseite reflektiert), die Anzahl der Unterteilungen (um so mehr hier angegeben werden, um so exakter und glatter wird die Kugel) sowie ein Appearance-Objekt, das in diesem Beispiel mit Defaultwerten erzeugt wird. Soviel vorweg: dieses Appearance-Objekt wird später dazu verwendet, um die vielfältigen Materialeigenschaften des zugehörigen 3D-Objektes zu verändern.

710

715

Nach dem Compilieren und Ausführen des Programmes öffnet sich das altbekannte Fenster, in dem dann – nein leider keine Kugel – sondern eine plumpe Scheibe zu finden ist. Die Ursache dafür, dass die gewünschte Kugel so flach erscheint, ist im Appearance-Objekt zu suchen. Dieses bewirkt mit den voreingestellten Werten diese unschöne Darstellung.

4.1 Farben

720

Als erstes soll deswegen die Farbe des Objektes verändert werden. Auf Grund des verwendeten Beleuchtungsmodells sowie einiger spezifischen Eigenschaften der 3D-Engine bewirkt das auch eine realistischere dreidimensionale Darstellung der Kugel. Da das bisher verwendete Universe mangels Lichtquellen komplett dunkel ist, sind allerdings noch einige andere Maßnahmen nötig:

725

```
(1) public BranchGroup createSceneGraph()  
(2) {  
(3)     BranchGroup      RootBG=new BranchGroup();  
(4)     TransformGroup   SphereTG=new TransformGroup();  
(5)     Transform3D       SphereT3D=new Transform3D();
```

```

(6)    Appearance      SphereAppearance=new Appearance();
(7)
(8)    AmbientLight    ALgt=new AmbientLight(new Color3f(1f,1f,1f));
735 (9)    DirectionalLight DLgt=new DirectionalLight(new Color3f(1f,1f,1f),new
      Vector3f(-0.5f,-0.5f,-1f));
(10)   BoundingSphere  BigBounds=new BoundingSphere(new Point3d(),100000);
(11)
(12)   ALgt.setInfluencingBounds(BigBounds);
740 (13)   DLgt.setInfluencingBounds(BigBounds);
(14)
(15)   SphereT3D.setTranslation(new Vector3f(0f,0f,-1.5f));
(16)   SphereTG.setTransform(SphereT3D);
(17)   SphereAppearance.setMaterial(new Material(new Color3f(0f,0f,1f),new
745   Color3f(0f,0f,0f),new Color3f(1f,0f,0f),new Color3f(1f,1f,1f),100f));
(18)   SphereTG.addChild(new
      Sphere(1.0f,Sphere.GENERATE_NORMALS,40,SphereAppearance));
(19)   RootBG.addChild(SphereTG);
(20)   RootBG.addChild(ALgt);
750 (21)   RootBG.addChild(DLgt);
(22)
(23)   RootBG.compile();
(24)   return RootBG;
(25)   }

```

755

4.1.1 AmbientLight und DirectionalLight

Diese Maßnahmen finden sich in Form der Zeilen 8 bis 13. Hier werden Nodes hinzugefügt, die für die Beleuchtung der Szenerie sorgen. Das Umgebungslicht setzt sich hier aus zwei Elementen zusammen, dem DirectionalLight und dem AmbientLight.

Beide Light-Nodes haben eines gemeinsam: mittels der Methode `setInfluencingBounds()` muss festgelegt werden, innerhalb welchen Bereiches das Licht auf Objekte wirken soll. Das wird hier mit der `BoundingSphere` getan, die in Zeile 10 erzeugt wird. Diese hat einen Radius von 100000 m, welcher mit dem zweiten Parameter des `BoundingSphere`-Konstruktors übergeben wird. Da die Light-Nodes ohne jegliche Transformation an das Universe gebunden werden, heißt das, dass sie auf alle Objekte wirken, die sich innerhalb dieses Radius um den Nullpunkt des Koordinatensystems befinden.

770

Nur wenn sich ein 3D-Objekt innerhalb dieser „Influencing Bounds“, also innerhalb der gewählten Grenzen des Einflußbereiches befindet, wirkt das Licht auf ihn. Das interessante an dieser Methode ist, dass es damit möglich ist, positionsabhängig

775 unterschiedliche Beleuchtungen zu definieren. Überlappen sich für bestimmte 3D-Objekte mehrere Bounds, so wird zumindest für die Umgebungslichttypen das des am nächsten liegenden Bounds-Objekt verwendet.

4.1.1.1 AmbientLight

780 Dieser Typ des Umgebungslichtes legt die Beleuchtungsanteile der gleichmäßigen Beleuchtung fest. „Gleichmäßig“ bedeutet hier, das bei diesem Typ das Licht in gleicher Intensität von allen Seiten kommt, keine spezifische Richtung hat und die Objekte deswegen von allen Seiten gleich beleuchtet. Aus diesem Grund wird für den Konstruktor auch nur ein Color3f-Objekt übergeben, das die Farbe des Umgebungslichtes festlegt.

785

Die Verwandtschaftsverhältnisse sehen für diesen Lichttyp folgendermaßen aus:

```
java.lang.Object
    javax.media.j3d.SceneGraphObject
790         javax.media.j3d.Node
                javax.media.j3d.Leaf
                        javax.media.j3d.Light
                                javax.media.j3d.AmbientLight
```

795 4.1.1.2 DirectionalLight

Die zweite Lichtkomponente spezifiziert ein Licht, das nur aus einer Richtung kommt und 3D-Objekte demzufolge nur teilweise beleuchtet. Für dieses gerichtete Licht ist deswegen auch ein erweiterter Konstruktor nötig, der neben der Farbe auch einen Vektor festlegt, 800 der die Richtung bestimmt, aus der das Licht kommt. Die Lichtquelle befindet sich dabei in unendlicher Entfernung, die Strahlen des gerichteten Lichtes verlaufen parallel und es findet keine entfernungsabhängige Änderung der Beleuchtung statt.

DirectionalLight leitet sich ähnlich ab wie AmbientLight:

805

```
java.lang.Object
    javax.media.j3d.SceneGraphObject
        javax.media.j3d.Node
            javax.media.j3d.Leaf
10         javax.media.j3d.Light
                javax.media.j3d.DirectionallLight
```

4.1.2 Appearance

815 Wie im obigen Beispielcode zu sehen wird in Zeile 18 die gewünschte Sphere erzeugt.

Dabei wird auch ein Objekt vom Typ Appearance übergeben. Diese Appearance ist ein Container für sämtliche Eigenschaften, die das Erscheinungsbild eines 3D-Objektes in Bezug auf sein Material, nicht aber seine geometrische Form beeinflussen. Das sind unter anderem die hier verwendeten Materialeigenschaften, die auch die Farbe festlegen.

820

Die wichtigsten und hier auch elementaren Methoden setzen und holen im wesentlichen die Komponenten, für die das Appearance-Objekt den Container darstellt. Da diese möglichen Komponenten im folgenden detailliert beschrieben werden, macht eine separate Auflistung der Methoden hier natürlich keinen Sinn. Einen Blick auf die Klassen, von denen sich Appearance ableitet, soll es aber auf alle Fälle wieder geben:

825

```
java.lang.Object
    javax.media.j3d.SceneGraphObject
        javax.media.j3d.NodeComponent
            javax.media.j3d.Appearance
```

830

4.1.3 Material

Im Beispielcode werden die Eigenschaften der Sphere mit Hilfe der Klasse „Material“ definiert. Diese beeinflusst im wesentlichen das Reflektionsverhalten dieses 3D-Objektes. Es kam hier folgender Konstruktor zum Einsatz, der bereits alles nötige definiert:

835

```
Material(new Color3f(0f,0f,1f),new Color3f(0f,0f,0f),new
Color3f(1f,0f,0f),new Color3f(1f,1f,1f),100f)
```

840

Es ist zu sehen, dass hier mehrere Farben gesetzt werden müssen. Diese sind im einzelnen in der Reihenfolge, in der sie dem Konstruktor übergeben werden:

- Ambient Color – die Farbe, die mit dem gleichmäßigen Umgebungslicht korrespondiert, der hier gesetzte Farbwert entspricht einem satten grün

845

- Emissive Color – die Farbe, in der das 3D-Objekt selbst leuchtet
Hier ist zu beachten, dass diese Farbe nichts mit Beleuchtungseffekten im Sinne einer Lichtabstrahlung zu tun hat und sie dementsprechend auch keinen Einfluß auf andere 3D-Objekte hat. Dieser Wert legt lediglich unabhängig von der Beleuchtung des zugehörigen 3D-Objektes durch externe Lichtquellen fest, in welcher Farbe es erscheinen soll. Mit anderen Worten: wird hier ein anderer Farbwert als 0,0,0 für schwarz angegeben, so würde das 3D-Objekt auch ohne externe Lichtquellen sichtbar sein, ganz so, als ob es selbst leuchten würde.

850

Da allerdings eine Beleuchtung vorhanden ist, die sich im 3D-Objekt reflektieren soll, wird diese Eigenschaft nicht benötigt. Aus diesem Grund wurde an dieser Stelle auch schwarz als „Farbe“ gewählt.

855

Die „Emissive Color“, die bei der Verwendung der voreingestellten Materialwerte auf weiß (also Color3f(1f,1f,1f)) gesetzt wird, ist im übrigen auch der Grund dafür, dass die Sphere zuvor komplett flach erschienen ist. Dieser Wert hat dazu geführt, dass die Kugel selbstleuchtend war. Dieser unschöne Effekt ist dadurch entstanden, dass dieses Licht völlig gleichmäßig angestrahlt wurde und deswegen durch externe Lichtquellen

860

(speziell das DirectionalLight) keinerlei Schattierungen mehr möglich waren, die die runde Form der Sphere hätten verdeutlichen können.

- Diffuse Color – die Farbe, die Reflektiert wird, wenn das zugehörige 3D-Objekt beleuchtet wird
865 Diese Beleuchtung kann dabei sowohl durch das DirectionalLight der Umgebung als auch durch eine separate Lichtquelle (wie beispielsweise einem noch zu beschreibenden PointLight) geschehen. Um einen klaren Unterschied zur AmbientColor zu sehen, wurde diese Farbe auf rot gesetzt.
- Specular Color – die Farbe des Glanzpunktes des Objektes
870 Dieser Farbwert, der in diesem Beispiel auf einen komplett weißen Farbton gesetzt wurde, bestimmt zusammen mit dem folgenden Wert am meisten den Eindruck, ein spezifisches Material vorliegen zu haben.
- Shininess – hier wird – abweichend zu den anderen Parametern – keine Farbe spezifiziert, sondern ein Faktor im Bereich von 1 .. 128 angegeben. Dieser Faktor
875 bestimmt das Reflexionsverhalten des Materials. Experimentiert man ein wenig mit diesem Wert, stellt man schnell fest, dass sich die Größe und das Aussehen des Glanzpunktes der Kugel verändert. Zusammen mit diesem und dem Farbwert für die Specular Color gewinnt das menschliche Auge einen Eindruck vom Material. Hierbei gilt: um so kleiner und schärfer abgegrenzt der Glanzpunkt ist, um so glatter und härter
880 erscheint das Material, um so größer er ist, um so weicher und matter scheint es zu sein.

Das Material-Objekt muß dem Appearance-Objekt hinzugefügt werden, damit es das Aussehen der Kugel beeinflussen kann. Das geschieht – wie in obigem Code zu sehen ist
885 – recht simpel mit der Methode `setMaterial()` der Klasse `Appearance`. Das aktuelle Material eines Appearance-Objektes kann mit dem Gegenstück dazu, also mit der Methode `getMaterial()`, ermittelt werden.

Des weiteren bietet die Klasse `Material` unter anderem verschiedene Methoden an, um die
890 oben bereits angesprochenen Farben und Faktoren mittels `set...()` einzeln zu setzen oder die aktuellen Werte vom Material mit Hilfe von `get...()` zu holen:

```
void getAmbientColor(Color3f color)
void setAmbientColor(Color3f color)
895 void setAmbientColor(float r, float g, float b)
```

Diese Methoden ermöglichen es, den Farbwert für die Ambient Color zu ermitteln oder ihn neu zu setzen. Bei der Methode `getAmbientColor()` wird der Farbwert dabei in das als Parameter übergebene `Color3f`-Objekt kopiert, während umgekehrt bei `setAmbientColor(Color3f color)` die Farben aus dem übergebenen Objekt zum
900 Material-Objekt kopiert werden.

```
int getColorTarget()
void setColorTarget(int colorTarget)
```

Wie sich in späteren Kapiteln noch zeigen wird, ist es im Zusammenhang mit

905 komplexeren Geometriedaten möglich, auch für jedes Polygon, aus denen sich ein 3D-Objekt immer zusammensetzt, eine separate Farbe festzulegen. Mit Hilfe dieser Methoden ist es nun möglich, zu ermitteln, für welche der vier möglichen Farbtypen innerhalb des Material-Objektes diese Farbinformation gilt, bzw. es kann neu festgelegt werden, für welche sie gelten soll. Das geschieht jeweils mit Hilfe einer Konstanten, die von
910 `getColorTarget()` zurückgeliefert wird bzw. Die `setColorTarget()` als Parameter mitzugeben ist. **AMBIENT** gibt dabei an, dass diese so genannten per-Vertex Farbinformationen der Polygone für die Ambient Color gelten sollen. Bei **EMISSIVE** wird die Farbinformation dementsprechend die Emissive Color beeinflussen. Nach dem gleichen Muster gilt für **DIFFUSE**, dass die oben bereits beschriebene Diffuse Color für diese Art der Verwendung bestimmt wird und bei **SPECULAR** die Specular Color. Ein
915 wenig abweichend davon legt **AMBIENT_AND_DIFFUSE** fest, dass sowohl Ambient Color als auch Diffuse Color durch die per-Vertex Farbinformationen beeinflusst werden sollen.

```
void getDiffuseColor(Color3f color)
920 void setDiffuseColor(Color3f color)
void setDiffuseColor(float r, float g, float b)
```

Diese Methoden ermöglichen es, den Farbwert für die Diffuse Color zu ermitteln oder neu zu setzen. Bei der Methode `getDiffuseColor()` wird der Farbwert dabei wiederum in das als Parameter übergebene `Color3f`-Objekt kopiert, während
925 umgekehrt bei `setDiffuseColor(Color3f color)` die Farben aus dem übergebenen Objekt in das Material-Objekt kopiert werden.

```
void getEmissiveColor(Color3f color)
void setEmissiveColor(Color3f color)
930 void setEmissiveColor(float r, float g, float b)
```

Ähnlich den vorangegangenen Methoden ermöglichen diese es nach dem gleichen Muster, den aktuellen Farbwert für die Emissive Color zu ermitteln oder neu zu setzen.

```
boolean getLightingEnable()
935 void setLightingEnable(boolean state)
```

Material-Objekte besitzen intern ein so genanntes Lighting Flag, das festlegt, ob das zugehörige 3D-Objekt Licht von einer separaten Lichtquelle reflektiert (`=true`) oder nicht (`=false`). Dieses Flag kann mit diesen beiden Methoden ermittelt bzw. neu gesetzt werden.

```
940 float getShininess()
void setShininess(float shininess)
```

Ähnlich den get- und set-Methoden für Farben erlauben es diese beiden Methoden, den aktuellen Wert für die Shininess zu ermitteln oder ihn neu zu setzen.

```
945 void getSpecularColor(Color3f color)
```



```
void setSpecularColor(Color3f color)
void setSpecularColor(float r, float g, float b)
```

950 Diese Methoden beziehen sich auf die letzte noch verbleibende Farbe, auf die Specular Color und erlauben es nach dem mittlerweile bekannten Prinzip, den aktuellen Farbwert für diese zu ermitteln oder ihn neu zu setzen.

Die Klasse Material leitet sich ähnlich wie Appearance ab:

```
955 java.lang.Object
      javax.media.j3d.SceneGraphObject
      javax.media.j3d.NodeComponent
      javax.media.j3d.Material
```

960 4.2 Texturen

965 Die Vielfalt an Farben und Mustern in den verschiedenen Objekten der realen Welt entsteht überwiegend durch eine Unmenge an feinen Strukturen. Das beginnt bei einfachen Steinen wie z.B. dem Granit, der aus sehr vielen, winzigen und unterschiedlich gefärbten Kristallen besteht und endet noch lange nicht bei so ganz alltäglichen Dingen wie z.B. Polstermöbeln, die nicht nur eine interessante, aufwändige Form, sondern auch einen bunt gemusterten Bezug haben können.

970 Wollte man versuchen, diese hochkomplexen Strukturen originalgetreu wiederzugeben, wäre das Ergebnis zwar sicher atemberaubend realistisch, der daraus resultierende Ressourcenverbrauch wäre jedoch indiskutabel hoch. Mehr als Standbilder wären bei selbst bei nur etwas komplexeren Szenen kaum noch zu erwarten.

975 Eine Lösung für dieses Problem findet sich in den so genannten Texturen. Diese sind nichts als ganz normale Bilder. Bilder allerdings von den Oberflächen der Objekte, die dargestellt werden sollen. Da diese Texturen nun bereits den optischen Eindruck beinhalten, den wir von den gewünschten feinen Strukturen haben, können die eigentlichen 3D-Objekte, auf die sie dann gelegt werden sollen, deutlich einfacher ausfallen. Einfacher heißt hier, es sind deutlich weniger Polygone nötig, um diese 3D-Objekte darzustellen. Das wiederum schont die Ressourcen und ermöglicht höhere Frameraten.

Für die ersten Experimente mit Texturen ist ein neuer Import fällig, der wie immer am Anfang des Programmes erfolgt:

```
985 import com.sun.j3d.utils.image.*;
```

Die altbekannte Methode zur Erzeugung des SceneGraph wächst nun wiederum etwas an und bietet diesmal gleich drei Neuerungen:

990

```
(1)public BranchGroup createSceneGraph()  
(2)  {  
(3)    BranchGroup      RootBG=new BranchGroup();  
(4)    TransformGroup   SphereTG=new TransformGroup();  
995 (5)    Transform3D     SphereT3D=new Transform3D();  
(6)    Appearance       SphereAppearance=new Appearance();  
(7)  
(8)    AmbientLight     ALgt=new AmbientLight(new Color3f(1f,1f,1f));  
1000 (9)    DirectionalLight DLgt=new DirectionalLight(new Color3f(1f,1f,1f),new  
      Vector3f(-0.5f,-0.5f,-1f));  
(10)   BoundingSphere   BigBounds=new BoundingSphere(new Point3d(),100000);  
(11)  
(12)   ALgt.setInfluencingBounds(BigBounds);  
(13)   DLgt.setInfluencingBounds(BigBounds);  
1005 (14)   SphereT3D.setTranslation(new Vector3f(0f,0f,-1.5f));  
(15)   SphereTG.setTransform(SphereT3D);  
(16)   SphereAppearance.setMaterial(new Material(new Color3f(0f,0f,1f),new  
      Color3f(0f,0f,0f),new Color3f(1f,0f,0f),new Color3f(1f,1f,1f),100f));  
(17)  
1010 (18)   SphereAppearance.setTexture((new  
      TextureLoader("tiger.jpg",null)).getTexture());  
(19)   SphereAppearance.setTexCoordGeneration(new  
      TexCoordGeneration(TexCoordGeneration.SPHERE_MAP,TexCoordGeneration.TEXTURE_C  
      OORDINATE_2));  
1015 (20)   SphereAppearance.setTextureAttributes(new  
      TextureAttributes(TextureAttributes.REPLACE,new Transform3D(),new  
      Color4f(),TextureAttributes.NICEST));  
(21)  
(22)   SphereTG.addChild(new  
1020   Sphere(1.0f,Sphere.GENERATE_NORMALS,40,SphereAppearance));  
(23)   RootBG.addChild(SphereTG);  
(24)   RootBG.addChild(ALgt);  
(25)   RootBG.addChild(DLgt);  
(26)   RootBG.compile();  
1025 (27)   return RootBG;  
(28)   }
```

Diese Neuerungen sind ab Zeile 18 zu finden. Die erste neue Appearance-Methode spricht eigentlich für sich selbst: `setTexture()` setzt ganz offensichtlich eine Textur.

1030 Interessanter ist hier jedoch erst einmal, wie diese Textur erzeugt wird.

4.2.1 TextureLoader

Der neu hinzugefügte Import war nötig, damit der TextureLoader verwendet werden kann.
1035 Dieser dient dazu – wie der Name eigentlich klar sagt – eine Textur bzw. ein Bild, das als solche verwendet werden soll, zu laden. Als Parameter des Konstruktors wird deswegen auch der Name der Bilddatei angegeben, die geladen werden soll. Das ist im Fall des zugehörigen Paketes mit den Beispielscodes und -dateien zwar kein eher langweiliges Muster von einer Ziegelwand oder ähnlichem, wie es bei Texturen, die Strukturen und
1040 Oberflächen darstellen sollen, verwendet werden würde, es erfüllt seinen Zweck für dieses Beispiel aber trotzdem.

Die Klasse TextureLoader besitzt einige Konstruktoren mehr als nur den im Beispielprogramm verwendeten. Diese erfüllen jeweils den gleichen Zweck, der
1045 Unterschied besteht im wesentlichen aus der Art der Datenquelle. Hier können neben dem Dateinamen auch ein URL, ein Image oder ein BufferedImage verwendet werden:

```
TextureLoader(BufferedImage bImage)
TextureLoader(Image image, Component observer)
1050 ...
```

Möglich ist ebenfalls ein Flag **Y_UP** für den Parameter `flags`, dass die Orientierung einer Textur auf den Kopf stellt, also die Ausrichtung entlang der Y-Achse verändert. Das bezieht sich auf die Texturkoordinaten, die im folgenden Abschnitt näher beleuchtet
1055 werden.

```
TextureLoader(String fname, int flags, Component observer)
TextureLoader(URL url, int flags, Component observer)
...
1060
```

Die zur Verfügung gestellten Methoden des TextureLoaders sind recht übersichtlich, mehr ist hier allerdings auch nicht notwendig:

```
Texture getTexture()
```

1065 Diese Methode gibt ein Texture-Objekt zurück, das unter Verwendung der angegebenen Datenquelle erzeugt wurde
Sie wurde auch im Beispielscode verwendet, wobei die Datenquelle hier das Bild ist, das als Datei vorliegt.

```
1070 ImageComponent2D getImage()
```

Es wird ein ImageComponent2D-Objekt zurückgegeben, das unter Verwendung der angegebenen Datenquelle erzeugt wurde; Sinn und Zweck dieses Objekttyps werden später noch erklärt.

1075 `ImageComponent2D getScaledImage(float xScale, float yScale)`

Diese Methode liefert ebenfalls ein `ImageComponent2D`-Objekt zurück, allerdings wird dieses zuvor in Breite und in Höhe um jeweils die Faktoren `xScale` und `yScale` skaliert.

1080 `ImageComponent2D getScaledImage(int width, int height)`

Auch diese Methode liefert ein skaliertes `ImageComponent2D`-Objekt zurück. Im Gegensatz zur vorhergehenden Methode werden hier aber nicht die Skalierungsfaktoren als Parameter erwartet, sondern die gewünschte Breite `width` und Höhe `height` des skalierten Images.

1085

Ähnlich übersichtlich wie die Anzahl der Methoden sind auch die Ableitungsverhältnisse der Klasse `TextureLoader`:

```
java.lang.Object
1090     com.sun.j3d.utils.image.TextureLoader
```

4.2.2 TexCoordGeneration

Wer in Sachen 3D nicht vorbelastet ist, dürfte das `TexCoordGeneration` Objekt, dass mit der Methode, die den wenig überraschenden Namen `setTexCoordGeneration()` trägt, an das `Appearance`-Objekt übergeben wird, sicher am mysteriösesten finden. Dabei erschließt sich der Sinn dieses Objektes bei näherer Betrachtung recht einfach. Um eine Textur auf ein Objekt legen zu können, muß die 3D-Engine auch wissen, wie sie das tun soll. Sie muß also klare Anweisungen erhalten, in welcher Lage, Ausrichtung und Position das Bild, das als Textur verwendet wird, auf das Objekt „gelegt“ werden soll. Das geht am besten mit speziellen Koordinaten, den Texturkoordinaten.

Um das vielleicht etwas zu verdeutlichen: Denken Sie an einen ganz normalen Tisch, der das 3D-Objekt darstellt. Eine Tischdecke soll nun die Textur darstellen. Es gibt hier praktisch unendlich viele Variationsmöglichkeiten, wie man die Tischdecke flach auf den Tisch legen kann: parallel zu den Tischkanten, um 90° versetzt, so das ein Teil seitlich herunterhängt, kreuz, quer, sonst wie schief oder sogar verkehrt herum, also mit der Unterseite nach oben. Die Anweisung, wie die Tischdecke korrekt zu liegen hat, wird Ihnen in diesem Beispiel Ihre Frau geben, die 3D-Engine erfährt es von den Texturkoordinaten (wenn Sie selbst eine sind, werden Sie natürlich keinen Hinweis benötigen sondern es sicher selbst bemerken).

Dateiformate von 3D-Modellen beinhalten zu diesem Zweck eben jene Texturkoordinaten (die oft auch als U- und V-Koordinaten bezeichnet werden). Da in diesem Beispiel aber ein dynamisch erzeugtes Objekt zum Einsatz kommt, sind diese Koordinaten nicht vorhanden und müssen demzufolge erst irgend wie generiert werden. Das geschieht - richtig! - mit eben jenem `TexCoordGeneration`-Objekt, das in Zeile 19 der `Appearance`

zugewiesen wird.

1120 Auch hier werden alle wesentlichen Parameter wieder mit dem Konstruktor übergeben:

```
TexCoordGeneration(int genMode, int format)
```

1125 Der Parameter `genMode` legt hier fest, welche Art von Textur-Koordinaten erzeugt werden sollen. Da es sich bei dem verwendeten Objekt um eine Kugel handelt, fällt die Wahl leicht und auf **SPHERE_MAP**, was Koordinaten für eine sphärische Projektion erzeugt. Weitere Alternativen wären hier gewesen:

- **OBJECT_LINEAR** (die Texturkoordinaten werden als lineare Funktion in Objekt-Koordinaten erzeugt)
- 1130 - **EYE_LINEAR** (die Koordinaten für die Textur werden als lineare Funktion in Eye-Koordinaten erzeugt)
- **NORMAL_MAP**
- **REFLECTION_MAP**

1135 Der zweite Parameter, `format`, legt fest, um welche Art von Textur es sich hier handelt:

- **TEXTURE_COORDINATE_2** (eine zweidimensionale Textur mit den Texturkoordinaten S und T, oftmals auch als U und V bezeichnet)
- **TEXTURE_COORDINATE_3** (eine dreidimensionale Textur, hier werden jeweils die Texturkoordinaten für S, T und R erzeugt)
- 1140 - **TEXTURE_COORDINATE_4** (eine vierdimensionale Textur mit S, T, R, und Q als Koordinaten)

1145 Bei den zur Verfügung gestellten Methoden sind wieder verschiedene Möglichkeiten vorhanden, die bereits mit dem Konstruktor übergebenen Werte zu holen bzw. sie auf andere Werte zu setzen. Daneben existieren noch Funktionalitäten, die Einfluß auf die Texturkoordinaten haben (die hier Q, R, S und T heißen). Sinn und Funktion werden in späteren Kapiteln näher beschrieben, der Vollständigkeit halber sollen sie hier an dieser Stelle aber trotzdem erwähnt werden:

1150 `boolean getEnable()`
`void setEnable(boolean state)`

Ein `TexCoordGeneration`-Objekt kann aktiviert oder deaktiviert sein. Dieser Zustand läßt sich mittels dieser Methoden in Erfahrung bringen oder verändern.

1155 `int getFormat()`
`void setFormat(int format)`

Mit diesen Methoden ist es möglich, das aktuelle, bei den Konstruktoren bereits angesprochene Format der Textur zu ermitteln oder mit einer der Konstanten

TEXTURE_COORDINATE_x neu zu setzen.

1160

```
int getGenMode()  
void setGenMode(int genMode)
```

1165 Auch hier dreht es sich um eine der bei den Konstruktoren bereits angesprochenen Eigenschaften. Diese Methoden liefern den aktuellen Wert für den Generation-Modus (also die Beschreibung, wie die Koordinaten erzeugt werden sollen) zurück bzw. ermöglicht es, diesen auf einen neuen Wert zu setzen.

```
void getPlaneQ(Vector4f planeQ)  
void getPlaneR(Vector4f planeR)  
1170 void getPlaneS(Vector4f planeS)  
void getPlaneT(Vector4f planeT)  
void setPlaneQ(Vector4f planeQ)  
void setPlaneR(Vector4f planeR)  
void setPlaneS(Vector4f planeS)  
1175 void setPlaneT(Vector4f planeT)
```

1180 Diese Methoden stehen – wie bereits erwähnt – in direktem Zusammenhang mit den erzeugten Texturkoordinaten. Die `get`-Methoden liefern dabei jeweils wieder die aktuellen Werte zurück, die sie in das übergebene `Vector4f`-Objekt kopieren. Anders herum kopieren die `set`-Methoden die als Parameter übergebenen `Vector4f`-Objekte in das `TexCoordGeneration`-Objekt. Verwendet werden diese so genannten „Plane Equation“-Werte allerdings nur in den Generation-Modi **OBJECT_LINEAR** und **EYE_LINEAR**, andernfalls werden sie ignoriert.

1185 An dieser Stelle sei noch erwähnt, dass der Weg über das `TexCoordGeneration`-Objekt hier eigentlich ein Umweg war, der nicht nötig gewesen wäre. Statt dessen gäbe es die wesentlich einfachere Methode, die benötigten Texturkoordinaten gleich bei der Erzeugung der Kugel mit anlegen zu lassen. Das ist mit Hilfe des Parameters `primflags` möglich und hätte für das Beispiel oben für die Konstruktion der Sphere dann so ausgesehen:

1190

```
new Sphere(1.0f, Sphere.GENERATE_NORMALS |  
Sphere.GENERATE_TEXTURE_COORDS, 40, SphereAppearance)
```

1195 Wie schon der Name des jetzt neu hinzugekommenen Flags **GENERATE_TEXTURE_COORDS** verrät, bewirkt es, dass zusammen mit der Kugel die dazu passenden Texturkoordinaten generiert werden, die dann natürlich sofort und ohne zusätzliche Klimmzüge zur Verfügung stehen.

Die „Verwandtschaftsverhältnisse“ bieten indes wieder wenig überraschendes:

1200

```
java.lang.Object
    javax.media.j3d.SceneGraphObject
        javax.media.j3d.NodeComponent
            javax.media.j3d.TexCoordGeneration
```

1205

4.2.3 TextureAttributes

Das TextureAttributes-Objekt, das in Zeile 20 mit der Methode `setTextureAttributes()` gesetzt wird, wäre an dieser Stelle eigentlich auch nicht nötig gewesen. Mit den Default-Werten, die von Appearance angenommen werden, wäre die Textur auch ohne diesen Umweg in einer vernünftigen Darstellung auf der Kugel sichtbar geworden. Allerdings finden sich hier zwei Elemente, die die Qualität von Java 3D unterstreichen und wirklich interessante Features ermöglichen. Doch immer der Reihe nach.

1210

1215

Auch an dieser Stelle wird im Beispielcode alles – der Schreibfaulheit halber – bereits mit dem Konstruktor erschlagen:

```
TextureAttributes(int textureMode, Transform3D transform, Color4f
1220 textureBlendColor, int perspCorrectionMode)
```

Hier ist mit `textureMode` nicht nur der erste wichtige Parameter sondern auch das erste, bereits angekündigte interessante Element zu finden. Mit diesem Wert wird festgelegt, wie die Textur auf dem Objekt dargestellt werden soll. Im Beispiel wurde **REPLACE** verwendet, was schlichtweg alle zuvor mühselig erzeugten Farb- und Materialinformationen des texturierten Objektes überschreibt. Vergleichbar mit Emissive Color wäre die Textur also auch zu sehen, wenn das Universum komplett dunkel wäre. Das lässt sich ganz leicht überprüfen, in dem einfach die Zeilen 24 und 25 auskommentiert werden und somit das Umgebungslicht entfernt wird. Dementsprechend wirkt die Kugel bei der Verwendung dieses Texture-Modus auch wieder ausgesprochen flach.

1225

1230

Neben **DECAL**, **BLEND** und **COMBINE** sei hier besonders auf **MODULATE** als alternativen Wert für den `textureMode`-Parameter hingewiesen. Dieser beseitigt das Problem, das die Materialeigenschaften nicht mehr zum tragen kommen dadurch, dass die Farbe der Textur für jeden dargestellten Pixel des Objektes mit der Farbe des Objektes (festgelegt durch die Materialeigenschaften) moduliert wird. Das führt dazu, dass das Objekt nicht mehr flach, sondern wieder rund aussieht und dass die Textur – besonders bei den im obigen Beispiel recht außergewöhnlich gewählten Materialfarben – in völlig neuen Farben erstrahlt. Diese interessante, fast schon psychedelische Mischung ergibt sich eben aus der mittels **MODULATE** aktivierten Modulation von Objekt- und Texturfarben.

1235

1240

Der zweite Parameter, `transform`, erwartet einen alten Bekannten: ein `Transform3D`-Objekt. Mit diesem ist es möglich, die Textur in ihrer Lage auf dem Objekt zusätzlich zu den eigentlich verwendeten Texturkoordinaten noch einmal zu transformieren, d.h. Zu

1245

verschieben, zu rotieren und zu skalieren. Auch an dieser Stelle lohnt es sich, ein wenig mit den Möglichkeiten zu experimentieren: wird ein separates Transform3D-Objekt erzeugt und in einer Variablen abgelegt, so kann mit dieser eine Rotation, eine Translation und/oder eine Skalierung für die Textur bewirkt werden.

1250

Auf die Existenz des Parameters `textureBlendColor` soll an dieser Stelle nur kurz eingegangen werden, da dieser zu den deutlich fortgeschritteneren Techniken gehört. Dieser Wert ermöglicht weitere Manipulationen an der Textur, die hier im Zusammenhang mit den `textureModes` **BLEND** und **COMBINE** erfolgen.

1255

`perspCorrectionMode` bringt nun die zweite, bereits lauwarm angekündigte Neuerung. Die wichtigsten, hier möglichen Werte sind **FASTEST** und **NICEST**. Wie am Namen schon fast zu erkennen, legen sie die Qualität der Darstellung fest. Da diese Möglichkeit auch bei anderen Elementen, die im Appearance-Objekt gesetzt werden können, besteht, ermöglicht das eine ziemlich feingranulare Skalierung der 3D-Umgebung. Dabei gilt immer: um so höher die Qualität (**NICEST**) um so niedriger ist die verbleibende Geschwindigkeit, die sich in der Framerate niederschlägt. Andersherum gilt demzufolge, um so höher die gewünschte Geschwindigkeit (**FASTEST**), um so geringer die erzielte Qualität in der Darstellung.

1265

Für Texturen bedeutet ein schnellerer und damit qualitativ schlechterer `perspCorrectionMode` mitunter durchaus eine recht stark gekrümmte und verzerrte Abbildung der Textur.

1270 Die Methoden, die von der Klasse `TextureAttributes` zur Verfügung gestellt werden, seien auch hier wieder beschrieben:

```
int getPerspectiveCorrectionMode()  
void setPerspectiveCorrectionMode(int mode)
```

1275 Diese Methoden erlauben es, den eben angesprochenen Wert für die perspektivische Korrektur zu ermitteln bzw. ihn unter Verwendung von **FASTEST** oder **NICEST** neu zu setzen.

```
void getTextureBlendColor(Color4f textureBlendColor)  
1280 void setTextureBlendColor(Color4f textureBlendColor)
```

Diese Methoden gehören zur ebenfalls bereits angesprochenen Blend Color für den Modus **BLEND** und erlauben es, den aktuellen Farbwert zu holen (wobei dieser in das übergebene `Color4f`-Objekt kopiert wird) oder ihn auf einen anderen Wert zu setzen.

```
1285 int getTextureMode()  
void setTextureMode(int textureMode)
```

Diese Methoden beziehen sich auf den Modus, in dem texturiert werden soll. Die möglichen Konstanten wie **BLEND**, **DECAL**, **MODULATE** und **COMBINE** werden von

diesen zurückgeliefert bzw. es kann der Modus auf einen anderen Wert gesetzt werden.

1290

```
void getTextureTransform(Transform3D transform)
```

```
void setTextureTransform(Transform3D transform)
```

1295 Hiermit wird die aktuell für die Texturierung verwendete Transformation ermittelt, in dem sie in das übergebene Transform3D-Objekt kopiert wird, oder sie wird neu gesetzt, in dem sie vom übergebenen Transform3D-Objekt in das TextureAttributes-Objekt kopiert wird.

Abschließend nur noch ein kurzer Blick auf die ebenfalls nicht mehr überraschende Ableitung der Klasse TextureAttributes:

1300

```
java.lang.Object
```

```
    javax.media.j3d.SceneGraphObject
```

```
        javax.media.j3d.NodeComponent
```

```
            javax.media.j3d.TextureAttributes
```

1305

4.2.4 Texture

1310 Auch wenn im letzten Beispielprogramm das Texture-Objekt ohne weitere Behandlung vom TextureLoader geholt und sofort dem Appearance-Objekt übergeben wurde, soll hier noch ein Blick auf diese Klasse geworfen werden, da sie einige interessante Funktionalitäten bietet, die nicht nur für fortgeschrittene Anwendungen von Interesse sein können.

1315 Da die Klasse Texture abstrakt ist, ist es nicht möglich, einen ihrer Konstruktoren direkt zu verwenden. Das geht nur bei einer der direkt abgeleiteten Klassen Texture2D oder Texture3D. Alternativ dazu gibt es natürlich auch die bereits bekannte Möglichkeit, ein Texture-Objekt von z.B. dem TextureLoader erzeugen zu lassen.

1320 Mehr Möglichkeiten bieten die zur Verfügung gestellten Methoden, die demzufolge auch von den Klassen Texture2D und Texture3D geerbt werden. Die für den Anfang wichtigsten sind hier:

```
public void setMagFilter(int magFilter)
```

```
public int getMagFilter()
```

1325 Setzt oder holt die so genannte Magnification Filter Function. Diese beschreibt, wie die Engine verfahren soll, wenn eine Textur auf Grund ihrer eigenen effektiven Größe und der darzustellenden Größe in der Szene vergrößert werden muß. Bereits bekannte Möglichkeiten sind hier wieder **FASTEST** und **NICEST**, die jeweils für eine hohe Darstellungsgeschwindigkeit bzw. für eine hohe Qualität stehen. Neu in Java 3D Version 1.3.x ist **FILTER4**. Damit ist es möglich, eine eigene Filterfunktion zu definieren, die dann
1330 mittels `public void setFilter4Func(float[] weights)` an das Textur-Objekt

übergeben werden muss.

```
public void setMinFilter(int magFilter)
```

```
1335 public int getMinFilter()
```

Setzt oder holt den Wert für die gewählte Minification Filter Function. Diese ist identisch mit der vorhergehenden Methode. Allerdings wird hier die Filterfunktion für den Fall gesetzt, dass eine Textur innerhalb der Szene verkleinert werden muss.

1340 Die Klasse Texture selbst leitet sich recht einfach ab:

```
java.lang.Object
```

```
    javax.media.j3d.SceneGraphObject
```

```
        javax.media.j3d.NodeComponent
```

```
1345             javax.media.j3d.Texture
```

4.3 Durchsichtiges

1350 Mit den bisher vorgestellten Möglichkeiten, das Aussehen von 3D-Objekten zu verändern und somit bestimmte Materialien darzustellen, lässt sich schon einiges anfangen. Allerdings fehlt hier noch etwas wichtiges, es ist mit den bisher bekannten Möglichkeiten nicht machbar, ein Objekt so zu gestalten, dass man durch dieses auch hindurchsehen kann. Das damit aber nicht nur transparente Objekte gemeint sind, soll die hierfür wiederum etwas erweiterte Methode des Beispielprogrammes zeigen:

1355

```
(1)public BranchGroup createSceneGraph()
```

```
(2)    {
```

```
(3)        BranchGroup      RootBG=new BranchGroup();
```

```
(4)        TransformGroup   SphereTG=new TransformGroup();
```

```
1360 (5)        Transform3D     SphereT3D=new Transform3D();
```

```
(6)        Appearance       CylinderAppearance=new Appearance(),SphereAppearance=new  
        Appearance();
```

```
(7)
```

```
(8)        AmbientLight     ALgt=new AmbientLight(new Color3f(1f,1f,1f));
```

```
1365 (9)        DirectionalLight DLgt=new DirectionalLight(new Color3f(1f,1f,1f),new  
        Vector3f(-0.5f,-0.5f,-1f));
```

```
(10)       BoundingSphere   BigBounds=new BoundingSphere(new Point3d(),100000);
```

```
(11)       ALgt.setInfluencingBounds(BigBounds);
```

```
(12)       DLgt.setInfluencingBounds(BigBounds);
```

```
1370 (13)       SphereT3D.setTranslation(new Vector3f(1f,0f,-1.5f));
```

```
(14)       SphereTG.setTransform(SphereT3D);
```

```

(15)
(16) SphereAppearance.setMaterial(new Material(new
1375 Color3f(0.1f,0.1f,0.1f),new Color3f(0f,0f,0f),new Color3f(0.8f,0.8f,0.8f),new
Color3f(0.6f,0.6f,0.6f),1f));
(17) SphereAppearance.setPolygonAttributes(new
PolygonAttributes(PolygonAttributes.POLYGON_LINE,PolygonAttributes.CULL_NONE,
0));
(18) SphereTG.addChild(new
1380 Sphere(0.75f,Sphere.GENERATE_NORMALS,30,SphereAppearance));
(19) RootBG.addChild(SphereTG);
(20)
(21) CylinderAppearance.setMaterial(new Material(new Color3f(0f,0f,1f),new
Color3f(0f,0f,0f),new Color3f(1f,0f,0f),new Color3f(1f,1f,1f),100f));
1385 (22) CylinderAppearance.setTransparencyAttributes(new
TransparencyAttributes(TransparencyAttributes.NICEST,0.6f));
(23) RootBG.addChild(new
Cylinder(0.5f,1f,Cylinder.GENERATE_NORMALS,40,1,CylinderAppearance));
(24)
1390 (25) RootBG.addChild(ALgt);
(26) RootBG.addChild(DLgt);
(27)
(28) RootBG.compile();
(29) return RootBG;
1395 (30) }

```

Dieses mal wird eine Szene mit zwei Objekten erstellt, von denen eines wieder die bekannte Kugel und das zweite ein neues Primitive ist: Ein Zylinder. Das Einfachste wieder vornweg - ein Blick auf den Konstruktor von Cylinder:

```

1400 Cylinder(0.5f,1f,Cylinder.GENERATE_NORMALS,40,1,CylinderAppearance)

```

Dieser hat starke Ähnlichkeit mit dem Konstruktor der Klasse Sphere. Der erste Wert gibt den Radius des Zylinders an, der zweite legt dessen Höhe fest. Beide beziehen sich wieder auf die Maßeinheit Meter. Als dritter Parameter können wiederum Flags angegeben werden, die bei der Erzeugung des 3D-Objektes benötigt werden. Der vierte und der fünfte Parameter legt wieder – wie auch bei der Sphere – die Auflösung des Objektes fest, wobei je ein Wert für die Auflösung der Grundfläche, also des Kreises steht und ein Wert für die Höhe, also die Anzahl Unterteilungen, die der Mantel des Zylinders haben soll. Wird für die Auflösung der Grundfläche, die oben mit 40 festgelegt wird, eine 4 angegeben, ist das Ergebnis kein Zylinder mehr, sondern ein Quader mit einer quadratischen Grundfläche.

Der letzte als Parameter zu übergebende Wert ist wiederum ein Appearance-Objekt, das im folgenden näher beleuchtet werden soll.

```

1415
Wird das Programm mit der modifizierten createSceneGraph() -Methode ausgeführt,

```

zeigt sich ein völlig neues Bild. Der Zylinder im Vordergrund erscheint in den von der Kugel her bekannten Farben. Allerdings ist er durchsichtig, so dass die Kugel, die sich hinter ihm befindet, durch ihn hindurch zu sehen ist.

1420

Die Kugel selber ist ebenfalls in einer anderen als der gewohnten Darstellung zu sehen, sie wird als so genanntes Drahtgittermodell sichtbar.

4.3.1 PolygonAttributes

1425

Die veränderte Darstellung der Sphere wurde mit Hilfe einer neuen Klasse erreicht, den PolygonAttributes. Ein entsprechendes Objekt wurde in altbekannter Weise dem Appearance-Objekt zugeordnet, so dass sich die Darstellung des 3D-Objektes auch entsprechend verändert.

1430

Da sich die Kugel nun nicht mehr aus geschlossenen Polygonen zusammensetzt, sondern nur noch aus den Verbindungslinien zwischen den Ecken der einzelnen Polygone (den so genannten Vertices) besteht, ist auch sie durchsichtig geworden. Das ist in der Szene, so wie sie momentan aufgebaut ist, nicht sofort zu sehen. Würde man allerdings ein weiteres Objekt hinter der Kugel platzieren, würde das deutlich werden.

1435

Die Beschreibung des Konstruktors soll hier wiederum erhellen, was mit dem Objekt passiert ist:

1440 `PolygonAttributes(int polygonMode,int cullFace,float polygonOffset)`

Bereits der erste Parameter legt fest, wie das Objekt dargestellt werden soll. Hier stehen zur Auswahl:

1445

- **POLYGON_FILL** – Die Polygone, aus denen sich das 3D-Objekt zusammen setzt, sollen in gewohnter Weise dargestellt werden, also als ausgefüllte Polygone. Das führt zu einer Darstellung, die das 3D-Objekt als festen, massiven Körper erscheinen lässt (wobei mit „massiv“ hier nicht gemeint ist, dass das 3D-Objekt auch wirklich gefüllt ist, Volumen wird von der Java-3D-Engine nicht ohne weitere programmtechnische Maßnahmen unterstützt). Dieser Wert ist voreingestellt und wird immer dann verwendet, wenn der Default-Konstruktor verwendet wird oder wenn einem Appearance-Objekt kein spezifisches PolygonAttributes-Objekt zugewiesen wird.

1450

- **POLYGON_LINE** – Dieser Wert wurde im obigen Beispiel verwendet, was dazu geführt hat, dass nur noch die Verbindungslinien zwischen den einzelnen Vertices gezeichnet werden. Das Resultat ist eine so genannte Drahtgitterdarstellung des 3D-Objektes. Statt ausgefüllter Polygone werden nur noch deren Umrandungen gezeichnet.

1455

- **POLYGON_POINT** – Wird dieser Wert übergeben, wird noch weniger gezeichnet. Hier bleibt nurmehr eine Darstellung der Vertex-Koordinaten in Form von einzelnen Punkten übrig. Das Ergebnis ist eine sogenannte Punktwolke.

1460 Der Sinn des zweiten Parameters erschließt sich nicht so ohne weiteres, da hier einige Eigenschaften der digitalen 3D-Welt betrachtet werden müssen. Wie schon angedeutet wurde, setzt sich ein Objekt eigentlich nur aus einer Ansammlung von Polygonen zusammen. Diese Polygone sind typischerweise Dreiecke, es können aber auch Vierecke oder unter bestimmten, später noch näher zu betrachtenden Voraussetzungen andere

1465 Vielecke zum Einsatz kommen. Was macht nun so ein Polygon aus? Es besteht eigentlich nur aus mehreren Koordinaten und einer Vorschrift, wie diese beim Zeichnen zu verbinden sind, so dass die gewünschte Form entsteht.

Und hier wird es interessant: Die Java-3D-Engine bzw. die darunterliegende Grafikschnittstelle (meist OpenGL, DirectX oder GLX) führen ein so genanntes Face-Culling durch. Übersetzen könnte man das so, dass bestimmte „Faces“ (also Polygone) entfernt werden. Genauer gesagt: es kann eine Seite entfernt werden. Betrachtet man so ein Polygon, von dem z.B. die Rückseite entfernt wurde von vorne, so sieht es ganz normal aus. Von hinten jedoch ist es schlichtweg nicht zu sehen. Diese Tatsache sollte man beim Erstellen oder

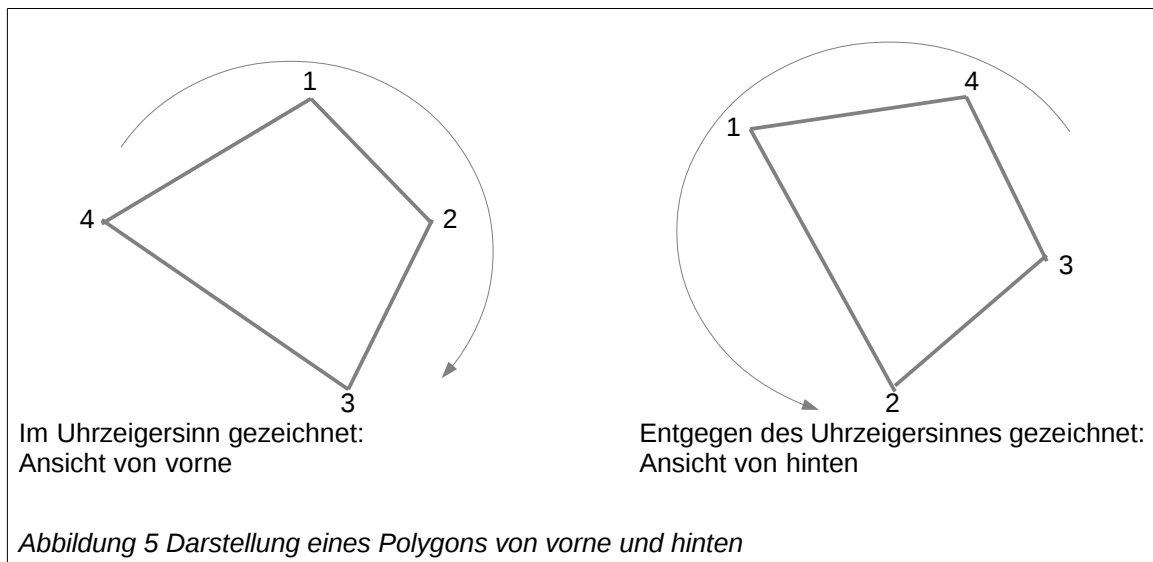
1475 Verarbeiten von 3D-Objekten immer im Hinterkopf behalten. Da so eine Verhaltensweise für Objekte der realen Welt nicht zu erwarten ist, kann das in einer 3D-Welt zu recht unerwarteten und überraschenden Effekten führen.

Doch zurück zum Parameter `cullFace`. Hier können folgende Werte übergeben werden:

- 1480
- **CULL_NONE** – Es wird kein Face-Culling durchgeführt, so dass die Polygone des zugehörigen 3D-Objektes von beiden Seiten aus sichtbar ist.
 - **CULL_BACK** – Es wird ein Backface-Culling durchgeführt, die Rückseite wird also nicht gezeichnet und das Objekt, wenn man es aus rückwärtiger Richtung betrachtet, unsichtbar.
 - 1485 - **CULL_FRONT** – Das Gegenstück zu **CULL_BACK**, hier werden die Polygone von vorne betrachtet unsichtbar und das 3D-Objekt ist nur bei einer Beobachterposition innerhalb selbigens wirklich zu sehen.

Wo bei einem Polygon nun „vorne“ und „hinten“ ist, legt sich übrigens wieder durch die Art

1490 fest, in der es erzeugt wird: Spezifiziert die Beschreibung, wie das Polygon gezeichnet werden soll aus Sicht des Betrachters eine Reihenfolge, bei der die Verbindung der einzelnen Vertices im Uhrzeigersinn erfolgt, so sieht besagter Betrachter gerade die Vorderseite des Polygons.



Der letzte Parameter des obigen PolygonAttributes-Konstruktors, `polygonOffset`, ist an dieser Stelle nicht weiter wichtig und wurde deswegen auf 0 gesetzt. Kurz zur Erläuterung für diejenigen, die jetzt schon wissen wollen, was dieser Parameter macht, ein Auszug aus der Java 3D Spezifikation: Die Tiefenwerte aller Pixel, die durch Polygonrasterization erzeugt werden, können hiermit durch einen Wert versetzt werden, der für dieses Polygon berechnet wird.

Dem Appearance-Objekt wurde das neu erzeugte Objekt mit der Methode `setPolygonAttributes()` zugewiesen, auch diese folgt wieder dem gleichen Prinzip der Namensgebung, bei der die Bezeichnung des zu setzenden Objekttyps im Namen enthalten ist.

Die PolygonAttributes bieten darüber hinaus auch die Möglichkeit, diese und andere Eigenschaften mit Hilfe diverser Methoden zu ermitteln oder aber die zugehörigen Werte zu ändern:

```
boolean getBackFaceNormalFlip()
```

```
void setBackFaceNormalFlip(boolean backFaceNormalFlip)
```

Diese Methoden beziehen sich auf die so genannten Normals, auf welche später noch detaillierter eingegangen werden wird. Hier nur kurz so viel, dass diese für die Beleuchtung von 3D-Objekten bzw. Flächen wichtig sind. Und genau dort greift das Backface-Normal-Flipping ein. Für die Rückseite eines Polygons wird die Darstellung der beleuchteten Fläche und damit der helligkeitsabhängigen Schattierung selbiger geändert, wenn dieser Wert `true` ist. Das kann wichtig werden, wenn Hohlkörper von innen sichtbar werden sollen und auch von innen beleuchtet werden.

Diese beiden Methoden erlauben es nun, den aktuellen Wert für das Backface-Normal-Flipping zu ermitteln bzw. dieses zu aktivieren (`true`) oder zu deaktivieren (`false`).

```
int getCullFace()
```

1525 `void setCullFace(int cullFace)`

Mit diesen Methoden ist es möglich, den aktuellen Wert für das Face-Culling zu ermitteln bzw. einen neuen festzulegen. Hier kommen die oben bereits beschriebenen Konstanten **CULL_NONE**, **CULL_BACK** und **CULL_FRONT** zum Einsatz.

1530 `int getPolygonMode()`

`void setPolygonMode(int polygonMode)`

Auch die Eigenschaft, auf die sich diese Methoden beziehen, wurden zusammen mit dem oben gezeigten Konstruktor bereits beschrieben. Diese Verwenden die Konstanten **POLYGON_FILL**, **POLYGON_LINE** sowie **POLYGON_POINT** legen fest, ob
1535 das zugehörige 3D-Objekt vollständig ausgefüllt gezeichnet werden oder ob ein Drahtgittermodell bzw. eine Punktwolke dargestellt werden soll.
Diese Methoden erlauben es, den aktuellen Polygon-Mode zu ermitteln bzw. einen neuen festzulegen.

1540 Auch `PolygonAttributes` leitet sich direkt von der Klasse `NodeComponent` ab:

```
java.lang.Object
    javax.media.j3d.SceneGraphObject
        javax.media.j3d.NodeComponent
1545             javax.media.j3d.PolygonAttributes
```

4.3.2 Transparency

Die zweite Möglichkeit, ein 3D-Objekt durchsichtig zu machen, ist eigentlich die, die der
1550 geneigte Leser bei der Einleitung zu diesem Abschnittes sicher als erstes erwartet hat: die Transparenz. Interessanterweise hat diese Objekteigenschaft, die sich relativ leicht und einfach definieren läßt, auch ihre Tücken. Dazu jedoch etwas später mehr. Zuerst wie immer ein Blick auf die Verfahrensweise, die den Zylinder durchsichtig gemacht hat und ihn damit als massives Glas erscheinen läßt.

1555

In alt bekannter Weise wird dem `TransparencyAttributes`-Objekt wieder alles nötige bei der Erzeugung mitgegeben und es dann mit Hilfe der Methode `setTransparencyAttributes()` dem `Appearance`-Objekt zugewiesen. Die Parameter des verwendeten Konstruktors sind recht übersichtlich aber ausreichend:

1560

```
TransparencyAttributes(int tMode, float tVal)
```

Für `tMode` wird wieder ein Wert **FASTEST** oder **NICEST** erwartet, der wie bereits von früheren Attribute-Klassen her bekannt die Qualität des Ergebnisses beeinflusst. Hier
1565 ließe sich alternativ auch **SCREEN_DOOR** oder **BLENDED** verwenden, die erstgenannten beiden Möglichkeiten stellen nur eine abstraktere Variante dar, die diese Konstanten 1:1 ersetzen. Eine weitere Variante ist mit der Konstanten **NONE** gegeben. Diese sorgt dafür, dass dieses `TransparencyAttributes`-Objekt festlegt, dass das zugehörige 3D-Objekt komplett undurchsichtig ist. Das führt dazu, dass der folgende

Parameter `tVal` recht unnütz wird. Wie Sie anschließend sehen werden, sollte diese Möglichkeit jedoch besser vermieden werden.

`tVal` legt fest, wie transparent das Objekt werden soll. 1.0 steht hier für ein vollständig transparentes (also faktisch unsichtbares) 3D-Objekt, während 0.0 dafür sorgt, dass es


```
void setTransparencyMode(int transparencyMode)
```

1615 Auch für den Transparenz-Modus, für den die Konstanten **FASTEST** und **NICEST** in der Regel die wichtigsten sind und die einen direkten Einfluß auf die Qualität der Darstellung haben, finden sich mit diesen beiden hier passende Methoden. Sie erlauben es, den aktuellen Wert im zugehörigen TransparencyAttributes-Objekt zu ermitteln oder aber einen neuen Wert festzulegen.

1620 Bevor nun endlich einige der möglichen Probleme angesprochen werden, die im Zusammenhang mit der Transparenz auftreten können, noch mal ein Blick auf die Ableitung der Klasse selbst:

```
1625 java.lang.Object
      javax.media.j3d.SceneGraphObject
      javax.media.j3d.NodeComponent
      javax.media.j3d.TransparencyAttributes
```

4.3.2.1 Transparenz und Z-Order

1630

Transparenzen sind für Grafikhardware und besonders für 3D-Engines mit eine der anspruchsvollsten Aufgaben. Das rührt daher, dass hier etwas mehr berechnet werden muss, als bei Szenen ausschließlich mit undurchsichtigen Objekten. Daraus resultieren auch die bereits angedeuteten, möglichen Probleme.

1635

Doch der Reihe nach. Um Transparenzen zu simulieren, ist es für die Darstellung nötig zu wissen, welche 3D-Objekte durch ein transparentes Objekt hindurch sichtbar sein sollen, um aus dessen Farbe und der Farbe des transparenten Objektes die resultierende Farbe zu errechnen. Das soll bei undurchsichtigen 3D-Objekten, die sich vor dem transparenten 3D-Objekt befinden, logischerweise nicht passieren. Dazu muss die Engine wissen, in welcher Reihenfolge sie vom Betrachter aus zu sehen sind.

1640

Diese Reihenfolge in der räumlichen Tiefe nennt man Z-Order entsprechend der absoluten Z-Achse einer 3D-Welt. Die Ermittlung dieser Z-Order für jede Position des Beobachters (die sich mitunter sehr schnell ändert) ist keine triviale Aufgabe.

1645

Und genau hier können Probleme entstehen. In den Java 3D Engines Version 1.2.x kam die Darstellung unter Umständen durcheinander. Reproduzierbar war das z.B. dadurch, dass undurchsichtige Objekte mit TransparencyAttributes und einem Transparenzwert von 1.0 erzeugt wurden. Ein Blick in den vorhergehenden Abschnitt zeigt, dass ein solcher Wert eigentlich in einem komplett undurchsichtigem Objekt resultieren müßte. Das tut er auch, lediglich mit der Darstellung können Probleme auftreten, spätestens dann, wenn sich andere, transparentere Objekte in der Szene befinden.

1650

1655 Hier konnte es passieren, dass die Darstellung sehr seltsam aussah, weil sich diese 3D-Objekte an den Stellen, an denen eines das andere verdecken sollten, überlappten – und

1660 zwar in umgekehrter Reihenfolge als es aus der realen Welt sein sollte – weiter hinten befindliche Teile waren plötzlich weiter vorne zu sehen, obwohl die perspektivische Darstellung dennoch so war, dass sie räumlich eigentlich weiter hinten angeordnet sind (was ja auch der Fall war).

1665 Deshalb gilt für die TransparencyAttributes: Sie sollten wirklich nur dann einem Appearance-Objekt hinzugefügt werden, wenn dieses auch wirklich transparent ist, niemals bei eigentlich komplett undurchsichtigen Objekten. Das heißt auch, dass bei 3D-Objekten, bei denen der Transparenzwert dynamisch veränderbar sein soll, das entsprechende TransparencyAttributes-Objekt gezielt hinzugefügt oder auch wieder entfernt werden sollte. Beim dynamischen Verändern von 3D-Objekten, die live (und kompiliert) sind, treten allerdings noch andere Schwierigkeiten auf, die in Zusammenhang mit den so genannten Capability Bits gelöst werden können. Doch dazu später mehr.

1670

4.3.2.2 Die OrderedGroup

1675 Eine weitere Möglichkeit, der 3D-Engine bei der Z-Order ein wenig zu helfen, findet sich in der Klasse OrderedGroup. Diese ist in vielen Dingen ähnlich einer BranchGroup, jedoch steht hier die Reihenfolge, in der 3D-Objekte und Sub-SceneGraphen als Child hinzugefügt werden in einem direkten Zusammenhang mit der Reihenfolge, in der sie auch gezeichnet werden. Um so weiter vorne ein 3D-Objekt oder Teil-SceneGraph sich in einer OrderedGroup befindet, um so eher wird es gezeichnet und um so weiter weg vom Beobachter sollte sich dieses demzufolge befinden um Überlappungen zu vermeiden, die 1680 nicht zur Darstellung der räumlichen Tiefe passen. Aus dieser Eigenschaft ergibt sich, dass die OrderedGroup in der Regel nur bei recht statischen Darstellungen verwendet werden sollte, bei der der Beobachter seine Position in Bezug auf die Objekte unterhalb so eines OrderedGroup-Objektes nicht ohne weiteres oder nur recht selten verändern kann. Dennoch empfiehlt es sich, die OrderedGroup immer da anzuwenden, wo das 1685 sinnvoll möglich ist, also wo deren Children nicht ständig umsortiert werden müssten um wieder zur aktuellen Darstellung zu passen. Denn auch wenn es keine Probleme mit der Transparenz gibt, entlastet die Verwendung dieser Klasse die 3D-Engine doch wesentlich. Das führt im Ergebnis zu einer höheren Framerate, was einem alten Grundsatz in der 3D-Echtzeitgrafik entgegenkommt: Framerate kann man nur durch eins ersetzen - durch 1690 noch mehr Framerate.

1695 Die Objektreihenfolge in einer OrderedGroup lässt sich auf zwei Arten festlegen. Bei der direkten Variante werden die Child-Objekte in der Reihenfolge gezeichnet, in der sie in der OrderedGroup positioniert wurden. Die indirekte Variante erlaubt es, mit Hilfe eines zusätzlichen int-Arrays festzulegen, welches Child der OrderedGroup wann gezeichnet werden soll. Hier legt das Array die Reihenfolge fest und verweist dabei aber auf die Position desjenigen Children, das gezeichnet werden soll. Anders gesagt enthält jeder Index dieses Arrays die Nummer des Child-Nodes, das gezeichnet werden soll. In diesem Fall wird der Node als erstes gezeichnet, der durch die Positionsnummer an Index 0 des 1700 Arrays festgelegt wird.

Voraussetzung für die Verwendung dieser indirekten Methode ist, dass dieses int-Array genau so viele Elemente enthält wie das OrderedGroup-Objekt Children. Wird ein Child aus einem OrderedGroup-Objekt entfernt, das so ein Index-Array besitzt, so wird dieses int-Array von der OrderedGroup automatisch in einer Weise geändert, die gewährleistet,

1705 dass die Daten konsistent bleiben.

Eine weitere Bedingung für das korrekte Funktionieren der OrderedGroup und auch der von ihr abgeleiteten DecalGroup ist, dass in den RenderingAttributes der Appearances einige Modifikationen vorgenommen werden. Hier müssen depthBufferEnable und
1710 depthBufferWriteEnable für diejenigen 3D-Objekte auf `false` gesetzt werden, für die die Rendering-Reihenfolge beeinflusst werden soll. Das ist mit den Methoden `setDepthBufferEnable()` und `setDepthBufferWriteEnable()` der Klasse `RenderingAttributes` möglich.

1715 Die Klasse `OrderedGroup` besitzt lediglich einen einzigen Konstruktor, der sicher nicht weiter erklärt werden braucht:

```
OrderedGroup()
```

1720 Bei den Methoden finden sich dementsprechend wesentlich mehr Möglichkeiten ein `OrderedGroup`-Objekt zu manipulieren:

```
void addChild(Node child)
```

Diese Methode fügt ein neues Child-Objekt an das Ende der Klassen-internen Liste an. Das bedeutet, dass die zugehörigen 3D-Objekte des als Parameter übergebenen Sub-SceneGraphen anschließend als letztes gezeichnet werden.
1725 Ist das `OrderedGroup`-Objekt Teil eines `SceneGraphen`, der live oder kompiliert ist, so muß das übergebene Objekt vom Typ `BranchGroup` sein, andernfalls wird eine `RestrictedAccessException` geworfen.

1730

```
void addChild(Node child, int[] childIndexOrder)
```

Auch diese Methode fügt einen neuen Node zum `OrderedGroup`-Objekt hinzu. Zusätzlich wird das bereits angesprochene `int`-Array übergeben, das die Reihenfolge festlegt, in der die Nodes gezeichnet werden sollen.
1735 Auch hier gilt: Ist das `OrderedGroup`-Objekt Teil eines `SceneGraphen`, der live oder kompiliert ist, so muß das übergebene Objekt vom Typ `BranchGroup` sein, andernfalls wird eine `RestrictedAccessException` geworfen.

```
int[] getChildIndexOrder()
```

1740

```
void setChildIndexOrder(int[] childIndexOrder)
```

Diese Methoden liefern die aktuelle Index-Liste zurück bzw. setzen eine neue Liste, die die Reihenfolge festlegt, in der die Children der `OrderedGroup` gezeichnet werden. Mit der `set`-Methode ist es möglich, die Reihenfolge der Children schnell umzusortieren, so das beispielsweise auf Grund einer veränderten Beobachterposition erforderlich werden könnte. Hier zeigt sich ein Vorteil der indirekten Methode: das `Index-Array` ermöglicht eine schnelle und ressourcenschonende Modifikation der Reihenfolge, bei der nur auf die Indices zugegriffen wird und nicht alle (deutlich größeren) Child-Objekte einzeln verändert werden müssen.
1745

1750 void insertChild(Node child, int index)

Mit dieser Methode wird ein neuer Node an der durch index festgelegten Position eingefügt, alle eventuell bereits vorhandenen Children an einer Position \geq index werden um eins nach hinten verschoben. Diese Methode kann nur dann benutzt werden, wenn noch keine Index-Liste für die Reihenfolge übergeben wurde, diese also null ist.

1755 Ist das OrderedGroup-Objekt Teil eines SceneGraphen, der live oder kompiliert ist, so muß das übergebene Objekt unbedingt vom Typ BranchGroup sein, andernfalls wird eine RestrictedAccessException geworfen.

void moveTo(BranchGroup branchGroup)

1760 Diese Methode verschiebt die übergebene BranchGroup an das Ende der internen Children-Liste. Das führt dazu, dass die Elemente dieser BranchGroup anschließend als letztes gezeichnet werden. Existiert bereits ein Index-Array für die indirekte Festlegung der Render-Reihenfolge, so wird dieses um ein Element erweitert und so verändert, dass die 3D-Objekte dieser BranchGroup ebenfalls als letztes gezeichnet werden.

1765

void removeAllChildren()

Mit dieser Methode werden alle Child-Elemente dieser OrderedGroup entfernt, sie ist anschließend völlig leer.

1770 void removeChild(int index)

void removeChild(Node child)

Diese Methoden entfernen ein bereits zum OrderedGroup-Objekt zugeordnetes Child, das durch dessen Position index innerhalb der internen Objekt-Liste bzw. durch eine Referenz auf das zu entfernende Objekt selbst spezifiziert wird. Existiert bereits eine Index-Liste, die die Objektreihenfolge festlegt, so wird diese entsprechend der Position des hiermit entfernten Children verändert und ihre Länge um eins verkürzt.

1775

Weitere Ähnlichkeiten mit der bereits angesprochenen BranchGroup finden sich auch bei den Vererbungsverhältnissen der Klasse OrderedGroup:

1780

java.lang.Object

 javax.media.j3d.SceneGraphObject

 javax.media.j3d.Node

 javax.media.j3d.Group

1785

javax.media.j3d.OrderedGroup

5 Licht nach Maß

1790 Nach dem im Rahmen der Beschreibung der Materialeigenschaften (die über die Klasse Appearance und die Objekte, für die Appearance eigentlich nur Container ist) bereits für eine Hintergrundbeleuchtung gesorgt werden musste, soll jetzt richtig in das Thema „Licht“ eingestiegen werden. Da mit dem Umgebungslicht eine Lichtart bereits angesprochen wurde, muss diese hier sicher nicht noch einmal erläutert und damit eigentlich nur aufgewärmt werden.

1795

Vielmehr soll dieser Abschnitt die Möglichkeiten beschreiben, die sich mit gezielt definier- und positionierbaren Lichtquellen bieten. Gerade diese Form des Lichtes, welche nicht alles gleichmäßig erleuchtet sondern räumlich begrenzt ist, ermöglicht bei geschicktem Einsatz sehr stimmungsvolle 3D-Szenen.

1800

Vorab wird die unvermeidliche Methode `createSceneGraph()` des Beispielprogrammes wieder etwas verändert. Das `DirectionalLight` wird entfernt und das `AmbientLight` stark abgeschwächt, so dass die Lichter, die demnächst zur Szene hinzugefügt werden sollen, auch gut zu erkennen sind. Weiterhin wird die Gelegenheit genutzt, um wieder ein neues Primitive vorzustellen: Die Box. Während der bereits bekannte `ColorCube` in seinen Variationsmöglichkeiten recht eingeschränkt ist (was von den Entwicklern bei Sun sicher beabsichtigt war, um ein Primitive zur Verfügung zu stellen, das schnell und einfach zu initialisieren ist, auf Grund seiner unterschiedlichen Färbung aber zu Test- und Demonstrationszwecken hervorragend geeignet ist), kann die Box einiges mehr. Im
1805
1810 folgenden Programmcode wird das Primitive wieder direkt der Szene hinzugefügt, ohne den Umweg über eine eigene Variable zu gehen. Der verwendete Konstruktor soll deswegen vorab kurz erläutert werden:

```
Box(1f,2f,3f,Sphere.GENERATE_NORMALS,BoxAppearance)
```

1815

Die ersten drei Werte geben die gewünschten Seitenlängen in x-, y- und z-Richtung an. Das ist ein weiterer großer Unterschied zum `ColorCube`, bei dem – da es sich um einen Würfel handelt – alle Seiten immer gleich lang sind. Bei einer Box kann das auch so sein, muß aber nicht. Der vierte Parameter ist wieder eines der bereits bekannten Flags, bei
1820 dem hier in bekannter Weise **GENERATE_NORMALS** gesetzt wird, damit die Box in der Lage ist, Licht an ihrer Außenseite zu reflektieren (das Gegenstück hierzu wäre **GENERATE_NORMALS_INWARD**, hier würde das Licht im Inneren der Box reflektiert werden). Und ganz zum Schluß wird wieder ein Appearance-Objekt übergeben, welches das farbliche Aussehen der Box beeinflusst.

1825

Doch nun zum gesamten Code der wieder einmal umgebauten Methode:

```
(1)public BranchGroup createSceneGraph()  
(2)    {  
1830 (3)    BranchGroup    RootBG=new BranchGroup();
```

```

(4)   TransformGroup   BoxTG=new TransformGroup();
(5)   Transform3D      BoxT3D=new Transform3D();
(6)   Appearance       BoxAppearance=new Appearance();
(7)
1835 (8)   PointLight      PLgt=new PointLight(new Color3f(1f,0f,0f),new Point3f(-
      2f,0f,0f),new Point3f(0f,0.25f,0f));
(9)   SpotLight        SLgt=new SpotLight(new Color3f(0f,1f,0f),new
      Point3f(5.75f,-1f,-8.9f),new Point3f(0f,0f,0f),new Vector3f(-1f,0f,0f),
      (float)Math.toRadians(25),7f);
1840 (10)
(11)  AmbientLight     ALgt=new AmbientLight(new Color3f(0.3f,0.3f,0.3f));
(12)  BoundingSphere   BigBounds=new BoundingSphere(new Point3d(),100000);
(13)  ALgt.setInfluencingBounds(BigBounds);
(14)  BoxT3D.setTranslation(new Vector3f(0f,0f,-8f));
1845 (15)  BoxT3D.setRotation(new AxisAngle4f(1f,1f,1f,(float)Math.toRadians(-
      60)));
(16)  BoxTG.setTransform(BoxT3D);
(17)  BoxAppearance.setMaterial(new Material(new Color3f(0.5f,0.5f,0.5f),new
      Color3f(0f,0f,0f),new Color3f(0.9f,0.9f,0.9f),new
1850   Color3f(0.8f,0.8f,0.8f),1f));
(18)  BoxTG.addChild(new Box(1f,2f,3f,Sphere.GENERATE_NORMALS,BoxAppearance));
(19)  RootBG.addChild(BoxTG);
(20)
(21)  PLgt.setInfluencingBounds(BigBounds);
1855 (22)  RootBG.addChild(PLgt);
(23)  SLgt.setInfluencingBounds(BigBounds);
(24)  RootBG.addChild(SLgt);
(25)
(26)  RootBG.addChild(ALgt);
1860 (27)  RootBG.compile();
(28)  return RootBG;
(29)  }

```

1865 Die bietet in weiten Teilen nichts unbekanntes. Jedoch haben es die wenigen neuen
 Zeilen in sich. In den Zeilen 8 und 9 werden die Lichtobjekte erzeugt, die im folgenden
 detailliert beschrieben werden. Eine Gemeinsamkeit beider, die auch schon von den
 Umgebungslichttypen her bekannt ist, findet sich in den Zeilen 21 und 23. Hier werden
 wieder Influencing Bounds, also Einflußbereiche definiert, innerhalb derer die Lichter
 Wirkung zeigen sollen. Es ist das gleiche Verfahren wie bei Directional- und
 1870 AmbientLight: Nur 3D-Objekte, die sich innerhalb des Einflußbereiches dieser Grenzen
 befinden, werden beleuchtet. Auch hier wird wieder eine BoundingSphere zur Definition
 dieses Bereiches verwendet.

1875 Was aber ist eigentlich der Sinn hinter der Geschichte mit diesen Influencing Bounds?
Kurz gesagt: es hat Performancegründe. Die Berechnung von Lichtern oder genauer: die Berechnung der Wirkung, die Lichter auf andere Objekte haben, sind eine aufwändige Angelegenheit, die entsprechend Rechenzeit benötigen und sich deswegen nicht eben positiv auf die Darstellungsgeschwindigkeit auswirken. Deswegen ist es sinnvoll, nur die Lichter zu aktivieren, die für den Benutzer auch wirklich gut sichtbar sind. Light-Objekte berechnen zu lassen, die sich ganz am Rande der Szene und/oder sehr weit weg vom Beobachter befinden und deswegen durch ihr Licht effektiv nur noch ein paar vereinzelte Pixel beeinflussen, ist hingegen nicht zweckmäßig. Wenn diese fehlen würden, würde es für den optischen Eindruck praktisch gar keinen Unterschied machen, für die Performance und damit die erzielbare Framerate allerdings schon.

1885
Und genau hier setzen diese Influencing Bounds an: Sie sind ein einfacher und effizienter Weg, um festzulegen, innerhalb welcher Entfernung zu einem Licht es noch Sinn macht, dieses zu berechnen. Deswegen haben diese definierbaren Einflußbereiche eine weitere Eigenschaft, die bisher unter den Tisch fallen gelassen wurde: Sie sorgen dafür, dass nur die Objekte beleuchtet werden, die sich innerhalb der definierten Bounds befinden. Des weiteren – und das ist jetzt neu – muß sich die Position des Beobachters ebenfalls innerhalb dieser Bounds befinden, damit die Lichter überhaupt aktiviert und damit sichtbar werden.

1895 **5.1 BoundingSphere**

Da es weiter oben unterlassen wurde, die BoundingSphere näher zu beleuchten, soll das jetzt nachgeholt werden. Der verwendete Konstruktor legt wie immer alle nötigen Eigenschaften fest:

1900
`BoundingSphere(Point3d center, double radius)`

Da es sich um eine Sphäre, also eine (hier aber nur gedachte, niemals wirklich sichtbare) Kugel handelt, sollte der Sinn der beiden Parameter klar sein: Der erste gibt den relativen Mittelpunkt der Sphäre an. Da sie bzw. Das Lichtobjekt, dem sie hinzugefügt wird, im verwendeten Beispiel nicht Child ein oder mehrerer TransformGroups ist, sind diese relativen Koordinaten gleich den Koordinaten der absoluten Position. Und da in diesem Beispiel ein Point3d-Objekt mit Hilfe des Default-Konstruktors erzeugt wurde, lauten diese Koordinaten 0, 0, 0.

1910
Der zweite Parameter gibt den Radius der Sphäre in der Einheit Meter an.

Des weiteren finden sich in dieser Klasse einige interessante Methoden, die keinesfalls verschwiegen werden sollen:

1915
`Bounds closestIntersection(Bounds[] boundsObjects)`
Von den als Parameter übergebenen Bounds-Objekten wird dasjenige

zurückgeliefert, dass sich mit dem eigenen Bounds-Objekt schneidet. Ist das für mehrere Bounds der Fall, so wird das Objekt als Rückgabewert verwendet, das sich am nächsten am Zentrum der BoundingSphere befindet.

1920

void combine(Bounds boundsObject)
void combine(Bounds[] boundsObjects)

1925 Kombiniert die BoundingSphere mit einem oder mehreren anderen Bounds-Objekten, so dass die resultierende BoundingSphere „this“ alle Bounds einschließt

void combine(Point3d point)
void combine(Point3d[] points)

1930 Hier wird die BoundingSphere mit einem oder mehreren Punkten kombiniert, so dass das eigene BoundingSphere-Objekt diese anschließend ebenfalls beinhaltet.

boolean equals(java.lang.Object bounds)

1935 Diese Methode stellt fest, ob das übergebene Objekt bounds gleich dem aktuellen BoundingSphere-Objekt ist, also ob es eine Instanz der BoundingSphere ist und ob alle Daten dieses Objektes mit denen des übergebenen Bounds-Objektes identisch sind. Ist das der Fall, so wird true zurückgeliefert, andernfalls false.

void getCenter(Point3d center)
void setCenter(Point3d center)

1940 Der Mittelpunkt der BoundingSphere ist ein wichtiger Parameter, der mit diesen beiden Methoden ermittelt oder aber auf einen neuen Wert gesetzt werden kann. Die Koordinaten werden dabei jeweils in das als Parameter übergebene Point3d-Objekt in das BoundingSphere-Objekt kopiert bzw. anders herum.

1945 double getRadius()
void setRadius(double r)

Eine weitere wichtige Eigenschaft, die die BoundingSphere beschreibt, ist ihr Radius. Dessen aktueller Wert kann mit diesen Methoden ermittelt werden bzw. es ist möglich, einen neuen Radius festzulegen.

1950

int hashCode()

1955 Es wird ein Hash-Code zurückgeliefert, der auf den Daten dieses Objektes basiert. Zwei unterschiedliche BoundingSphere-Objekte mit gleichen Daten werden auch den gleichen Hash-Code zurückliefern. Zwei unterschiedliche Objekte mit unterschiedlichen Daten könnten unter Umständen ebenfalls den gleichen Code zurückliefern, allerdings ist das wirklich extrem unwahrscheinlich.

boolean intersect(Bounds boundsObject)

- `boolean intersect(Bounds[] boundsObjects)`
- 1960 Testet, ob sich die BoundingSphere mit dem oder den übergebenen Bounds-Objekten überschneidet. Wenn das so ist, wird `true` zurückgegeben, andernfalls `false`.
- `boolean intersect(Bounds[] boundsObjects, BoundingSphere newBoundSphere)`
- 1965 `boolean intersect(Bounds boundsObject, BoundingSphere newBoundSphere)`
- 1970 Testet, ob sich die übergebene BoundingSphere `newBoundSphere` mit dem oder den übergebenen Bounds-Objekten überschneidet. Wenn das so ist, wird `true` zurückgegeben, andernfalls `false`.
- `boolean intersect(Point3d point)`
- `boolean intersect(Point3d origin, Vector3d direction)`
- 1975 Testet, ob eine Überschneidung zwischen der BoundingSphere und dem übergebenen Punkt oder einem Strahl, der sich aus dem als Parameter übergebenen Punkt und dem Richtungsvektor zusammensetzt, vorhanden ist.
- `boolean isEmpty()`
- 1980 Es wird überprüft, ob die BoundingSphere leer ist. Das ist beispielsweise dann der Fall, wenn ihr Radius negativ ist. In diesen Fällen wird `true` zurückgeliefert. Eine BoundingSphere ist hingegen jedoch nicht leer, wenn ihr Volumen gleich oder größer 0 ist.
- `void set(Bounds boundsObject)`
- 1985 Bei der Verwendung dieser Methode werden die Werte der aktuellen BoundingSphere durch das als Parameter übergebene Bounds-Objekt überschrieben.
- `void transform(Transform3D matrix)`
- `void transform(Bounds boundsObject, Transform3D matrix)`
- 1990 Diese Methoden modifizieren die eigene bzw. die als Parameter übergebene BoundingSphere `boundsObject` so, dass die resultierende BoundingSphere das übergebene und transformierte `boundsObject` anschließend ebenfalls mit beinhaltet
- Eine BoundingSphere leitet sich aus folgenden Klassen ab:
- 1995
- ```

java.lang.Object
 javax.media.j3d.Bounds
 javax.media.j3d.BoundingSphere

```

- 2000 Da es mehrere unterschiedliche Bounds-Klassen als nur die BoundingSphere gibt, sollen die anderen an dieser Stelle ebenfalls entsprechend gewürdigt werden – auch wenn das die Spannung unerträglich macht, weil die Beschreibung der Lichter dadurch noch etwas nach hinten geschoben wird.
- 2005 Anzumerken bleibt noch, dass Java 3D in der Lage ist, um 3D-Objekte und (Teil-) SceneGraphen herum Bounds selbständig zu ermitteln. Das hierfür verwendete Bounds-Objekt ist dann allerdings immer eine BoundingSphere. Wird eine der folgend beschriebenen Bounds-Typen benötigt, so sind diese manuell zu erzeugen und mittels der Methode `setBounds( )` an das jeweilige Node-Objekt zu übergeben.

2010

## 5.2 BoundingBox

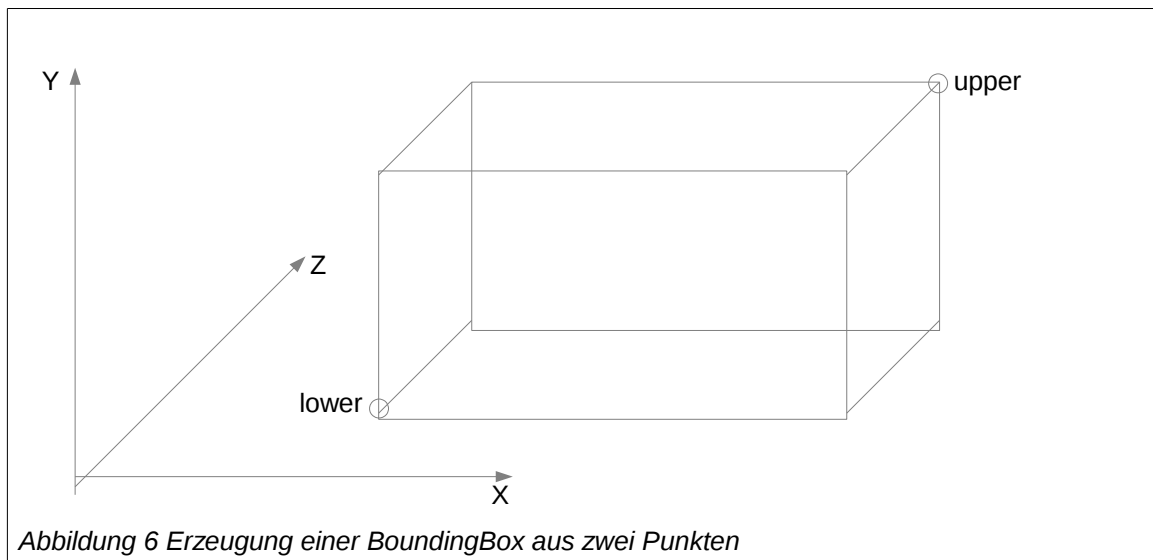
- 2015 Während das vorhergehend beschriebene Bounds-Objekt relativ simpel aufgebaut war und einen Raumbereich eigentlich nur mit Hilfe eines Umkreises um einen Mittelpunkt beschreibt, ist die BoundingBox etwas komplizierter. Diese ebenfalls nur gedachte und niemals in einer Szene sichtbare Box beschreibt einen quaderförmigen Raum und kann beispielsweise mit diesem Konstruktor erzeugt werden:

```
BoundingBox(Point3d lower, Point3d upper)
```

2020

- 2025 Dieser Quader definiert sich mit Hilfe zweier Koordinaten-Sets, die in Form der beiden Point3d-Objekte als Parameter übergeben werden. Diese beiden Punkte genügen in der Tat, um eine Box eindeutig zu definieren: sie beschreiben z.B. einen unteren, vorderen rechten (lower) und einen oberen, hinteren, linken Eckpunkt der Box. Aus diesen ergeben sich alle anderen Eckpunkte und damit auch die Seitenflächen. Voraussetzung dafür, dass diese zwei Punkte genügen, um die Box eindeutig zu beschreiben, ist natürlich, dass die BoundingBox niemals im Raum rotiert werden darf, ihre Kanten müssen also immer zu einer der Achsen X, Y oder Z parallel sein.

2030



Die von der Klasse BoundingBox zur Verfügung gestellten Methoden sind im wesentlichen mit denen der BoundingSphere identisch, so dass neben der Beschreibung der BoundingBox-spezifischen Methoden auf den vorhergehenden Abschnitt und die Java 3D Spezifikation verwiesen werden soll:

2035

```
void getLower(Point3d p1)
void setLower(double xmin, double ymin, double zmin)
void setLower(Point3d p1)
```

2040

Es werden die Koordinaten des unteren Begrenzungspunktes ermittelt bzw. auf neue Werte gesetzt. Bei der Verwendung des Point3f-Objektes als Parameter werden die Koordinaten aus diesem bzw. in dieses kopiert, die Klasse BoundingBox arbeitet also nicht mit Referenzen.

2045

```
void getUpper(Point3d p1)
void setUpper(double xmax, double ymax, double zmax)
void setUpper(Point3d p1)
```

Diese Methoden sind das Gegenstück zu den vorangegangenen, sie behandeln den oberen Begrenzungspunkt der BoundingBox. Mit diesen Methoden ist es möglich, die aktuellen Koordinatenwerte zu ermitteln oder aber neue zu setzen.

2050

Die Klasse BoundingBox leitet sich ähnlich ab wie die BoundingSphere:

2055

```
java.lang.Object
 javax.media.j3d.Bounds
 javax.media.j3d.BoundingBox
```

## 5.3 BoundingPolytope

2060 Ein BoundingPolytope ist ein wesentlich komplexeres Bounds-Objekt. Ein Blick auf einen der elementaren Konstruktoren dürfte erst einmal verwirren:

```
BoundingPolytope(Vector4d[] planes)
```

2065 Ein Bounds-Objekt muß immer einen geschlossenen Raum darstellen. Interessant ist hier, dass er mit Hilfe von Vektoren definiert werden kann. Das Prinzip hinter dem BoundingPolytope erschließt sich aber, wenn man weiß, dass eine `java.lang.IllegalArgumentException` geworfen wird, wenn ein Array von `Vector4d`-Objekten übergeben wird, das kleiner als 4 ist. Somit beschreiben diese Vektoren, in welcher Richtung und in welchem Abstand sich die gedachten Seitenflächen dieses Bounds-Objekts befinden müssen. Und um einen geschlossenen Raum zu erhalten, sind mindestens vier Seitenflächen nötig (ein aus vier Seiten gebildeter Körper wäre dann eine dreiseitige Pyramide).

2070 Neben diversen, bereits von den anderen Bounds-Klassen her bekannten Methoden, finden sich auch bei BoundingPolytope einige klassenspezifische Methoden:

```
int getNumPlanes()
```

Diese Methode liefert die Anzahl an Planes (also Seitenflächen) zurück, aus denen sich dieses BoundingPolytope zusammensetzt.

2080

```
void getPlanes(Vector4d[] planes)
```

```
void setPlanes(Vector4d[] planes)
```

Die Planes selbst, aus denen sich ein BoundingPolytope zusammensetzt, können mit diesen Methoden geholt oder aber neu gesetzt werden.

2085

Da das BoundingPolytope eher selten benötigt werden dürfte, soll der kleine Exkurs in die Welt der Begrenzungsobjekte mit einem Blick auf die Klassenverwandtschaft beendet werden:

2090 `java.lang.Object`

`javax.media.j3d.Bounds`

**`javax.media.j3d.BoundingPolytope`**

## 5.4 PointLight

2095

Doch nun sollen endlich die Lichter im aktuellen Beispielprogramm und damit die Zeile 8

näher beleuchtet werden. Wie bereits der Name des dort erzeugten Objektes sagt, handelt es sich hier um eine punktförmige Lichtquelle, die ihr Licht gleichmäßig in alle Richtungen abstrahlt.

2100

Zu allen Light-Objekten wäre vorab noch zu sagen, dass diese niemals selber in irgend einer Form in einer Szene sichtbar werden, sondern immer nur indirekt durch die Wirkung, die ihr abgestrahltes Licht auf andere 3D-Objekte hat. Möchte man also eine Glühbirne oder ähnliches darstellen, so ist ein zusätzliches 3D-Objekt an der gleichen Position wie die Lichtquelle nötig, die eben dieses leuchtende Objekt darstellt (in dem Fall also die Glühbirne bzw. ihre Glühwendel).

2105

Auch wird beim Raycasting (dem heutzutage für Echtzeit-3D-Rendering verwendeten Verfahren) zwar das Licht, also seine Richtung, seine Farbe und damit seine Wirkung auf andere Elemente der Szene berechnet, niemals aber der Schatten, der eigentlich geworfen werden müsste, wenn sich ein 3D-Objekt im Strahl einer Lichtquelle befindet. Die Berechnung solcher Details ist dem Raytracing vorbehalten, einem Verfahren, das mit heutigen Rechenleistungen nicht in Echtzeit machbar ist und welches damit nur für die Berechnung von Standbildern in Frage kommt (oder aber auch von Animationen, niemals aber von 3D-Welten, mit denen man interagieren kann).

2110

2115

Doch zurück zu einer der Lichtquellen, die sich im Beispielprogramm finden. Dieses wurde mit Hilfe des folgenden Konstruktors erzeugt:

```
2120 PointLight(Color3f color, Point3f position, Point3f attenuation);
```

Der erste an den Konstruktor übergebene Parameter sollte eigentlich selbsterklärend sein: Er spezifiziert die Farbe des abgestrahlten Lichtes. Der Zweite wiederum ist auch nicht weiter spannend, mit Hilfe dieses Point3f-Objektes wird die Position in einer 3D-Welt festgelegt, an der sich die Lichtquelle befindet und von der aus das Licht in alle Richtungen abgestrahlt werden soll.

2125

Deutlich interessanter und für stimmungsvolle Szenen und Effekte wichtig ist der letzte Parameter. Dieser legt mit mehreren Werten fest, wie die Lichtintensität mit zunehmendem Abstand zur Lichtquelle abnehmen soll. Das dafür ein Point3f-Objekt verwendet wird, welches sonst eigentlich eher für Raumkoordinaten verwendet wird, mag verwirren, kann aber getrost ignoriert werden. Die drei „Koordinaten“ x, y und z dieses Objektes werden hier verwendet um eine konstante Dämpfung (x), eine mit dem Abstand zur Lichtquelle linear (y) und eine mit dem Abstand quadratisch (z) zunehmende Abschwächung anzugeben. Es gilt also für einen Point3f-Konstruktor:

2130

2135

```
Point3f(float konstant,float linear,float quadratisch)
```

Die Abschwächung des Lichtes in Abhängigkeit von der Entfernung zur Lichtquelle berechnet sich dabei nach folgender Formel:

2140

$$\text{Abschwächung} = \frac{\text{konstant}}{r^2}$$

2145 Einige der bei der Erzeugung des PointLight-Objektes übergebenen Parameter finden sich auch bei den Methoden dieser Klasse wieder:

```
void getAttenuation(Point3f attenuation)
void setAttenuation(Point3f attenuation)
void setAttenuation(float constant, float linear, float quadratic)
```

2150 Die Attenuation-Werte lassen sich mit diesen Methoden ermitteln bzw. neu setzen. Die einzelnen Werte für die konstante, die lineare und die quadratische Dämpfung werden dabei in die bzw. aus den übergebenen Point3f-Objekten kopiert, deren Daten wie oben bereits beschrieben behandelt werden.

```
2155 void getPosition(Point3f position)
void setPosition(Point3f position)
void setPosition(float x, float y, float z)
```

Auch für die Position des PointLight-Objektes finden sich passende Methoden. Mit diesen ist es möglich, die aktuellen Koordinaten zu ermitteln oder aber neue Werte festzulegen. Auch hier gilt bei der Verwendung eines Point3f-Objektes als Parameter, dass dessen Daten kopiert werden.

2160

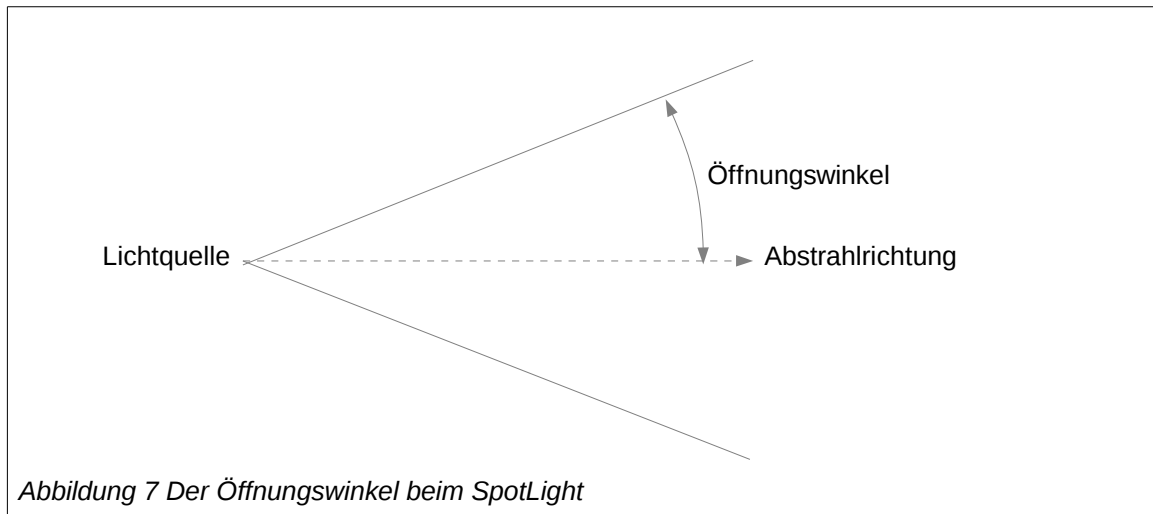
Die Klasse PointLight leitet sich folgendermaßen ab:

```
2165 java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Light
2170 javax.media.j3d.PointLight
```

## 5.5 SpotLight

2175 Das zweite im Beispiel verwendete Light-Objekt ist etwas komplexer und bietet abseits des vorgestellten Codes weitere Experimentiermöglichkeiten. Wie der Name des Objektes bereits andeutet, handelt es sich um einen Spot, also um eine Art Scheinwerfer, der sein Licht von einer definierten Lichtquelle aus kegelförmig abstrahlt. Wie spitz oder wie stumpf dieser Kegel ist, läßt sich natürlich wieder frei definieren. Dazu dient ein

2180 Parameter, der den Öffnungswinkel festlegt, allerdings bezieht sich dieser auf den Mittelpunkt des Kegels. Folgende Darstellung soll das etwas verdeutlichen:



2185 Um so größer dieser Öffnungswinkel also wird, um so größer ist – bei gleichbleibendem Abstand zur Lichtquelle - die beleuchtete Fläche. Andersherum ergibt sich auf Grund der Kegelform des abgestrahlten Lichtes, dass die beleuchtete Fläche mit wachsendem Abstand ebenfalls immer größer wird. Allerdings kann es sein, dass die Fläche dabei aber immer schwächer beleuchtet wird, da auch das Spotlight eine entfernungsabhängige Dämpfung kennt, wie sie schon beim PointLight zum Einsatz kam.

2190 Ein Blick auf den auch in Zeile 9 des Beispielprogrammes verwendeten Konstruktor sollte weitere Erleuchtung in Bezug auf diesen Lichttyp bringen:

```
SpotLight(Color3f color, Point3f position, Point3f attenuation,
Vector3f direction, float spreadAngle, float concentration)
```

2195

2200 Die ersten drei Parameter sind bereits vom PointLight her bekannt, es handelt sich um die Farbe des abgestrahlten Lichtes (hier ein freundliches grün um den Unterschied zu der anderen Lichtquelle in der gleichen Szene deutlich zu machen), die Position der Lichtquelle sowie die Dämpfung in Abhängigkeit von der Entfernung. Die weiteren Parameter werden dann allerdings Spotlight-spezifisch. Mit Hilfe des Vektors wird die Abstrahlrichtung festgelegt, der übergebene Winkel gibt den bereits erwähnten Öffnungswinkel des Lichtkegels an und darf maximal 90° groß werden (worauf sich diese Parameter genau beziehen, ist in Bild 7 zu sehen). Der letzte Parameter wiederum ist ebenfalls für den realistischen Teil der Darstellung verantwortlich, er legt fest, wie stark das Licht innerhalb des Lichtkegels, also vom Zentrum her zu seinem Rand, abgeschwächt werden soll. Wird ein Wert von 0 angegeben, findet keine Abschwächung statt, 128 hingegen ist der maximal erlaubte Wert mit der stärksten Dämpfung des Lichtes zum Rand hin.

2210 Auch das Spotlight kennt einige klassenspezifische Methoden:

```
float getConcentration()
```

```
void setConcentration(float concentration)
```

2215       concentration ist der beim Konstruktor zuletzt beschriebene Parameter, der die Abschwächung des Lichtes innerhalb des Kegels festlegt. Die Stärke dieser Abschwächung vom Zentrum bis zum Rand wird mit diesen Methoden behandelt, sie erlauben es, den aktuellen Wert zu ermitteln oder aber einen neuen Wert im Bereich von 0 bis 128 zu setzen.

2220   void getDirection(Vector3f direction)

```
void setDirection(float x, float y, float z)
```

```
void setDirection(Vector3f direction)
```

      Mit diesen Methoden ist es möglich, die aktuelle Abstrahlrichtung des Spotlights zu ermitteln oder aber einen neuen Vektor für eine andere Richtung festzulegen.

2225

```
float getSpreadAngle()
```

```
void setSpreadAngle(float spreadAngle)
```

      Der letzte noch verbleibende, Spotlight-spezifische Parameter ist der Öffnungswinkel des Lichtkegels. Dessen aktueller Wert kann hiermit geholt bzw. es kann ein neuer Wert dafür festgelegt werden.

2230

Wie man im folgenden sehen kann, wurde SpotLight vom bereits bekannten PointLight abgeleitet, was bedeutet, dass auch die vom PointLight her bekannten Methoden verfügbar sind:

2235

```
java.lang.Object
```

```
 javax.media.j3d.SceneGraphObject
```

```
 javax.media.j3d.Node
```

```
 javax.media.j3d.Leaf
```

2240

```
 javax.media.j3d.Light
```

```
 javax.media.j3d.PointLight
```

```
 javax.media.j3d.SpotLight
```

## 5.6 Die Basisklasse Light

2245

Wie in den vorhergehenden Abschnitten zu sehen war, leiten sich die verschiedenen Lichttypen von der Basisklasse Light ab. Da es sich dabei um eine abstrakte Klasse handelt, ist es nicht möglich, ein Light-Objekt direkt durch die Verwendung eines der Konstruktoren zu erzeugen. Das geht nur über den „Umweg“ einer der von Light



2250 abgeleiteten Klassen. Da diese aber auch die vielfältigen und nicht unwichtigen Methoden erben, sollen diese hier ebenfalls beschrieben werden.

Neben der bereits beschriebenen Möglichkeit und Notwendigkeit, mit `setInfluencingBounds()` einen bestimmten räumlichen Einflußbereich festzulegen, innerhalb dessen das Licht wirksam werden soll, gibt es mit dem so genannten „Scope“ noch eine weitere Option. Dieser Scope besteht aus einer Liste von Teil-SceneGraphen, die explizit an das Light-Objekt übergeben werden. Ist ein solcher Scope definiert, wirkt das Licht nur noch auf Objekte die sich innerhalb des definierten Einflußbereiches befinden und die explizit als zum Scope zugehörig angegeben wurden. Ist kein Scope definiert, werden alle Elemente der Szene, die sich innerhalb der Influencing Bounds befinden, in alt bekannter Weise beleuchtet. Der Scope bzw. die Objekte, die als Scope definiert werden sollen, lassen sich mit den folgenden Methoden bearbeiten:

`void addScope(Group scope)`

2265 Es wird eine neue Group zum aktuellen Scope hinzugefügt. Light-Objekt-intern wird das neue Element an das Ende der Liste der Scope-Objekte angehängt. Die Group, die zum Scope hinzugefügt wird, darf dabei nicht bereits kompiliert worden sein, da sonst eine `RestrictedAccessException` geworfen wird.

2270 `java.util Enumeration getAllScopes()`

Es werden alle aktuell definierten Scope-Elemente in Form einer Enumeration zurückgeliefert.

`Group getScope(int index)`

2275 Diese Methode liefert ein Group-Objekt von der mit `index` spezifizierten Position der Light-internen Scope-Liste zurück.

`int indexOfScope(Group scope)`

2280 Hierbei handelt es sich um das Gegenstück zur vorangegangenen Methode, diese hier liefert den Index-Wert (also die Position) zurück, die das als Parameter übergebene Objekt `scope` innerhalb der klasseninternen Scopeliste hat. Befindet sich das gesuchte Group-Objekt nicht in der Scopeliste, so ist der Rückgabewert -1.

`void insertScope(Group scope,int index)`

2285 Es wird ein neues Group-Objekt zum aktuellen Scope hinzugefügt. Im Gegensatz zur `addScope()`-Methode wird dieses aber an der durch `index` spezifizierten Position in die klasseninterne Scopeliste eingefügt, alle Scope-Objekte an oder nach dieser Position werden dementsprechend um eine Position nach hinten verschoben. Auch bei dieser Methode ist es nicht erlaubt, ein Group-Objekt hinzuzufügen, das Teil eines kompilierten SceneGraphen ist. Andernfalls würde auch hier eine `RestrictedAccessException` ausgelöst werden.

2290

`int numScopes()`

2305 Diese Methode liefert die Anzahl der bereits als Scope definierten Group-Objekte zurück.

`void removeAllScopes()`

2300 Mittels dieser Methode wird die gesamte Scopeliste geleert, anschließend legen wieder nur die Influencing Bounds fest, welche Elemente der 3D-Szene beleuchtet werden sollen, da ja kein Scope mehr definiert ist.

`void removeScope(Group scope)`

`void removeScope(int index)`

2305 Diese Methoden entfernen ein Element aus der internen Scopeliste. Um welches Group-Objekt es sich dabei handelt, wird mit Hilfe des `index`, der die Position innerhalb der Scopeliste angibt oder aber mittels einer Referenz auf das zu entfernende Group-Objekt `scope` selbst angegeben.

`void setScope(Group scope, int index)`

2310 Es wird ein neues Scope-Objekt `scope` an der durch `index` spezifizierten Position in die Scopeliste eingefügt. Befindet sich an dieser Position bereits ein Objekt, so wird dieses durch `scope` ersetzt und ist anschließend nicht mehr Teil des Scopes und wird dann dementsprechend auch nicht mehr beleuchtet. Auch hier gilt, dass eine `RestrictedAccessException` geworfen wird, wenn versucht wird, ein bereits  
2315 kompiliertes Group-Objekt als Scope zu setzen.

Neben diesen Scope-spezifischen Methoden existieren weitere, die sich auf die Basiseigenschaften der Klasse `Light` beziehen:

2320 `void getColor(Color3f color)`

`void setColor(Color3f color)`

Die Farbe des Lichtes lässt sich mit diesen Methoden ermitteln oder aber neu setzen. Wie bereits von anderen Klassen her bekannt, wird der Inhalt der übergebenen `Color3f`-Objekte kopiert.

2325

`boolean getEnable()`

`void setEnable(boolean state)`

Ein `Light`-Objekt kann aktiviert (`true`) oder deaktiviert (`false`) werden. Anders gesagt, kann man es ein- oder ausschalten. Der aktuelle Zustand lässt sich mit diesen  
2330 Methoden ermitteln oder verändern.

`Bounds getInfluencingBounds()`

`void setInfluencingBounds(Bounds region)`

2335 Ein wichtiges Element bei Light-Objekten sind die Influencing Bounds. Ist kein  
zusätzlicher Scope definiert, so bestimmen alleine diese Bounds, welche anderen Objekte  
der 3D-Szene beleuchtet werden sollen. Das aktuell verwendete Bounds-Objekt läßt sich  
mit diesen Methoden holen oder aber durch ein anderes ersetzen.

2340 Die Ableitungsverhältnisse der Klasse Light dürften aus den vorhergehenden Abschnitten  
bereits bekannt sein:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Light
```

## 6 Bewegungen

2350 Die bisher erstellten Szenen waren zwar ganz nett, allerdings bieten sie nicht wirklich viel Abwechslung, da sie allesamt recht statisch waren. Aus diesem Grund soll jetzt etwas Bewegung in die Szenerie kommen. Auch wenn Java 3D sehr elegante Möglichkeiten bietet, Objekte zu animieren, soll das aus Gründen des besseren Verständnisses vorerst einmal von Hand erledigt werden.

2355

Das wesentliche Rüstzeug dafür ist eigentlich bekannt: Einen SceneGraphen zusammenbauen ist für Sie an dieser Stelle kein Problem mehr und mit der TransformGroup bzw. dem Transform3D-Objekt und deren Methoden sind alle wesentlichen Elemente vorhanden, mit denen Sie ein 3D-Objekt in Lage und Position  
2360 verändern können. Ein einfacher und logischer Ansatz wäre also eine zyklische Veränderung des Transform3D-Objektes, z.B. eine Rotation um die Y-Achse. Als Basis dafür soll das Programm aus dem vorhergehenden Abschnitt verwendet werden. Die nötigen Veränderungen hierfür sind recht simpel: Die Variablen für Transform3D und TransformGroup müssen global bekannt gemacht werden, da die Methode `rotY()` des  
2365 Transform3D-Objektes aus einer anderen Methode heraus verwendet werden soll, und dieses anschließend mit `setTransform3D()` der TransformGroup hinzugefügt werden muss. Des weiteren wird ein `javax.swing.Timer` benötigt, der dafür sorgt, dass der aktuelle Rotationswinkel des 3D-Objektes verändert wird, um die gewünschte Drehung des 3D-Objektes zu erhalten.

2370

Da die Änderungen nicht wirklich kompliziert sind, soll hier auf eine vollständige Wiedergabe des Sourcecodes verzichtet werden. Nichts desto trotz befindet sich das Beispielprogramm wieder im zugehörigen Code-Repository, diesmal im Unterverzeichnis „Capability“. Woher dieser Name rührt, zeigt sich, wenn das Programm gestartet wird.

2375

Doch zuvor noch einige Details. Wichtigster Teil des modifizierten Programmes ist eine Methode, die vom ActionListener aus aufgerufen wird, welcher wiederum vom Timer getriggert wird:

```
2380 (1) void handleTimer()
 (2) {
 (3) Transform3D tempT3D=new Transform3D();
 (4)
 (5) ActAngle+=1;
2385 (6) if (ActAngle>=360) ActAngle-=360;
 (7) BoxT3D.setRotation(new AxisAngle4f(1f,0f,1f,(float)Math.toRadians(-60)));
 (8) tempT3D.rotY(Math.toRadians(ActAngle));
 (9) BoxT3D.mul(tempT3D);
 (10) BoxTG.setTransform(BoxT3D);
2390 (11) }
```

So weit werden Sie hier vorerst nichts aufregend Neues finden. Es existiert ein global definierter Zähler `ActAngle`, der den Winkel speichert. Das Objekt `tempT3D` wird benötigt, weil der direkte Aufruf von `BoxT3D.rotY()` sonst die bereits mit Zeile 7 in

2395 `BoxT3D` modifizierte Matrix komplett initialisieren würde. Also werden mit `BoxT3D.setRotation()` nur die Rotationsanteile dieser Matrix überschrieben, die zusätzliche Rotation `rotY()` auf ein temporäres Objekt `tempT3D` angewandt und dann in Zeile 9 mittels `BoxT3D.mul(tempT3D)` der eigentlichen Ziel-TransformGroup hinzugefügt.

2400

Anschließend wird `BoxT3D` in Zeile 10 noch dem TransformGroup-Objekt `BoxTG` zugewiesen. Das ist nötig, um die neue Matrix mit der veränderten Rotation der TransformGroup bekannt zu machen.

2405 Diese Konstruktion sollte nun theoretisch einen netten, rotierenden Quader ergeben, der dazu noch farbig beleuchtet wird. Das Ergebnis nach dem Compilieren und Ausführen des Programmes ist jedoch ein etwas anderes, es wird eine Exception geworfen:

```
2410 javax.media.j3d.CapabilityNotSetException: Group: no capability to set
transform
 at javax.media.j3d.TransformGroup.setTransform(TransformGroup.java:105)
 at Universe.handleTimer(Universe.java:65)
 at Universe$SymAction.actionPerformed(Universe.java:102)
 at javax.swing.Timer.fireActionPerformed(Timer.java:271)
2415 at javax.swing.Timer$DoPostEvent.run(Timer.java:201)
 ...
```

Die Aussage dahinter ist klar wenn auch an dieser Stelle für den nicht vorbelasteten Leser sicher nicht sofort verständlich. Der Aufruf von `BoxTG.setTransform(BoxT3D)` konnte

2420 nicht erfolgreich ausgeführt werden, weil der TransformGroup eine so genannte „Capability“, also eine Fähigkeit, fehlt. Hier geht es um die Fähigkeit, die Transformation setzen zu können, was aus der Fehlermeldung und aus der Zeilennummer hervorgeht, die hier angegeben wird.

## 2425 6.1 Capability

Im Abschnitt zur BranchGroup wurde das Thema bereits kurz angeschnitten. Wird ein SceneGraph mittels `compile()` compiliert, so optimiert das diesen zwar, beraubt ihn und seine Sub-Nodes aber auch gewisser Fähigkeiten, eben jener Capabilities. Eine

2430 Möglichkeit wäre es nun also, auf die Optimierung mittels `compile()` komplett zu verzichten – zum Preis deutlicher Geschwindigkeitseinbußen allerdings. Wäre es statt dessen nicht viel besser, vor der Optimierung festzulegen, welche Fähigkeiten erhalten bleiben sollen? Dann würde für diese Funktionalitäten zwar keine oder keine so

2435 tiefgreifende Optimierung mehr möglich sein, es wäre aber deutlich besser als die  
Rasenmähermethode, bei der immer nur alles oder nichts compiliert und damit optimiert  
werden kann. Der Weg einer solchen feingranularen Spezifikation der noch benötigten  
Fähigkeiten wird noch wichtiger, wenn man weiß, dass Optimierungen nicht nur beim  
Aufruf von `compile()` stattfinden. Ein `SceneGraph`, der live ist – also mit einem Universe  
verbunden ist und dessen Elemente deswegen dort sicht- oder wirksam sind – wird Java-  
2440 3D-intern ebenfalls anders behandelt. Auch hier sind Methoden und Funktionalitäten nicht  
mehr zugänglich, so fern es keinen Weg gibt, diese Fähigkeiten dennoch verfügbar zu  
machen.

2445 Da es nun wie gesagt nicht akzeptabel wäre, auf die tiefgreifende Optimierung auf dem  
Wege der Compilierung zu verzichten und da es eben so wenig zweckmäßig wäre, wenn  
Objekte, deren Eigenschaften verändert werden sollen, jedes mal erst aus dem Universum  
entfernt werden müssten, so dass sie eben nicht mehr live sind, ist in der Tat eine  
passende Lösungsmöglichkeit vorgesehen worden. Mit Hilfe der so genannten „Capability  
Bits“ - die genau genommen jedoch eher Capability-Konstanten sind - und der Methode  
2450 `setCapability(int)`, die für die meisten Nodes zur Verfügung steht und bisher  
allerdings immer verschwiegen wurde, ist es nun möglich, genau die Fähigkeiten  
detailliert zu spezifizieren, die in einem `SceneGraph`, der live oder compiliert ist, weiterhin  
benötigt werden.

2455 Als Parameter erwartet diese Methode `setCapability()` nun eine Konstante, die  
Node-spezifisch festlegt, welche Capabilities erhalten bleiben sollen. Ich schreibe hier mit  
Absicht „Konstante“, da diese Java-3D-intern zwar wirklich „Capability Bits“ genannt  
werden, hier aber tatsächlich keine mit ODER verknüpfbare Flags zum Einsatz kommen,  
sondern Zahlenwerte, für die die Methode `setCapability()` jeweils separat aufgerufen  
2460 werden muß. Das erklärt auch, warum die Methode

```
boolean getCapability(int bit)
```

zum einen einen Parameter erwartet und zum anderen nur ein `boolean` zurückliefert.  
2465 Auch hier muß jeder Capability-Wert einzeln behandelt und abgefragt werden. Die  
Methode gibt dann mittels `true` oder `false` die Information zurück, ob dieses Capability  
„Bit“ gesetzt ist, oder nicht.

2470 Die Methoden `setCapability()` und `getCapability()` entstammen übrigens der  
Klasse `SceneGraphObject`, die unter anderem für alle in einer Szene irgend wie sichtbar  
zu machenden Objekttypen (so also auch `Node`) die Basisklasse ist. Somit stehen diese  
Methoden in fast allen Java-3D-Klassen zur Verfügung.

2475 Doch zurück zu dem Problem mit der Exception, das noch immer nach einer Lösung  
schreit. Für das Beispielprogramm, das bisher so schlecht funktioniert, heißt das  
eigentlich nur, dass vor dem `compile()` eine neue Zeile

```
BoxTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

- 2480 eingefügt werden muß. Die Fehlerinformation der geworfenen Exception besagte ja, dass die Fähigkeit, eine Transformation zu setzen, fehlte. Und genau das ermöglicht das oben aufgeführte Capability Bit **ALLOW\_TRANSFORM\_WRITE** für diese TransformGroup auch wenn sie compiliert und / oder live ist.
- 2485 Wenn Sie einen Blick in die Beschreibung der Klasse Node werfen, werden Sie weitere Methoden finden, die sich mit den Capabilities befassen. Als Erweiterung von `setCapability()` gibt es dort beispielsweise die Methode `setCapabilityIsFrequent()`. Diese ist seit Java 3D Version 1.3 verfügbar und bietet eine noch weitergehende Möglichkeit, Java 3D das Optimieren zu erleichtern und der Engine Hinweise darauf zu geben, was wie verwendet wird. Während `setCapability()` lediglich festlegt, dass eine Fähigkeit überhaupt benutzbar sein soll, wenn der zugehörige SceneGraph live oder compiliert ist, so gibt `setCapabilityIsFrequent()` darüber hinaus an, dass die zugehörige Fähigkeit häufig verwendet werden soll. Anders gesagt, wird die entsprechende Eigenschaft des Nodes häufig verändert. Diese Methode erlaubt es Java 3D, noch zielgerichteter zu optimieren und im Ergebnis noch bessere Resultate mit höheren Frameraten zu erreichen.
- 2490
- 2495

### 6.1.1 Capability Bits der TransformGroup

- 2500 Da das Prinzip der Capability Bits recht leicht verständlich und die Aussagekraft der eventuell geworfenen Exceptions hoch genug ist, sollen im folgenden nur die möglichen Capabilities der eigentlich bereits behandelten TransformGroup exemplarisch näher beleuchtet werden. Diese Klasse kennt für sich spezifisch nur die ersten beiden Capability-Konstanten vom Typ `static int`, alle anderen werden von den übergeordneten Klassen geerbt:
- 2505

#### **ALLOW\_TRANSFORM\_READ**

#### **ALLOW\_TRANSFORM\_WRITE**

- Gibt die Fähigkeit, Transform3D-Objekte zu lesen (`getTransform()`) oder zu setzen (`setTransform()`)
- 2510

Capabilities von `javax.media.j3d.Group` geerbt:

#### **ALLOW\_CHILDREN\_READ**

- 2515 **ALLOW\_CHILDREN\_EXTEND**

#### **ALLOW\_CHILDREN\_WRITE**

- Gibt die Fähigkeit, vorhandene Children zu holen, neue hinzuzufügen oder zu entfernen.
- Mit diesen Capabilities korrespondierende Methoden finden sich z.B. bei der bereits bekannten BranchGroup, sie beeinflussen die Nutzbarkeit von Methoden wie `addChild()`, `getChild()`, `setChild()` etc
- 2520

## **ALLOW\_COLLISION\_BOUNDS\_READ**

## **ALLOW\_COLLISION\_BOUNDS\_WRITE**

2525        Gibt die Fähigkeit, Bounds für die Kollisionsabfrage zu holen oder zu setzen. Da Kollisionen und Collision Bounds bisher nicht beschrieben wurden, bleibt nur kurz zu erwähnen, dass diese Capabilities für Methoden wie `setCollisionBounds()` und `getCollisionBounds()` benötigt werden.

2530    Capability-Konstanten, die von `javax.media.j3d.Node` geerbt wurden:

## **ALLOW\_AUTO\_COMPUTE\_BOUNDS\_READ**

## **ALLOW\_AUTO\_COMPUTE\_BOUNDS\_WRITE**

2535        Gibt die Fähigkeit, den aktuellen Zustand der Auto-Computing-Funktion für Nodes zu lesen oder auf einen neuen Wert zu setzen. Diese Fähigkeit bezieht sich auf einen Fakt, der im Rahmen der Beschreibung der `BoundingSphere` erwähnt wurde. Java 3D berechnet diese Bounds automatisch, wenn das Auto-Computing dafür aktiviert ist (was es in der Voreinstellung immer ist, für einen anderen Zustand müsste das Auto-Computing also explizit abgeschaltet werden). Ändern  
2540    oder den aktuellen Zustand feststellen kann man mit den Methoden `setBoundsAutoCompute()` und `getBoundsAutoCompute()`.

## **ALLOW\_BOUNDS\_READ**

## **ALLOW\_BOUNDS\_WRITE**

2545        Gibt die Fähigkeit, die Bounds eines Nodes zu lesen oder neu zu setzen. Auch diese Capabilities beziehen sich auf die Bounds, nur geht es hier darum, ob es nach einem `compile()` oder wenn der Node live ist, möglich ist, diese mit `getBounds()` zu lesen oder mit `setBounds()` neu zu setzen.

2550    **ALLOW\_COLLIDABLE\_READ**

## **ALLOW\_COLLIDABLE\_WRITE**

       Gibt die Fähigkeit, den Status zur Kollidierbarkeit zu lesen oder zu setzen. Diese Capabilities stehen im Zusammenhang mit einem weiteren, noch nicht besprochenen Feature, der Kollisionsfeststellung und beziehen sich auf die Methoden  
2555    `getCollidable()` und `setCollidable()`.

## **ALLOW\_LOCAL\_TO\_VWORLD\_READ**

       Gibt die Fähigkeit, die Transformation von den lokalen Koordinaten zu den absoluten Koordinaten der virtuellen Welt zu holen, die für diesen Node bzw. einen  
2560    spezifizierten SceneGraph gilt. Dieses Capability Bit bezieht sich auf die Methode `getLocalToWorld()`.

## **ALLOW\_PICKABLE\_READ**



## **ALLOW\_PICKABLE\_WRITE**

2565           Gibt die Fähigkeit, die Eigenschaft „pickable“ zu setzen bzw. deren Zustand zu lesen.  
Diese Konstanten beziehen sich auf die Funktionen `getPickable()` und `setPickable()`, die im Zusammenhang mit dem noch zu besprechenden „Picking“ stehen. Nur kurz zur Erläuterung: dieses „Picking“ stellt eine Möglichkeit dar, bestimmte 3D-Objekte innerhalb einer Szene zu finden.

## **ENABLE\_COLLISION\_REPORTING**

### **ENABLE\_PICK\_REPORTING**

2575           Gibt die Fähigkeit, Kollisionen zu melden oder Selektierungen mittels „Picking“ zu ermöglichen.  
Auch die mit diesen Capabilities in Zusammenhang stehenden Funktionalitäten werden später detaillierter beschrieben.

## **6.2 Der elegante Weg**

2580           Diese Möglichkeit, eine einfache Rotation zu realisieren, war noch recht einfach zu handhaben. Komplexer wird es allerdings, wenn Pendelbewegungen und / oder Beschleunigungen innerhalb der Bewegung hinzukommen sollen. Der Codieraufwand dafür würde um einiges steigen. Es ist allerdings nicht notwendig das alles komplett selbst zu realisieren, da in Java 3D auch dafür eine elegante Lösung in Form der Interpolatoren zusammen mit der Klasse Alpha vorhanden ist. Die Klasse Interpolator bzw. Ihre Unterklassen stellen zusammen mit eben jener Klasse Alpha alle notwendigen Funktionalitäten zur Verfügung, um auch komplexe zyklische Veränderungen innerhalb einer Szene zu realisieren.

2590           Der Interpolator berechnet dabei die gewünschte Bewegung (oder genauer: die gewünschte Änderung eines Zustandes) aus zwei oder mehreren Punkten, sei es nun eine Rotation, eine Verschiebung, eine Farbüberblendung oder ähnliches. Alpha ist für die Art des (Bewegungs)Ablaufes zuständig, also wie lange die Änderung in welche Richtung stattfinden soll, ob sie beschleunigt oder verzögert werden soll, ob und wie lange an einem der Endpunkte verharret werden soll und vieles mehr. Alpha kümmert sich gewissermaßen um das „Delta“, also die Änderung in der zu interpolierenden Veränderung.

2600           Das vorhergehende Beispielprogramm soll deswegen jetzt so abgeändert werden, dass eine Kombination aus RotationInterpolator und Alpha die Rotation des Quaders kontrollieren soll. Weiterhin wäre es sicher auch ganz spannend, ihn nicht mehr nur stumpfsinnig rotieren zu lassen, sondern gleich eine beschleunigte Pendelbewegung aus insgesamt fünf Umdrehungen zu erzeugen. Deswegen wird die Methode  
2605           `createSceneGraph()` - die eigentlich mehr eine Dauerbaustelle ist - folgendermaßen abgeändert:

```

(1)public BranchGroup createSceneGraph()
(2) {
2610 (3) BranchGroup RootBG=new BranchGroup();
(4) Appearance BoxAppearance=new Appearance();
(5) TransformGroup BoxTG=new TransformGroup(),RotTG=new
 TransformGroup();
(6) Transform3D BoxT3D=new Transform3D();
2615 (7) DirectionalLight DLgt=new DirectionalLight(new
 Color3f(0.8f,0.8f,1.0f),new Vector3f(-0.5f,-1f,-0.5f));
(8) AmbientLight ALgt=new AmbientLight(new Color3f(0.8f,0.8f,0.8f));
(9) BoundingSphere BigBounds=new BoundingSphere(new Point3d(),100000);
(10)
2620 (11) RotationInterpolator rotator;
(12) Alpha rotationAlpha;
(13)
(14) ALgt.setInfluencingBounds(BigBounds);
(15) DLgt.setInfluencingBounds(BigBounds);
2625 (16) RootBG.addChild(ALgt);
(17) RootBG.addChild(DLgt);
(18) BoxT3D.setTranslation(new Vector3f(0f,0f,-8f));
(19) BoxT3D.setRotation(new AxisAngle4f(1f,0f,1f,(float)Math.toRadians(-
 60)));
2630 (20) BoxAppearance.setMaterial(new Material(new Color3f(0.5f,0.5f,0.5f),new
 Color3f(0f,0f,0f),new Color3f(0.9f,0.9f,0.9f),new
 Color3f(0.8f,0.8f,0.8f),1f));
(21)
(22) BoxTG.setTransform(BoxT3D);
2635 (23) BoxTG.addChild(RotTG);
(24) RotTG.addChild(new
 com.sun.j3d.utils.geometry.Box(1f,2f,3f,Sphere.GENERATE_NORMALS,BoxAppearance
));
(25) RotTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
2640 (26) rotationAlpha=new Alpha(-1,Alpha.DECREASING_ENABLE|
 Alpha.INCREASING_ENABLE,0,0,9000,3000,500,9000,3000,500);
(27) rotator=new RotationInterpolator(rotationAlpha,RotTG);
(28) rotator.setMinimumAngle(0f);
(29) rotator.setMaximumAngle((float)Math.toRadians(360*5));
2645 (30) rotator.setSchedulingBounds(BigBounds);
(31) RootBG.addChild(rotator);
(32) RootBG.addChild(BoxTG);
(33)
(34) RootBG.compile();

```

```

2650 (35) return RootBG;
 (36) }

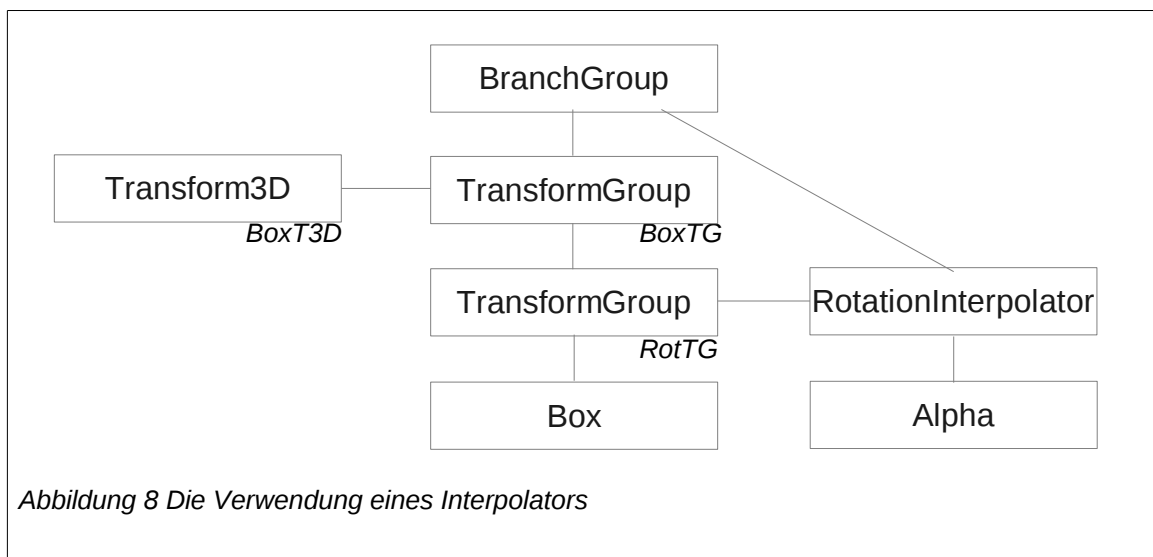
```

Der Übersichtlichkeit wegen werden nur noch Directional- und AmbientLight verwendet, das jedoch in alt bekannter Weise.

2655

Das RotationInterpolator-Objekt `rotator` verwendet eine separate TransformGroup. Das ist notwendig, da dieser Interpolator nur die Rotationsanteile der Transformation behandelt, die Verschiebung und die initiale Rotation der Box jedoch erhalten bleiben sollen. Es ergibt sich somit folgender Teil-SceneGraph:

2660



Als erstes fällt ins Auge, dass RotTG kein eigenes Transform3D-Objekt zugewiesen bekommen hat. Das ist nicht mehr nötig, dieses Objekt wird jetzt vom RotationInterpolator erzeugt und kontrolliert. Des weiteren wird der RotationInterpolator ebenfalls als Child der BranchGroup RootBG in den SceneGraph eingehängt. Das ist eine der drei Voraussetzungen, damit ein Interpolator arbeiten kann. Anders gesagt: erst wenn ein Interpolator „live“, also Teil eines SceneGraphen ist, der einem aktiven Universum hinzugefügt wurde und damit in der Regel auch sichtbar ist, so wird er wirksam. Die zweite Voraussetzung ist, dass für den Interpolator ein Einflußbereich definiert wurde und sich der Beobachter innerhalb dieser Scheduling Bounds befindet.

Eine weitere und damit die dritte Voraussetzung ist, dass ein Alpha-Objekt zugewiesen wurde, dass sich schließlich um den mathematischen Teil der Berechnung der Bewegung kümmern muß.

2675

Nun zum Beispielcode im Detail: `rotator` und `rotationAlpha` werden in Zeile 11 und 12 vorerst nur definiert, nicht jedoch instantiiert, da für deren Konstruktor Teile des SceneGraphen benötigt werden, die an dieser Stelle noch nicht vorhanden sind.

2680 In den Zeilen 22 und 23 findet sich auch nichts neues, hier wird das Transform3D-Objekt BoxT3D der zugehörigen TransformGroup zugewiesen – ganz wie im Bild 8 oben zu sehen. In Zeile 24 wird die Box erzeugt und als Child der neu hinzugekommenen TransformGroup RotTG dem aufzubauenden SceneGraphen hinzugefügt.

2685 In Zeile 25 wird es dann wieder etwas interessanter: für RotTG wird das Capability Bit gesetzt, das es später ermöglicht, der TransformGroup ein Transform3D-Objekt hinzuzufügen. Wozu jedoch das? Das Beispielprogramm ändert doch gar nichts mehr selbst, wenn der SceneGraph compiliert wurde? Die Antwort ist auch hier recht simpel: Der selbstgeschriebene Teil des Programmes tut es nicht, aber der Interpolator muß in

2690 der Lage sein, während der Laufzeit Veränderungen vorzunehmen. Hier heißt das, er muß zyklisch ein geändertes Transform3D-Objekt setzen können, bei dem die gewünschte Rotation berechnet wurde.

In Zeile 26 wird nun das bisher nur kurz angesprochene Alpha-Objekt erzeugt.

2695

### 6.2.1 Alpha

Alpha bietet viele Möglichkeiten, eine Veränderung berechnen zu lassen, die dann mit Hilfe eines Interpolators z.B. in Bewegungen umgewandelt werden kann. Da es viele sehr

2700 verschiedene Arten von Interpolatoren gibt, muß das allerdings auf einer abstrakten Ebene geschehen, so dass Alpha wirklich vielfältig einsetzbar bleibt.

Das wurde gelöst, in dem Alpha einfach nur Änderungen in Form eines Zahlenwertes berechnet, der sich im Bereich von 0.0 bis 1.0 bewegt. Daraus läßt sich in einem

2705 Interpolator dann in der Tat fast alles nötige machen. Die Interpolatoren verwenden diesen Zahlenwert anschließend als Faktor – im oben gezeigten Beispielprogramm wird dieser Faktor einfach mit dem Maximalwinkel der gewünschten Rotation multipliziert.

Was kann Alpha nun alles? Zum einen ist es möglich, das Verhalten für den Anstieg von

2710 0.0 nach 1.0 („increasing“) als auch das für den Weg zurück von 1.0 nach 0.0 („decreasing“) berechnen zu lassen. Beides ist – wie schon erwähnt – jeweils mit einer Beschleunigung realisierbar, so dass sanfte Bewegungen entstehen. Weiterhin kann festgelegt werden, wie lange im Zustand von 0.0 oder 1.0 verharret werden soll. Das alles schlägt sich in dem im Beispielprogramm verwendeten Konstruktor für Alpha nieder, der in

2715 diesem Fall der komplexeste der verfügbaren Varianten ist:

```
public Alpha(int loopCount,
 int mode,
 long triggerTime,
2720 long phaseDelayDuration,
 long increasingAlphaDuration,
 long increasingAlphaRampDuration,
 long alphaAtOneDuration,
```

2725           long decreasingAlphaDuration,  
              long decreasingAlphaRampDuration,  
              long alphaAtZeroDuration)

2730       Es lässt sich leicht erkennen, dass eigentlich drei Parametersätze benötigt werden: einer für den allgemeinen Teil, einer für den „increasing part“, also den Anstieg von 0.0 nach 1.0 und einer für den „decreasing part“, also den Abfall von 1.0 zurück nach 0.0.

Der erste Parameter, `loopCount`, ist sicher klar. Er legt fest, wie oft der Zyklus durchlaufen werden soll. Ein Wert von -1 steht dabei für eine endlose Schleife.

2735       `mode` legt fest, welche Teile des Zyklus berechnet werden sollen. Hier sind die folgenden Flags gültig, die auch per ODER verknüpft übergeben werden können:

- **INCREASING\_ENABLE** legt fest, dass der ansteigende Teil verwendet werden soll
- **DECREASING\_ENABLE** legt auf die Verwendung des absteigenden Teils fest

2740       Soll eine gleichförmige Änderung (wie sie beispielsweise für eine fortwährende Rotation in immer die gleiche Richtung benötigt werden würde) berechnet werden, so ist nur einer der beiden Flags zu setzen, da die Bewegung in jeweils die andere Richtung schließlich nicht erwünscht wäre.

2745       Der Parameter `triggerTime` legt zusammen mit `phaseDelayDuration` fest, wann Alpha in Bezug auf die aktuelle Zeit erstmalig starten soll. Da es im Beispiel sofort losgehen soll, werden beide Werte auf 0 gesetzt.

Es folgen die Parameter, die den Anstieg von 0.0 nach 1.0 beeinflussen und nur beachtet werden, wenn auch das Flag **INCREASING\_ENABLE** gesetzt wurde:

- 2750 - `increasingAlphaDuration` legt die Gesamtdauer der Änderung von 0.0 nach 1.0 fest, inklusive der eventuell mit dem folgenden Parameter definierten Beschleunigungszeiten. Die Einheit des angegebenen Zahlenwertes ist msec (Millisekunden).
- 2755 - `increasingAlphaRampDuration` ist für die Beschleunigung zuständig. Es wird ein Wert in msec übergeben, der jeweils zur Hälfte spezifiziert, wie lange die Beschleunigungsphase am Anfang und am Ende der Änderung nach 1.0 dauern soll. Wird hier – wie im Beispiel oben – ein Wert von 3000 angegeben, dann heißt das, dass nach dem Start bei 0.0 für 1,5 Sekunden beschleunigt wird und kurz vor Erreichen des Wertes 1.0 noch einmal 1.5 Sekunden lang abgebremst wird.
- 2760 - `alphaAtOneDuration` legt fest, für wie viele Millisekunden beim Maximalwert 1.0 verharret werden soll, bevor eine weitere Bewegung erfolgt. Das kann eine Rückbewegung von 1.0 nach 0.0 sein, es kann aber je nach gewählten Werten auch sofort wieder bei 0.0 begonnen werden.

!!!

2765       Für die Rückbewegung und damit die folgenden Parameter muss dementsprechend das Flag **DECREASING\_ENABLE** gesetzt worden sein:

- `decreasingAlphaDuration` legt die Gesamtdauer der Änderung von 1.0 nach 0.0

fest, inklusive der eventuell mit dem folgenden Parameter definierten Beschleunigungszeiten. Die Einheit des angegebenen Zahlenwertes ist msec.

- 2770 - `decreasingAlphaRampDuration` ist wiederum für die Beschleunigung zuständig. Es wird ein Wert in msec übergeben, der jeweils zur Hälfte spezifiziert, wie lange die Beschleunigungsphase am Anfang und am Ende der Änderung dauern soll. Wird hier ein Wert von 0 angegeben, so gibt es keine „sanfte“ Änderung, vielmehr setzt sie sofort ein.
- 2775 - `alphaAtZeroDuration` legt fest, für wie viel Millisekunden bei 0.0 verharret werden soll, bevor eine weitere Bewegung erfolgt.

2780 Mit ein wenig Fantasie kann man hier schon sehen, dass sich die Änderung, die von Alpha berechnet wird, sehr gut für z.B. die gewünschte Pendelbewegung verwendet werden kann. Der `RotationInterpolator` muß im Prinzip nichts weiter tun, als den von Alpha gelieferten Faktor mit einem Wert für den Maximalwinkel zu multiplizieren und diesen in Form eines Rotationswinkels an die Transformation übergeben.

2785 Neben einigen Methoden, die das Setzen und Holen der hier bereits mit dem Konstruktor übergebenen Werte erlauben, finden sich auch einige andere, sicher nicht uninteressante Funktionalitäten:

`boolean finished()`

2790 Liefert diese Methode `true` zurück, so hat das Alpha-Objekt sämtliche Schleifendurchläufe beendet und ist nicht mehr aktiv. Wurde ein `loopCount` von -1 gesetzt so wird diese Methode immer `false` zurückliefern, da Alpha dann in einer Endlosschleife läuft.

`long getAlphaAtOneDuration()`

2795 `void setAlphaAtOneDuration(long alphaAtOneDuration)`

Diese Methoden sind für den Zustand am oberen Endpunkt von Alpha, also den Wert 1.0 zuständig. Sie erlauben es, die Zeitdauer für das Verharren an diesem Endpunkt zu ermitteln oder aber eine neue Zeitspanne in der Einheit Millisekunden festzulegen.

2800 `long getAlphaAtZeroDuration()`

`void setAlphaAtZeroDuration(long alphaAtZeroDuration)`

Diese Methoden arbeiten wie die vorangegangenen beiden, sie ermitteln oder setzen jedoch die Zeitspanne für das Verharren am unteren Endpunkt von Alpha (bei 0.0).

2805 `long getDecreasingAlphaDuration()`

`void setDecreasingAlphaDuration(long decreasingAlphaDuration)`

2810 Wie der Parameter `decreasingAlphaDuration` des oben beschriebenen Konstruktors beziehen sich diese Methoden auf die Zeit, die für den Übergang von 1.0 auf 0.0 verstreichen soll. Mit ihnen ist es möglich, die aktuell verwendete Zeitspanne zu ermitteln oder aber einen neuen Wert in der Einheit Millisekunden zu setzen.

```
long getDecreasingAlphaRampDuration()
void setDecreasingAlphaRampDuration(long
decreasingAlphaRampDuration)
```

2815       Dementsprechend erlauben es diese beiden Methoden, die Zeit für die Beschleunigung bei der Rückbewegung von 1.0 nach 0.0 zu holen oder aber auf einen neuen Wert zu setzen.

```
long getIncreasingAlphaDuration()
2820 void setIncreasingAlphaDuration(long increasingAlphaDuration)
```

Wie für die Rückbewegung (decreasing) existiert eine definierbare Zeitspanne für die Hinbewegung von 0.0 nach 1.0. Diese Methoden liefern die dafür aktuell vorgesehen Zeitspanne zurück bzw. erlauben es, einen neuen Wert zu setzen.

```
2825 long getIncreasingAlphaRampDuration()
void setIncreasingAlphaRampDuration(long
increasingAlphaRampDuration)
```

Analog zu den vorangegangenen Methoden kann auch die Beschleunigungszeit für die Hinbewegung (also „increasing“) mit diesen Methoden ermittelt bzw. auf einen neuen Wert gesetzt werden.

2830

```
int getLoopCount()
void setLoopCount(int loopCount)
```

Diese Methode liefert die Anzahl der bereits ausgeführten Durchläufe zurück bzw. erlaubt es, einen neuen Wert für loopCount zu setzen.

2835

```
int getMode()
void setMode(int mode)
```

Diese Methoden liefern den aktuellen Modus zurück bzw. erlauben es, einen neuen zu definieren. Der Modus wird mit Hilfe der Flags **INCREASING\_ENABLE** und **DECREASING\_ENABLE** festgelegt, die ODER-verknüpft werden können.

2840

```
long getPhaseDelayDuration()
void setPhaseDelayDuration(long phaseDelayDuration)
```

2845       Diese Methoden liefern den bereits oben beim Konstruktor beschriebenen phaseDelayDuration-Wert zurück bzw. erlauben es, hierfür eine neue Zeitspanne in der Einheit Millisekunden festzulegen.

```
long getStartTime()
2850 void setStartTime(long startTime)
```

Auch diese Methoden bilden das Pendant zum gleichnamigen Parameter `startTime` des Konstruktors und liefern die aktuelle Start-Verzögerungszeit zurück bzw. erlauben es, eine neue Zeitspanne zu definieren.

```
2855 void pause()
void pause(long time)
```

Stoppt das Alpha-Objekt auf unbestimmte Zeit bzw. Für die spezifizierte Anzahl an Millisekunden.

```
2860 long getPauseTime()
```

Wie lange das Alpha-Objekt gestoppt wurde, kann mit dieser Methode ermittelt werden. Sie liefert die Zeitspanne in Millisekunden zurück, für die das Alpha-Objekt sich im Zustand „Pause“ befunden hat.

```
2865 boolean isPaused()
```

Diese Methode liefert `true` zurück, wenn das Alpha-Objekt sich in einer Pause befindet, anderenfalls natürlich `false`.

```
void resume()
```

```
2870 Läßt ein Alpha-Objekt, dass sich in einer Pause befindet, weiterlaufen.
```

```
float value()
```

```
float value(long atTime)
```

```
2875 Liefert den aktuellen Alpha-Wert zurück bzw. den Alpha-Wert, der zu dem
festgelegten Zeitpunkt gültig ist
```

Alpha leitet sich von folgenden Klassen ab:

```
java.lang.Object
```

```
2880 javax.media.j3d.SceneGraphObject
```

```
 javax.media.j3d.NodeComponent
```

```
 javax.media.j3d.Alpha
```

```
2885 Da Alpha keine Unterklasse von Node ist, kennt diese Klasse auch keine Capabilities, es
ist also nicht erforderlich, mittels setCapability() oder setCapabilityIsFrequent()
Informationen zur Optimierung von Alpha-Objekten zu übergeben.
```



## 6.2.2 Interpolator

2890 Da der im Beispiel verwendete RotationInterpolator nur ein Beispiel von mehreren, sehr unterschiedlichen Interpolatoren ist, die mit dem Java 3D Paket mitgeliefert werden, an dieser Stelle kurz ein Blick auf die Basisklasse. Diese leitet sich nämlich interessanterweise von einer anderen, sehr nützlichen und später noch zu besprechenden Klasse ab, dem Behavior:

```
2895 java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
2900 javax.media.j3d.Behavior
 javax.media.j3d.Interpolator
```

Da Interpolator von Node erbt, sind die Node-Capabilities bekannt. Eigene Konstanten werden jedoch nicht definiert. Bei den Methoden findet sich dann aber etwas wieder, was 2905 wichtig wird, wenn man eigene Interpolatoren schreiben möchte. Detailliert werden diese aber später beschrieben, wenn mit der Klasse Behavior auf die Grundlagen eingegangen wird:

```
public void initialize()
```

2910 Die Default-Initialisierungsroutine. Diese wird benötigt, damit die Definitionen vorgenommen werden, mit deren Hilfe der Interpolator einmal pro Frame aufgeweckt wird, um die Änderungen entsprechend seiner Aufgabe vorzunehmen.

```
public abstract void processStimulus(java.util.Enumeration
2915 criteria)
```

Diese Methode stammt eigentlich aus der Klasse Behavior und wurde von dort geerbt. Die wird jedes mal dann aufgerufen, wenn wieder ein Frame gezeichnet wurde und der Interpolator arbeiten soll. Dementsprechend muß sie für eigene Interpolatoren überschrieben werden.

2920

### 6.2.2.1 RotationInterpolator

Doch zurück zu dem im Beispielprogramm verwendeten RotationInterpolator. Dieser berechnet – wie der Name schon sagt – eine Rotation, allerdings nur um die Y-Achse. Der 2925 im obigen Beispielprogramm verwendete Konstruktor ist deswegen schnell erklärt:

```
RotationInterpolator(Alpha alpha, TransformGroup target)
```

2930 Als erster Parameter wurde das zuvor erzeugte Alpha-Objekt übergeben, da dieses schließlich benötigt wird, um die Änderungen zum jeweiligen Zeitpunkt zu berechnen und das Ergebnis dem Interpolator zur Verfügung zu stellen. Der Sinn des zweiten Parameters sollte eben so verständlich sein. Hier handelt es sich um das TransformGroup-Objekt, für das die Rotation berechnet werden soll und bei dem demzufolge die Transformation durch den RotationInterpolator gesetzt werden muß.

2935

Die Zeilen 28 und 29 des Beispielprogrammes erfüllen die Bedingung, die an die Rotation geknüpft war. Das Objekt soll sich fünf mal um die eigene Achse drehen. Also muß eine Rotation beginnend bei 0 bis zum fünffachen einer kompletten Umdrehung berechnet werden. Somit ergibt sich der verwendete Term  $5 * 360$ . Diese Informationen werden mit Hilfe von `setMinimumAngle()` (dem Startwinkel) und `setMaximumAngle()` (dem Endwinkel) an den RotationInterpolator übergeben.

2940

Damit ein Interpolator arbeiten kann, müssen mehrere Bedingungen erfüllt sein. Die ViewPlatform (also die Position des Beobachters), muß sich innerhalb des Einflußbereiches des Interpolators befinden, es muß ein Alpha-Objekt zugewiesen worden sein und der Interpolator muß live sein. Die neben der Zuweisung des Alpha-Objektes `rotationAlpha` verbleibenden zwei Bedingungen werden in den Zeilen 30 und 31 erfüllt. Die Methode `setSchedulingBounds()` ist dabei bereits von den Light-Objekten her bekannt – sie spezifiziert eben jenen Einflußbereiche, die „Influencing Bounds“, für das RotationInterpolator-Objekt. In Zeile 31 schließlich wird dafür gesorgt, dass der Rotator auch irgend wann einmal live werden kann, er wird dem Teil-SceneGraphen hinzugefügt, der anschließend durch das Einklinken in das Universe selbst live wird.

2945

2950

Auch hier hat der Einsatz dieser Scheduling Bounds gute Gründe. Das sind – wie auch bei den Lichtern – Performancegründe, die den Einsatz dieses Konstrukts auch für Interpolatoren rechtfertigen. Es ist leicht einzusehen, dass ein Interpolator (der ja eigentlich ein Behavior ist) einiges an Rechenzeit verbraucht, wenn er einmal pro Frame in Aktion tritt. Auch hier ermöglichen es diese Bounds wieder, einen Bereich zu definieren, innerhalb dessen es sinnvoll ist, dass der Interpolator aktiv ist und damit Rechenzeit verbraucht. Verläßt der Benutzer diesen Bereich, wird der Interpolator deaktiviert und benötigt nicht mehr so viele Ressourcen. Also sollte die Größe der Scheduling Bounds immer so gewählt werden, dass ein Interpolator nur dann aktiv sein muß, wenn er auch noch gut und halbwegs detailliert sichtbar ist – nicht wie in obigem Beispielprogramm, in dem der Einflußbereich riesig, ja sogar größer als die maximale Sichtweite des Betrachters ist.

2955

2960

2965

Wie man im geänderten Sourcecode sehen kann, besteht der Teil, der sich um die Änderung in der Scene, hier also um die Rotation kümmert, aus nur wenigen Zeilen. Es ist also deutlich effektiver, das Konzept der Interpolatoren zu verwenden, als jedes mal etwas eigenes zu schreiben. Hinzu kommt, dass die im vorhergehenden Beispielprogramm angewandte Methode mit dem Timer aus dem Paket `javax.swing` eigentlich auch ein unnötiger Aufwand war (der für Java 3D dazu noch ziemlich ineffizient ist). Hier gibt es mit den bereits erwähnten und noch zu besprechenden Behaviors wesentlich bessere Wege.

2970

2975 Auf Grund der spezifischen Eigenschaften dieser Interpolator-Klasse bietet sie auch

einige ebenso spezifische Methoden:

```
void computeTransform(float alphaValue, Transform3D transform)
```

2980 Es wird der Transformationswert für einen spezifischen Alphawert berechnet. Als Parameter erwartet diese Methode zum einen den Alpha-Wert, für den die Transformation berechnet werden soll, als auch ein Transform3D-Objekt, das entsprechend dieses Alphawertes verändert wird.

```
float getMaximumAngle()
```

2985 void setMaximumAngle(float angle)

Der Maximum Angle ist der Rotationswinkel, der beim Erreichen des oberen Endpunktes der Rotation erreicht werden soll. Diese Methoden liefern dessen aktuellen Wert zurück bzw. erlauben es, einen neuen Maximalwinkel zu setzen.

2990 float getMinimumAngle()

```
void setMinimumAngle(float angle)
```

2995 Dementsprechend ist der Minimum Angle der untere Endpunkt bzw. der Anfangspunkt der gewünschten Rotation. Dieser wird von getMinimumAngle() zurückgeliefert bzw. kann mit setMinimumAngle() auf einen neuen Wert gesetzt werden.

Mit einem Blick auf die Ableitung des RotationInterpolator soll auf diesem nicht weiter herumgekaut werden. Informationen über einige der anderen, ebenfalls zusammen mit dem Java 3D Paket verfügbaren Interpolatoren sind sicher interessanter.

3000

```
java.lang.Object
```

```
 javax.media.j3d.SceneGraphObject
```

```
 javax.media.j3d.Node
```

```
 javax.media.j3d.Leaf
```

3005

```
 javax.media.j3d.Behavior
```

```
 javax.media.j3d.Interpolator
```

```
 javax.media.j3d.TransformInterpolator
```

```
 javax.media.j3d.
```

```
 RotationInterpolator
```

3010

### 6.2.2.2 Andere Interpolatoren

3015 Da die verschiedenen Interpolatoren wirklich recht einfach zu benutzen und sehr gut dokumentiert sind, soll hier nur ein Überblick darüber gegeben werden, welche des weiteren verfügbar sind, und wie sie grundlegend zu benutzen sind.

Der **ColorInterpolator** modifiziert die Materialeigenschaft Color (also die Farbe des zugehörigen 3D-Objektes). Notwendige Parameter sind deswegen neben dem in jedem Fall benötigten Alpha-Objekt auch das Material-Objekt, bei welchem die Farbe verändert werden soll sowie natürlich die beiden Farbwerte, die jeweils Ausgangspunkt für die Veränderung sind.

```
3020 ColorInterpolator(Alpha alpha,Material target)
ColorInterpolator(Alpha alpha,Material target,Color3f
3025 startColor,Color3f endColor)
```

Ähnlich dazu verhält sich der **TransparencyInterpolator**. Im Gegensatz zum vorangegangenen Beispiel wird hier allerdings die Transparenz beeinflusst. Notwendige Parameter sind deshalb das TransparencyAttributes-Objekt das auch dem zum 3D-Objekt zugehörigen Appearance-Objekt zugewiesen worden sein muß sowie die beiden Transparenzwerte, die wiederum als Ausgangspositionen dienen sollen:

```
TransparencyInterpolator(Alpha alpha,TransparencyAttributes target)
TransparencyInterpolator(Alpha alpha,TransparencyAttributes
3035 target,float minimumTransparency,float maximumTransparency)
```

Der **TransformInterpolator** kümmert sich um Modifikationen der Transformation des zugehörigen Teil-SceneGraphen. Da der RotationInterpolator eine direkt untergeordnete Klasse ist, dürfte die Funktionsweise aus dem vorherigen Abschnitt her bekannt sein. Ein Unterschied ist hier, dass beim TransformInterpolator allgemein festgelegt werden kann, welche Achse transformiert werden soll (oder besser: welcher Teil der Matrix verändert werden soll). Welcher Teil nun auch immer zu verändern ist, das notwendige Transform3D-Objekt ist mit dem Parameter axisOfTransform zu übergeben:

```
3045 TransformInterpolator(Alpha alpha,TransformGroup target)
TransformInterpolator(Alpha alpha,TransformGroup target,Transform3D
axisOfTransform)
```

Da es sich hier um eine abstrakte Klasse handelt, ist der TransformInterpolator nicht direkt verwendbar und muß – wie das auch mit dem RotationInterpolator getan wurde – als Basisklasse für eigene Ableitungen verwendet werden.

Eine weitere Subklasse des TransformInterpolator findet sich in Form des **ScaleInterpolators**, der die Größe des zugehörigen Teil-SceneGraphen (also der Objekte und Entfernungen in ihm) zwischen zwei Werten ändert. Erforderliche Objekte sind für diesen wiederum die TransformGroup, bei der die Skalierung gesetzt werden soll. Weiterhin werden wieder die Ausgangswerte in Form von minimumScale und maximumScale benötigt:

3060 `ScaleInterpolator(Alpha alpha,TransformGroup target)`  
`ScaleInterpolator(Alpha alpha,TransformGroup target,Transform3D`  
`axisOfTransform,float minimumScale,float maximumScale)`

3065 Der **PositionInterpolator** wiederum ist für Änderungen der Position eines 3D-Objektes bzw. Teil-SceneGraphen entlang der X-Achse zuständig. Neben den bekannten Werten werden hier die `startPosition` und die `endPosition` benötigt, die für die jeweilige X-Koordinate stehen und zwischen denen mit Hilfe des altbekannten Alpha-Objektes die Position geändert werden soll. Wie zu erwarten leitet sich auch diese Klasse vom `TransformInterpolator` ab.

3070 `PositionInterpolator(Alpha alpha,TransformGroup target)`  
`PositionInterpolator(Alpha alpha,TransformGroup target,Transform3D`  
`axisOfTransform,float startPosition,float endPosition)`

3075 Der **PathInterpolator** erweitert die Fähigkeiten der Interpolatoren noch etwas mehr. Er bietet die Möglichkeit, nicht mehr nur zwei Werte festzulegen, zwischen denen dann die Änderung stattfinden, sondern ein ganzes Array davon. Dieses Array wird hier als „Path“ bezeichnet, da es quasi den Weg beschreibt, dem dieser Interpolatortyp folgen soll. Der Wert, den Alpha liefert, legt dann also fest, wo in diesem Array von Werten die aktuelle Position ist und welcher Zustand deswegen angenommen werden soll. Bei dieser Klasse ist es so, dass der Alpha-Wert aussagt, wo im Path (also wo im Array) sich das Objekt gerade befindet. Die Positionen zwischen den einzelnen Werten innerhalb des übergebenen Arrays werden dann vom Interpolator selbst ebenfalls an Hand des aktuellen Alpha-Wertes berechnet.

3080  
3085 Der `PathInterpolator` ist die Basisklasse für mehrere Interpolatoren, die dieses Prinzip praktisch umsetzen. Als Subklasse des `TransformInterpolator` sind die Konstruktoren diesem erwartungsgemäß sehr ähnlich, erweitert um die `knots`, welche das Array sind, das den Path definiert:

3090 `PathInterpolator(Alpha alpha,TransformGroup target,float[] knots)`  
`PathInterpolator(Alpha alpha,TransformGroup target,Transform3D`  
`axisOfTransform,float[] knots)`

3095 Eine der Subklassen, der **PositionPathInterpolator**, verwendet ein Array aus `Point3f`-Objekten. Diese spezifizieren die Positionen, zwischen denen sich der zugeordnete SceneGraph bei Einsatz dieses Interpolators bewegt. Die Konstruktoren beinhalten demzufolge auch dieses hier benötigte Array:

3100 `PositionPathInterpolator(Alpha alpha,TransformGroup`  
`target,Transform3D axisOfTransform,float[] knots,Point3f[]`  
`positions)`

3105 Der **RotationPathInterpolator** wird verwendet, um entlang des gewünschten Path zwischen verschiedenen Rotationen zu wechseln, diese sind in einem Array aus Quat4f-Objekten enthalten:

```
RotationPathInterpolator(Alpha alpha,TransformGroup
target,Transform3D axisOfTransform,float[] knots,Quat4f[] quats)
```

3110 Der **RotPosPathInterpolator** kombiniert die Fähigkeiten der beiden vorangegangenen PathInterpolator und bietet die Möglichkeit, Positionen und Rotationswinkel zu definieren. Demzufolge enthält der Konstruktor zwei Arrays gleicher Länge aus Quat4f- und Point3f-Objekten:

```
3115 RotPosPathInterpolator(Alpha alpha,TransformGroup
target,Transform3D axisOfTransform,float[] knots,Quat4f[]
quats,Point3f[] positions)
```

3120 Der **RotPosScalePathInterpolator** wiederum erweitert den RotPosPathInterpolator – ohne eine direkte Subklasse des selben zu sein – um eine weitere Funktionalität. Ein zusätzliches Array aus floats spezifiziert Scale-Faktoren, die die Größe des zugeordneten Teil-SceneGraphen verändern:

```
3125 RotPosScalePathInterpolator(Alpha alpha,TransformGroup
target,Transform3D axisOfTransform,float[] knots,Quat4f[]
quats,Point3f[] positions,float[] scales)
```

## 7 Beobachtungen

3130 Ausgehend vom bisherigen Beispielprogramm soll im folgenden eine Transformation geändert werden, von der bisher nur ansatzweise die Rede war, die aber wirklich elementare Bedeutung hat. Dabei handelt es sich um die so genannte ViewPlatform, also die Position des Beobachters (oftmals auch als Kameraposition bezeichnet).

3135 Ohne die Möglichkeit, die absolute Position der ViewPlatform in einer virtuellen Welt zu modifizieren wäre es sehr schwierig, Bewegungen des Beobachters zu simulieren. Die einzige Alternative bestünde darin, die Positionen aller 3D-Objekte in einer Welt zu ändern, was aber ziemlich umständlich wäre. Glücklicherweise ist das nicht nötig, da Java 3D mit eben jener ViewPlatform auch die Möglichkeit verbindet, für diese eine Transformation zu setzen, die dann unter anderem auch wieder eine Positionsinformation enthält.

3145 Zusammen mit dem erneut umgebauten und veränderten Beispielprogramm soll auch wieder ein neues Primitive vorgestellt werden: der Kegel. Deswegen zuvor wieder ein kurzer Blick auf den verwendeten Konstruktor, der nichts wirklich aufregend Neues mehr bietet:

```
Cone(0.5f,1.5f,Cone.GENERATE_NORMALS,40,1,ConeAppearance)
```

3150 Der erste Parameter legt den Radius der Grundfläche fest und der Zweite die Höhe des Kegels. Das Flag **GENERATE\_NORMALS** wiederum dient dazu, dass die so genannten Normals generiert werden, welche hier dafür sorgen, dass der Kegel in der Lage ist, Licht an seiner Außenseite zu reflektieren. Die beiden folgenden Werte legen die Auflösung fest, einmal für die Grundfläche und einmal wiederum für die Höhe. Abgeschlossen wird 3155 das Ganze mit der Übergabe eines Appearance-Objektes, welches das Aussehen des Kegels bestimmt ist. Der Parameter, der die Auflösung der Grundfläche bestimmt, ist übrigens interessant: wird hier statt eines so großen Zahlenwertes beispielsweise nur 3 oder 4 angegeben, so ist das Ergebnis kein Kegel mehr, sondern eine drei- bzw. vierseitige Pyramide.

3160 Doch zurück zum eigentlich zu betrachtenden Beobachterposition. Diesmal sind in der `init()`-Methode einige weitere Änderungen notwendig. So wird der Aufruf von

```
u.getViewingPlatform().setNominalViewingTransform()
```

3165 komplett entfernt und durch eine neu zu schreibende Methode `setViewPosition()` ersetzt. Zur Erinnerung: die jetzt nicht mehr benötigte Zeile diente bereits dazu, die Position des Beobachters so zu verändern, dass er die Objekte, die sich auf Position 0, 0, 0 befanden, trotzdem gut erkennen kann. Es handelte sich hier eigentlich schon um die 3170 erste Modifikation der ViewPlatform.

## 7.1 Die ViewPlatform

Der Name ist Programm: Die ViewPlatform ist eine Art Aussichtsplattform, von der aus ein Blick in das selbst geschaffene Universum möglich ist. Auch wenn diese Methode von herkömmlichen Aussichtsplattformen nicht bekannt ist, in der Welt von Java 3D muß genau diese ViewPlatform verändert werden, um einen anderen Blickwinkel auf die Szenerie zu bekommen. Hingegen habe ich es in der realen Welt noch nie erlebt, das sich ein Tourist mitsamt einer Aussichtsplattform durch die Gegend bewegt.

3180

Das Verändern dieser Plattform passiert in der neu hinzugekommenen Methode `setViewPosition()`, die zur Abwechslung mal wieder klein und übersichtlich ist:

```
(1)void setViewPosition()
3185 (2) {
 (3) TransformGroup ViewTG;
 (4) Transform3D ViewT3D=new Transform3D();
 (5)
 (6) ViewTG=u.getViewingPlatform().getViewPlatformTransform();
3190 (7) ViewTG.setTransform(ViewT3D);
 (8) ViewT3D.setTranslation(new Vector3f(0f,0f,4f));
 (9) ViewT3D.setRotation(new AxisAngle4f(0,0,1,(float)Math.toRadians(45)));
 (10) ViewTG.setTransform(ViewT3D);
 (11) }
```

3195

Zeile 3 und 4 beinhalten wieder ein paar notwendige Definitionen. In Zeile 6 schließlich wird eine Referenz auf die TransformGroup der ViewPlatform geholt. Das passiert mittels eines zweifachen Methodenaufrufes. Das SimpleUniverse `u` liefert mit `getViewingPlatform()` ein Objekt vom Typ `ViewingPlatform` zurück, welches wiederum mit Hilfe von `getViewPlatformTransform()` die Referenz auf die gesuchte TransformGroup zurückliefert.

3200

Da es sich nur um eine Referenz, also ein Verweis auf das eigentliche Objekt handelt, ist der weitere Ablauf klar: es wird zuerst das aktuelle Transform3D-Objekt ermittelt (was an dieser Stelle gar nicht nötig gewesen wäre, jedoch sehr nützlich ist, wenn man z.B. die aktuelle Position der ViewPlatform ermitteln möchte), dieses wird anschließend so verändert, dass die neue Position 4 m in Y-Richtung vom Nullpunkt der virtuellen Welt entfernt ist. Des weiteren wird in Zeile 9 eine Rotation um die Z-Achse ausgeführt. Das bedeutet nichts anderes, als dass der Beobachter seinen Kopf zur Seite neigt. Wirksam werden die Veränderungen in dem Moment, in dem sie (in Zeile 10) der TransformGroup zugewiesen werden, die ja die Transformation der Aussichtsplattform, also der ViewingPlatform beeinflusst.

3205

3210

Wird das Programm kompiliert und gestartet, ergibt sich das gewünschte Bild: der



3215 Beobachter steht in respektablem Abstand zum Kegel und sieht diesen – da er seinen Kopf um 45° nach links geneigt hat – schräg.

3220 Da keine anderen Bezugsobjekte in der Szene vorhanden sind, ist nur mit einem Blick auf den Sourcecode festzustellen, dass sich tatsächlich der Beobachter bewegt hat und nicht etwa der Kegel verändert wurde. Da dieser aber direkt und ohne jegliche Transformation zur BranchGroup `RootBG` hinzugefügt wurde, kann der Kegel weder verschoben noch verändert worden sein, es muß also tatsächlich die ViewPlatform ihre Position und Lage geändert haben.

3225 Bedauerlicherweise schweigt sich die JavaDoc-Spezifikation zu Java 3D Version 1.3.x speziell über das Thema ViewPlatform aus. Hier sei deswegen auf die Spezifikation (nicht die JavaDoc-Beschreibung!) zu Java 3D 1.2.x verwiesen. Dort finden sich einige tiefgreifendere Detailbeschreibungen, die für einige Spezialfälle sicher von Interesse sind.

## 3230 7.2 Navigation

Die Wirkungsweise der im vorherigen Abschnitt beschriebenen Transformation der ViewingPlatform fordert geradezu dazu heraus, sinnvoll genutzt zu werden. So ist es naheliegend, die Cursortasten hinzuziehen zu wollen, um mit deren Hilfe die  
3235 Transformation benutzergesteuert auszuführen. Mit anderen Worten: es sollte doch damit möglich sein, dem User eine Steuerung in die Hand zu geben, mit der er sich durch seine virtuelle 3D Welt bewegen kann.

3240 Allerdings fehlt hier noch etwas. Es fehlt eine Möglichkeit, die Cursortasten abzufragen. Herkömmliche Java-Anwendungen lösen das über KeyEvents und den zugehörigen KeyListener. So ähnlich geht das auch hier. Die KeyEvents lassen sich abfangen und auswerten. In Java 3D bietet sich dafür statt eines Listeners allerdings ein Behavior an.

3245 Diese Klasse wurde im Abschnitt zu den Interpolatoren bereits erwähnt. Dort war es so, dass der Interpolator einmal pro Frame aufgerufen wurde, um in Aktion treten zu können (was an dieser Stelle vom Autor einfach behauptet wurde, ohne es wirklich zu zeigen). Wenn es nun möglich wäre, einen Behavior immer dann aufzuwecken, wenn ein KeyEvent auftritt, so ließe sich damit eine Keyboard-Navigationsmöglichkeit durch das virtuelle  
3250 Universum realisieren. Und in der Tat, diese Möglichkeit gibt es in Form der WakeupCriteria und den davon abgeleiteten Klassen.

Doch zuerst soll der Beispielcode erneut etwas umgebaut werden. Aus der Methode `setViewPosition()` wird die Zeile, in der die ViewingPlatform rotiert wurde, entfernt.  
3255 Schließlich soll der Beobachter nicht ständig mit einem schiefen Kopf herumlaufen müssen, das verursacht auf Dauer Haltungsschäden.

In die Methode `createSceneGraph()` werden entsprechend der Verfahrensweise bei den Interpolatoren – die ja von Behavior abgeleitet wurden - folgende Zeilen eingefügt:

```

3260 (1)KeyBehavior KeyCtrl=new
 KeyBehavior(u.getViewingPlatform().getViewPlatformTransform());
 (2)KeyCtrl.setSchedulingBounds(BigBounds);
 (3)RootBG.addChild(KeyCtrl);

```

3265 Diese Zeilen bieten nichts wirklich überraschendes: Es wird ein Behavior -Objekt erzeugt, dem die TransformGroup der ViewingPlatform übergeben wird (welche es schließlich benötigt, um die Position des Beobachters verändern zu können). Dieses Objekt `KeyCtrl` bekommt wieder einen Einflußbereich zugewiesen, innerhalb dessen es aktiv sein soll und wird anschließend dem SceneGraphen hinzugefügt. Hier ist es sinnvoll, den

3270 Einflußbereich auf jeden Fall sehr groß zu wählen. Andererseits kann es passieren, dass der User die ViewingPlatform aus dem Bereich, in dem der Behavior aktiv ist herausbewegt und sich anschließend dank deren Inaktivität nicht mehr bewegen kann. Das ist also eine herrliche Möglichkeit, schwer aufzuspürende Bugs einzubauen.

## 3275 7.3 Behavior

Der eigentlich interessante Teil des Beispielprogrammes findet sich jetzt in einer neuen Klasse „KeyBehavior“, die sich von Behavior ableitet:

```

3280 (1)import java.awt.*;
 (2)import java.awt.event.*;
 (3)import javax.media.j3d.*;
 (4)import javax.vecmath.*;
 (5)import java.util.*;
3285 (6)
 (7)public class KeyBehavior extends Behavior
 (8) {
 (9) private TransformGroup transformGroup;
 (10) private Transform3D trans=new Transform3D(),tempTrans=new Transform3D();
3290 (11) private WakeupCriterion criterion;
 (12)
 (13) public KeyBehavior(TransformGroup tg)
 (14) {
 (15) transformGroup=tg;
3295 (16) }
 (17)
 (18) public void initialize()
 (19) {
 (20) criterion=new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED);

```

```

3300 (21) wakeupOn(criterion);
 (22) }
 (23)
 (24) public void processStimulus(Enumeration criteria)
 (25) {
3305 (26) WakeupOnAWTEvent ev;
 (27) AWTEvent AWTEv[];
 (28) KeyEvent KeyEv;
 (29)
 (30) while (criteria.hasMoreElements())
3310 (31) {
 (32) ev=(WakeupOnAWTEvent)criteria.nextElement();
 (33) AWTEv=ev.getAWTEvent();
 (34) for (int i=0; i<AWTEv.length; i++)
 (35) {
3315 (36) KeyEv=(KeyEvent)AWTEv[i];
 (37) transformGroup.getTransform(trans);
 (38) tempTrans.setIdentity();
 (39) if (KeyEv.getKeyCode()==KeyEvent.VK_UP)
 tempTrans.setTranslation(new Vector3f(0f,0f,-0.2f));
3320 (40) else if (KeyEv.getKeyCode()==KeyEvent.VK_DOWN)
 tempTrans.setTranslation(new Vector3f(0f,0f,0.2f));
 (41) else if (KeyEv.getKeyCode()==KeyEvent.VK_LEFT)
 tempTrans.rotY(Math.toRadians(2));
 (42) else if (KeyEv.getKeyCode()==KeyEvent.VK_RIGHT)
3325 tempTrans.rotY(Math.toRadians(-2));
 (43) trans.mul(tempTrans);
 (44) transformGroup.setTransform(trans);
 (45) }
 (46) }
3330 (47) wakeupOn(criterion);
 (48) }
 (49)
 (50)}

```

3335 Beim groben Überfliegen des Codes fallen die Methoden `initialize()` in Zeile 18 und `processStimulus()` in Zeile 24 auf.

Diese wurden im Abschnitt zu den Interpolatoren bereits kurz erwähnt. Zur Erinnerung: `initialize()` dient dazu, den Behavior zu initialisieren. Genau das ist hier zu sehen.

3340 Es wird eine Variable `criterion` für ein neu erzeugtes `WakeupCriterion`, also ein

3345 Aufwachkriterium verwendet. Dabei handelt es sich hier um ein WakeUpOnAWTEvent-Objekt, dessen Name und Parameter bereits verraten, dass hiermit eine Aufwachbedingung spezifiziert wird, die dafür sorgt, dass jeder Tastendruck registriert wird. Die Methode `wakeupOn()` in Zeile 21 aktiviert schließlich diese Aufwachbedingung für das Behavior-Objekt.

3350 Wird nun eine Taste gedrückt, während der Canvas3D den Fokus hat, so wird automatisch die zweite bei den Interpolatoren kurz erwähnte Methode `processStimulus()` angesprungen. Da ein Behavior mehrere sehr unterschiedliche Aufwachbedingungen haben kann, sollte dort im allgemeinen festgestellt werden, um welche Art von WakeupCriterion es sich handelt um dann dementsprechend zu reagieren.

3355 Auch wenn diese Behavior-Klasse an sich recht kompliziert aussieht, das Verfahren zu Ihrer Verwendung ist tatsächlich so einfach, wie eben beschrieben.

3360 Wichtig zu sagen wäre noch, dass ein WakeupCriterion immer nur einmal gültig ist. Deswegen wird es in Zeile 47 auch erneut gesetzt. Würde das nicht erfolgen, könnte der Behavior auf exakt einen Tastendruck reagieren und wäre anschließend nutzlos. Also muß nach jedem Stimulus durch eine Aufwachbedingung „nachgeladen“ werden.

### 7.3.1 WakeupOnAWTEvent

3365 Ohne dass das Programm jetzt im Detail bekannt ist, soll für ein besseres Verständnis zuerst doch ein Blick auf das verwendete Aufwachkriterium geworfen werden.

Der verwendete Konstruktor

```
WakeupOnAWTEvent(int AWTId)
```

3370 legt im Beispiel oben mit Hilfe des Parameters `KeyEvent.KEY_PRESSED` fest, dass jede gedrückte Taste als gültige Aufwachbedingung anzusehen ist. Das ist bereits alles, was an Definitionen nötig ist. Nach dem Aktivieren dieses Kriteriums mittels `wakeupOn()` ist dieser Behavior „scharf“.

3375 Wie bereits erwähnt kann ein Behavior mehrere Aufwachbedingungen haben. Damit ausgewertet werden kann, welches Ereignis eben eingetreten ist, wird der Methode `processStimulus()` das jeweilige WakeupCriterion-Objekt mitgegeben.

3380 Dieses kann – wie am Parametertyp zu sehen ist – mit mehreren anderen WakeupCriterion-Objekten in einer Enumeration verpackt sein und muß dort einzeln herausgeholt werden. Das geschieht wie im Beispiel zu sehen. Was diesem Programm jetzt allerdings fehlt, ist die Unterscheidung der verschiedenen unterschiedlichen

3385 WakeupCriterion-Objekte mittels `instanceof`. Das wurde der Einfachheit halber hier weggelassen – schließlich wurde für diesen Behavior lediglich der `WakeupOnAWTEvent` definiert.

In Zeile 33 schließlich findet sich eine wichtige Methode dieses Objektes:  
3390 `java.awt.AWTEvent[] getAWTEvent()`. Diese liefert ein Array aus `AWTEvent`-Objekten zurück, dessen Elemente anschließend nur noch einzeln ausgewertet werden müssen. Das geschieht hier in den Zeilen 34 bis 45. Die hier ebenfalls nötige Überprüfung auf die Art des `AWTEvents` wurde abermals unterlassen, da davon ausgegangen wird, dass nur `KeyEvent`s auftreten können. Kommt doch einmal ein anderer Event dazwischen, resultiert das unweigerlich in einer `ClassCastException`. Die Auswertung des `KeyEvent`s hingegen erfolgt in der von AWT bekannten Art und Weise. Je nach dem, welche  
3395 Cursortaste betätigt wurde, wird die Position der ViewingPlattform um 20 cm vor oder zurück verschoben bzw. sie wird um zwei Grad nach rechts oder links rotiert. Damit ist bereits ein zugegebenermaßen sehr einfacher Key-Navigator fertig realisiert.

3400 Bis auf die bereits beschriebene und auch im Beispiel verwendete Methode `getAWTEvent()` bringt diese Klasse keine weiteren, eigenen Methoden mit. Bevor nun also ein Blick auf die anderen möglichen `WakeupCriteria`s geworfen wird, kurz zur Ableitung der Klasse `WakeupOnAWTEvent`:

```
3405 java.lang.Object
 javax.media.j3d.WakeupCondition
 javax.media.j3d.WakeupCriterion
 javax.media.j3d.WakeupOnAWTEvent
```

3410 Da die `WakeupCriteria`s sich nicht von den `Nodes` ableiten, kennen sie auch keine `Capability Bits`, die für sie gesetzt werden müßten. Sämtliche Fähigkeiten eines Behaviors sind also auch im optimierten Zustand oder wenn sie live sind verfügbar.

## 7.3.2 Eine Übersicht über die `WakeupCriteria`s

3415 Neben dem oben angesprochenen und im Beispielprogramm verwendeten `WakeupOnAWTEvent`-Kriterium existieren noch viele andere Aufwachbedingungen, von denen einige im folgenden noch kurz beschrieben werden sollen.

### 7.3.2.1 `WakeupOnActivation`

3420 Wenn die ViewingPlattform erstmalig den Einflußbereich des zugehörigen Behaviors schneidet, so ist das Kriterium `WakeupOnActivation` erfüllt. Da für diese Bedingung keine weiteren Informationen benötigt werden (die nötigen `Scheduling Bounds` sind schließlich schon Teil des Behaviors), ist der Konstruktor der einfachstmögliche:

3425

```
WakeupOnActivation()
```

### 7.3.2.2 WakeupOnBehaviorPost

3430 Dieses Kriterium ist scheinbar paradox: es kann verwendet werden, damit sich ein  
Behavior selbst aufweckt. Nur scheinbar deswegen, weil das unter gewissen Umständen  
durchaus Sinn machen kann. So ist es beispielsweise möglich, dass eine von Behavior  
abgeleitete Klasse zusätzliche einen beliebigen anderen Listener implementiert. Aus  
3435 dessen Methoden heraus kann es dann sinnvoll und notwendig sein, ein Aufwachereignis  
zu triggern.

Der Konstruktor bietet deswegen die Möglichkeit, einmal den Behavior, um den es geht,  
festzulegen und zum anderen eine frei wählbare `postId` zu spezifizieren. Mit Hilfe dieses  
Identifiers wäre es möglich, auch mehrere Events dieses Typs zu unterscheiden:

3440

```
WakeupOnBehaviorPost(Behavior behavior, int postId)
```

Das Aufwachereignis selbst wird mit Hilfe der Methode `postId()` der Klasse Behavior  
getriggert.

3445

Eine wichtige Methode der Klasse WakeupOnBehaviorPost ist wiederum

```
int getPostId()
```

3450 mit der sich der frei wählbare Identifier `postId` für die Auswertung abfragen läßt.

### 7.3.2.3 WakeupOnCollisionEntry, WakeupOnCollisionExit und WakeupOnCollisionMovement

3455 Da diese drei WakeupCriteria eine Einheit bilden und sehr ähnliche Zustände  
behandeln, sollen sie hier auch zusammen beschrieben werden. Bei ihrer Verwendung ist  
die Abbruchbedingung immer dann erfüllt, wenn zwei Objekte neu miteinander kollidieren  
(...Entry), wenn sie zum ersten mal nicht mehr miteinander kollidieren (...Exit) bzw. wenn  
sie sich in kollidiertem, also sich räumlich überlappendem Zustand bewegen  
3460 (...Movement). Dementsprechend sind auch ihre Konstruktoren identisch, weswegen zur  
Beschreibung ausnahmsweise einmal Platzhalter verwendet werden sollen:

```
WakeupOnCollision...(Bounds armingBounds)
```

```
WakeupOnCollision...(Node armingNode)
```

3465 `WakeupOnCollision...(Node armingNode, int speedHint)`  
`WakeupOnCollision...(SceneGraphPath armingPath)`  
`WakeupOnCollision...(SceneGraphPath armingPath, int speedHint)`

3470 `armingBounds` legt fest, innerhalb welcher Grenzen eine Kollision erfolgen muß, damit sie als Aufwachkriterium gilt. Das gleiche gilt für die Parameter `armingNode` und `armingPath`, hier muss innerhalb des angegebenen Nodes (oder seiner Sub-Nodes) bzw. Innerhalb eines `SceneGraphPath` eine Kollision erfolgen, um registriert zu werden.

3475 Der Parameter `speedHint` legt fest, wie exakt die Kollisionserkennung erfolgen soll. **USE\_GEOMETRY** arbeitet sehr exakt, dafür aber auch langsamer. Hier wird anhand der realen geometrischen Struktur der überwachten Objekte festgestellt, ob eine Kollision erfolgt ist oder nicht. Damit werden bei z.B. Hohlkörpern, bei denen ein Objekt berührungslos in einem anderen liegt, keine falschen Kollisionen erkannt (weil die Objekte eben nur ineinander liegen und sich nicht wirklich schneiden). Das ist bei der Verwendung von **USE\_BOUNDS** nicht so, hier werden lediglich die Bounds eines Objektes verwendet, um festzustellen, ob eine Kollision erfolgte. Diese Methode ist zwar deutlich ressourcenschonender und damit schneller, im schlimmsten Fall aber auch sehr ungenau, wenn nur eine `BoundingBox` für die Erkennung verwendet wird.

3485 Des weiteren ist es für die Kollisionsfeststellung erforderlich, dass Nodes auch wirklich kollidierbar sind, d.h. dass die Fähigkeit dafür nicht mit `setCollidable(false)` deaktiviert wurde. Gegebenenfalls muß die Eigenschaft „collidable“ also mittels `setCollidable(true)` für die relevanten Nodes des Teil-SceneGraphen (wieder) aktiviert werden. Gleiches gilt für die Capability **ENABLE\_COLLISION\_REPORTING**  
3490 welches für eine Kollisionsfeststellung in den gewünschten Nodes gesetzt sein muß.

3495 Noch ein paar Worte zum Thema Kollisionen. Diese Aufwachkriterien verleiten zu der Annahme, dass man sie verwenden könnte, um eine Kollisionsfeststellung für bewegte Objekte zu realisieren, die eigentlich nicht kollidieren dürfen. Das ist jedoch falsch. Die `WakeupOnCollision...`-Kriterien werden erst dann aktiv, wenn es bereits zu spät ist, also wenn sich Objekte bereits in einer für die reale Welt unmöglichen Art und Weise überlappen. Was für solche Fälle benötigt wird, ist eine Kollisionsvermeidung. Die sieht so aus, dass für die Bewegungsrichtung des 3D-Objektes überprüft wird, ob andere 3D-Objekte im Weg sind. Ist das der Fall, so ist eine Kollision zu erwarten und das Programm  
3500 muß Maßnahmen ergreifen, noch bevor das passiert.

Natürlich bietet Java 3D Funktionalitäten, die es ermöglichen, andere Objekte zu erkennen. Diese als „Picking“ bezeichnete Methode wird später noch genauer beschrieben.

3505

#### 7.3.2.4 WakeupOnElapsedFrames

Diese Aufwachbedingung läßt sich sehr gut verwenden, um z.B. Dinge wie einen Zähler

3510 für die Framerate zu realisieren. Auch kommt sie bei den bereits bekannten Interpolatoren zum Einsatz. Da diese sehr sanfte Übergänge realisieren sollen, müssen sie für jedes Frame eine Änderung berechnen. Und genau dafür ist `WakeupOnElapsedFrames` gut: es triggert einen Stimulus nach einer bestimmte Anzahl Frames. Der Konstruktor beinhaltet deswegen als Argument auch die gewünschte Frameanzahl `frameCount`:

3515 `WakeupOnElapsedFrames(int frameCount)`

Wenn `frameCount` auf 1 gesetzt wird, so bedeutet das, die Methode `processStimulus()` wird für jedes Frame aufgerufen. Der Wert 0 ist allerdings auch möglich. Dann wird der Behavior zusätzlich mit dem Rendering synchronisiert.

3520

### 7.3.2.5 `WakeupOnElapsedTime`

Diese Aufwachbedingung ist der Vorangegangenen ziemlich ähnlich. Hier ist sie erfüllt, wenn eine bestimmte Zeit abgelaufen ist. Diese Variante der Zeitermittlung ist für Java 3D Applikationen auf Grund der für diesen Anwendungszweck höheren Genauigkeit dem `javax.swing.Timer` vorzuziehen.

3525

Der Konstruktor erwartet deswegen auch – wenig überraschend – die Zeitspanne in Millisekunden, nach der das Kriterium erfüllt sein soll:

3530

`WakeupOnElapsedTime(long milliseconds)`

## 7.3.3 Kombinieren von `WakeupCriteria`s

3535 Wie bereits angedeutet ist es möglich, mehrere Aufwachkriterien zu verwenden. Viel mehr als das können sie sogar in logische Zusammenhänge gebracht werden, sie können mit einem logischen UND bzw. ODER verknüpft werden. Dafür reicht das bisher bekannte jedoch nicht aus, da ein simples, mehrfaches Aufrufen von `wakeupOn()` nicht das gewünschte Ergebnis liefert. Vielmehr würde hier immer nur eine Aufwachbedingung  
3540 gesetzt werden, die Werte aller vorhergehenden Aufrufe würden überschrieben werden.

Um mehrere Kriterien verknüpfen zu können, werden vielmehr Klassen verwendet, die sich von `WakeupCondition` ableiten. Die Namen dieser Klassen deuten bereits an, welche logische Verknüpfung sie realisieren.

3545

Statt der Methode `wakeupOn()` direkt nur ein einzelnes `WakeupCriterion` zu übergeben, werden die gewünschten `Criteria`s mit Hilfe dieser weiter unten beschriebenen „Verknüpfungsklassen“ logisch angeordnet und diese Klasse – die jetzt die verschiedenen Kriterien als eine Art Container beinhaltet – anschließend an `wakeupOn()` übergeben.



3550

### 7.3.3.1 WakeupAnd

Die Klasse WakeupAnd bietet die Möglichkeit, mehrere Aufwachkriterien mit einem logischen UND zu verknüpfen. Das heißt, beim zugehörigen Behavior wird nur dann ein Stimulus getriggert, wenn alle Aufwachbedingungen zusammen gültig sind. Da die Methode `processStimulus()` passenderweise eine Enumeration als Parameter mitbekommt, sind in dieser dann zwangsläufig alle WakeupCriterion-Objekte enthalten und können der Reihe nach ausgewertet werden. Der Konstruktor sollte eigentlich klar verständlich sein, als Parameter `conditions` wird ein Array all der WakeupCriterion-Objekte übergeben, die UND-verknüpft werden sollen:

```
WakeupAnd(WakeupCriterion[] conditions)
```

### 7.3.3.2 WakeupOr

3565

Ähnlich der zuvor beschriebenen Klasse ermöglicht es WakeupOr alle in einem Array übergebenen WakeupCriteria zu verknüpfen, diesmal allerdings mit einem logischen ODER. Das heißt, wenn mindestens eines der in diesem Array enthaltenen Aufwachbedingungen zutrifft, so wird ein Ereignis getriggert und das zugehörige WakeupCriterion-Objekt in bekannter Weise an die Methode `processStimulus()` übergeben. Der Konstruktor ist hier wenig überraschend und erwartet das Array aus den zu verknüpfenden WakeupCriterion-Objekten:

```
WakeupOr(WakeupCriterion[] conditions)
```

3575

### 7.3.3.3 WakeupAndOfOrs

Die vorhergehend beschriebene WakeupCondition WakeupOr läßt sich nun wiederum erneut verknüpfen, mit Hilfe der Klasse WakeupAndOfOrs mit einem logischen UND. Dargestellt mit Hilfe von Pseudocode würde das für die möglichen Aufwachkriterien `c` also so aussehen:

$$((C_1 \mid \mid C_2 \mid \mid \dots C_n) \&\& (C_{10} \mid \mid C_{11} \mid \mid C_{1n}) \&\& \dots)$$

Demzufolge benötigt der Konstruktor dieser Klasse als einzigem zu übergebenden Parameter ein Array aus den WakeupOr-Objekten, die mittels UND verknüpft werden sollen:

```
WakeupAndOfOrs(WakeupOr[] conditions)
```

3590

#### 7.3.3.4 WakeupOrOfAnds

3595 Diese Klasse stellt nun das zu erwartende Gegenstück zu WakeupAndOfOrs dar. Hiermit ist es möglich, mehrere bereits UND-verknüpfte WakeupAnd-Objekte zusätzlich ODER zu verknüpfen. Der Pseudocode stellt sich deswegen folgendermaßen dar:

$$((C_1 \ \&\& \ C_2 \ \&\& \ \dots \ C_n) \ || \ (C_{10} \ \&\& \ C_{11} \ \&\& \ C_{1n}) \ || \ \dots)$$

3600 Auch hier wird wieder ein Array erwartet, diesmal natürlich eines aus WakeupAnd-Objekten:

```
WakeupOrOfAnds(WakeupAnd[] conditions)
```

#### 7.3.4 MouseRotation Behavior

3605

Die in den vorhergehenden Abschnitten beschriebenen Klassen bieten jede erdenkliche Möglichkeit, um die unterschiedlichsten Behaviors zu realisieren. In vielen Fällen dürfte es aber unnötig sein, diese komplett neu zu schreiben, da sich im Paket `com.sun.j3d.utils.behaviors` bereits viele vorgefertigte Behavior-Klassen befinden, die bereits ein breites Spektrum an Einsatzbereichen abdecken. Eine dieser Klassen ist der MouseRotation Behavior. Da es damit sicher nicht möglich sein wird, die an den Computer angeschlossene Maus selbst zu rotieren, bleibt nur noch der Umkehrschluß. Das heißt, diese Behavior-Klasse ermöglicht es wohl, mit der Maus 3D-Objekte oder Teil-SceneGraphen zu rotieren.

3615

Für ein kleines Beispielprogramm wird der zuvor selbstgeschriebene Key-Navigator wiederum etwas abgeändert. Die Klasse KeyBehavior wird nicht mehr benötigt, da jetzt ein fertiger Behavior aus dem Paket `com.sun.j3d.utils.behaviors.mouse` verwendet wird. Des weiteren muß die Methode `createSceneGraph()` des Beispielprogrammes erneut etwas geändert werden:

3620

```
(1)public BranchGroup createSceneGraph()
(2) {
(3) BranchGroup RootBG=new BranchGroup();
3625 (4) TransformGroup ConeTG=new TransformGroup();
(5) Appearance ConeAppearance=new Appearance();
(6) DirectionalLight DLgt=new DirectionalLight(new
 Color3f(0.8f,0.8f,0.8f),new Vector3f(-0.5f,-1f,-0.5f));
(7) AmbientLight ALgt=new AmbientLight(new Color3f(0.8f,0.8f,0.8f));
3630 (8) BoundingSphere BigBounds=new BoundingSphere(new Point3d(),100000);
```

```

(9)
(10) ALgt.setInfluencingBounds(BigBounds);
(11) DLgt.setInfluencingBounds(BigBounds);
(12) RootBG.addChild(ALgt);
3635 (13) RootBG.addChild(DLgt);
(14) ConeAppearance.setMaterial(new Material(new Color3f(0.9f,0.5f,0.5f),new
 Color3f(0f,0f,0f),new Color3f(0.3f,0.7f,0.7f),new
 Color3f(0.8f,0.8f,0.8f),1f));
(15) ConeTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
3640 (16) ConeTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
(17) ConeTG.addChild(new
 Cone(0.5f,1.5f,Cone.GENERATE_NORMALS,40,1,ConeAppearance));
(18) RootBG.addChild(ConeTG);
(19)
3645 (20) MouseRotate MouseCtrl=new MouseRotate(ConeTG);
(21) MouseCtrl.setSchedulingBounds(BigBounds);
(22) RootBG.addChild(MouseCtrl);
(23)
(24) RootBG.compile();
3650 (25) return RootBG;
(26) }

```

Da jetzt nicht mehr die ViewPlatform verändert werden soll, sondern der Kegel, wird in Zeile 4 eine TransformGroup erzeugt, die in Zeile 18 in den SceneGraphen eingehängt wird. Das Cone-Objekt wiederum soll von der Transformation dieser TransformGroup beeinflusst werden, also muß dieses 3D-Objekt ein Child von ihr werden. Das passiert mit dem Aufruf von addChild() in Zeile 17. Wichtig ist es auch wieder, der TransformGroup die nötigen Capabilities mitzugeben, die es dem MouseRotate Behavior später erlauben, die aktuelle Transformation zu lesen (**ALLOW\_TRANSFORM\_READ**) und anschließend eine verändertes Transform3D-Objekt zu setzen (**ALLOW\_TRANSFORM\_WRITE**). Konsequenterweise könnte man diese Capabilities zusätzlich mit setCapabilityIsFrequent() setzen, da zu erwarten ist, dass die Transformation sehr häufig gelesen und geschrieben wird. Da die Szene jedoch noch nicht sehr komplex und damit weit von möglichen Performanceproblemen entfernt ist, ist das eher ein Nice-to-have-Feature.

Das MouseRotate-Behavior-Objekt wird in Zeile 20 erzeugt:

```

MouseRotate(TransformGroup)
3670

```

Der Konstruktor bietet altbekanntes und erwartet als Parameter zwangsläufig die TransformGroup, auf die die Veränderungen wirken sollen. Das heißt auch hier, das alle SceneGraph-Elemente, die unterhalb dieser TransformGroup folgen, von der zugehörigen Transformation beeinflusst werden. Das ist in diesem Beispiel nur ein einzelnes Primitive,

3675 es könnte aber auch deutlich mehr sein, wenn hier ein komplexerer Teil-SceneGraph folgen würde.

3680 Die Zeilen 21 und 22 schließlich bieten wenig überraschendes. Es wird wieder ein Einflußbereich definiert, innerhalb dessen der Behavior wirksam sein soll und er wird dem aktuellen SceneGraphen hinzugefügt. Anschließend kann das Programm compiliert und gestartet werden.

Das Ergebnis ist ein rotierender Kegel, wenn die Maus mit gedrückter Maustaste innerhalb des Canvas3D bewegt wird.

### 3685 7.3.5 Weitere Mouse Behaviors

3690 Da die anderen in `com.sun.j3d.utils.behaviors.mouse` zur Verfügung gestellten Mouse Behaviors ähnlich einfach zu verwenden sind, sollen sie hier nur noch kurz aufgelistet werden. Sie sind praktisch genau so zu verwenden wie im Beispielcode für den MouseRotation Behavior gezeigt.

3695 Basisklasse für all diese Behaviors ist **MouseBehavior**, welche ebenfalls in diesem Paket zu finden ist und natürlich zur Ableitung weiterer eigener Mouse Behaviors verwendet werden kann.

3700 Der **MouseTranslate** Behavior bewirkt eine Veränderung der Position in der zugeordneten TransformGroup in X- und Y-Richtung. Anders als der MouseRotate Behavior muß dazu allerdings die rechte Maustaste gedrückt werden, statt die linke. Auf diese Art lassen sich MouseTranslate Behavior und MouseRotate Behavior auch kombinieren.

3705 Der Behavior **MouseZoom** wiederum benutzt die dritte verbleibende Maustaste um ebenfalls mit den beiden anderen Behaviors kombiniert werden zu können. Er bewirkt ebenfalls eine Veränderung der Position, im Gegensatz zum MouseTranslate Behavior allerdings nur entlang der Z-Achse, was zu einem Effekt des Hinein- oder Herauszoomens führt.

### 7.3.6 KeyNavigatorBehavior

3710 Im Paket `com.sun.j3d.utils.behaviors.keyboard` befinden sich zwei Klassen, die zusammen gehören. KeyNavigator wird hier vom KeyNavigatorBehavior benötigt, welche wiederum in einen SceneGraphen eingebaut werden muß. Diese Klassen realisieren etwas, was in deutlich einfacherer Form in einem der vorangegangenen Abschnitte in Form des unvermeidlichen Beispielprogrammes bereits implementiert wurde: Eine  
3715 Möglichkeit, mit den (Cursor-)Tasten durch eine Szene zu navigieren.

3720 Der KeyNavigatorBehavior aus diesem Paket ist natürlich auch nicht der Weisheit letzter  
Schluß, er läßt sich aber für viele Dinge verwenden oder aber erweitern und ausbauen,  
damit er den eigenen Bedürfnissen gerecht wird. Das wird dadurch erleichtert, das fast  
alle Klassen aus `com.sun.j3d` im SDK als Sourcecode mitgeliefert werden. Somit steht  
einer Weiterverwendung und Erweiterung (entsprechend den in den Fileköpfen  
enthaltenen Weitergabebestimmungen) eigentlich nichts mehr im Wege.

3725 Andererseits besteht auch die Möglichkeit, eine der vielen fertigen Behavior-klassen  
Dritter zu verwenden, die ebenfalls nach dem gleichen Prinzip funktionieren und  
anzuwenden sind. Für den KeyNavigator gibt es z.B. unter <http://www.3dchat.org/dev.php>  
funktional deutlich erweiterte Klassen, die sogar Collision Prevention (also eine  
Kollisionsvermeidung) und Terrain Following (also ein korrektes Beachten einer  
Landschaftsstruktur, so dass eine simulierte Person dieser korrekt folgt) beinhalten.

3730

## 8 Geometry und komplexe 3D-Objekte

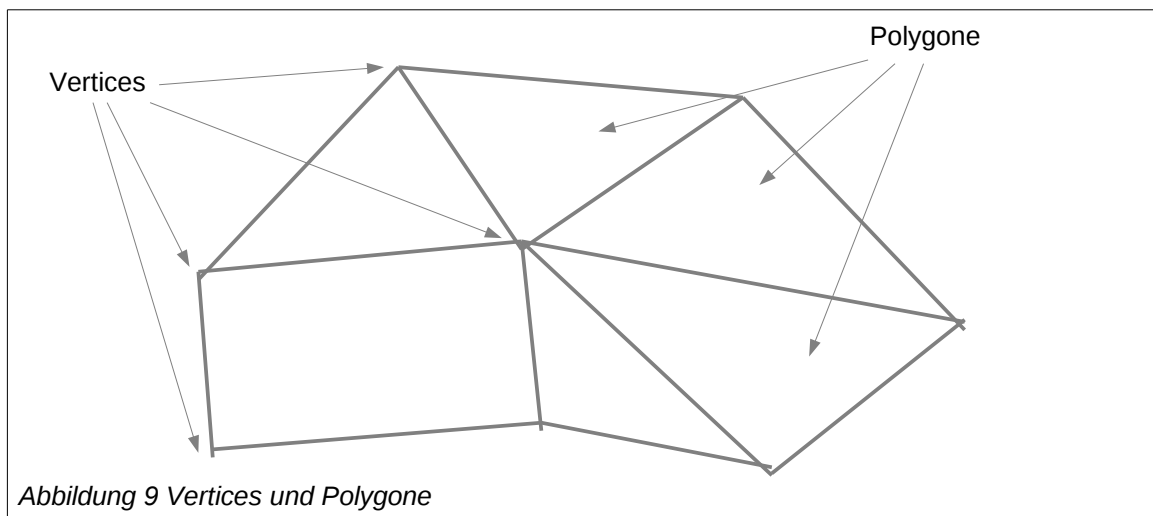
3735 Die in den bisher gezeigten Beispielprogrammen verwendeten 3D-Objekte waren zwar ganz nett und eventuell lässt sich mit diesen auch sehr vieles realisieren, aber letztendlich besteht früher oder später sicher einmal der Wunsch, komplexere Modelle zu verwenden. Das kann ganz praktische Gründe haben, so z.B. Die Tatsache, dass die Objekte in spezieller 3D-Modelliersoftware erzeugt werden sollen, was in diesen im Allgemeinen sehr komfortabel möglich ist. Schön wäre es also, wenn man diese Modelle dann in eine Java 3D Szene bringen könnte.

3740

Um eines gleich vorwegzunehmen: In diesem Kapitel sollen keine spezifischen Dateiformate bis ins Kleinste beschrieben werden. Vielmehr geht es darum, den prinzipiellen Weg aufzuzeigen, der es ermöglicht, aus externen Daten 3D-Objekte zu erstellen. Wie diese Daten in den verschiedenen Dateiformaten der unterschiedlichen  
3745 Programme angeordnet sind, ist dabei zweitrangig, denn letztendlich finden sich die grundlegend benötigten Informationen in allen Formaten in sehr ähnlicher Weise wieder.

### 8.1 Vertex und Polygon

3750 Was sind nun diese Daten? Das sind zum einen die **Vertices**. Das sind nichts weiter als 3D-Koordinaten, die zusammen mit den **Polygonen** die Form eines Objektes festlegen.



3755 Ein solches Polygon ist in der Regel ein Dreieck oder ein Viereck, einige Dateiformate kennen aber auch Vielecke mit einer fast beliebig großen Anzahl Ecken. Sie haben jedoch alle gemein, dass sie durch die Angabe der Vertices, aus denen sie sich zusammensetzen, eindeutig beschrieben werden. Dazu vielleicht ein Beispiel:

Vertex 0 1 0 # 1  
3760 Vertex 1 1 0 # 2

Vertex -1 1 0 # 3

Vertex -1 0 0 #4

Triangle 1 2 3

3765 Triangle 3 4 1

Es werden zuerst vier Vertices, also vier Raumkoordinaten definiert. Der mittels des „Lattenkreuzes“ abgetrennte Zahlenwert ist übrigens nicht relevant. Alles was hinter dem „#“ folgt sind Kommentare und dienen in diesem Beispiel der besseren Übersichtlichkeit.

3770

Diese Vertices werden weiter unten verwendet, um ein zugegebenermaßen noch sehr flaches 3D-Objekt zu definieren. Das passiert hier mittels zweier Dreiecke. Hinter der Anweisung „triangle“ sind hier jedoch nur noch einzelne Zahlenwerte zu finden. Das ist vollkommen korrekt, diese Zahlen verweisen auf die Nummern der Vertices, sie stellen also die Verbindung zu den für die Dreiecke nötigen Raumkoordinaten indirekt her.

3775

Alternativ hätte man statt der beiden Dreiecke auch ein Viereck verwenden können, dann würde statt der letzten beiden Zeilen folgendes stehen:

3780 quad 1 2 3 4

Das sichtbare Ergebnis innerhalb einer virtuellen 3D-Welt wäre in beiden Fällen das Gleiche: Ein flaches, zweidimensionales Quadrat, das auf der Nulllinie steht und dessen Eckpunkte sich an den Raumkoordinaten (0, 1, 0), (1, 1, 0), (-1, 1, 0) und (-1, 0, 0) befinden.

3785

Diese oben gezeigte Pseudo-Dateistruktur lehnt sich übrigens ein wenig an das RWX-Format an. Dennoch soll dies nur der Verdeutlichung dienen, für detailliertere Informationen über dieses RenderWare-Format sei auf die Spezifikation verwiesen.

3790

## 8.2 Texturkoordinaten

Im Zusammenhang mit der Texturierung von Primitives kam es bereits zur Sprache: Es wird in jedem Fall eine Information darüber benötigt, wie eine Textur auf einem Objekt genau positioniert werden soll. Das geschieht mit Hilfe von Texturkoordinaten, die oftmals auch als UV-Koordinaten bezeichnet werden. Das könnte z.B. so aussehen:

3795

UV 5 0 # 1

UV 5 2 # 2

3800 UV 0 2 # 3

UV 0 0 # 4

3805 Diese UV-Koordinaten spezifizieren die Lage einer zweidimensionalen Textur. Was sagen sie aber aus? Nun, in diesem recht einfachen Beispiel wird durch diese Koordinaten festgelegt, dass die Textur exakt parallel zu den Kanten des Quadrats verläuft. Weiterhin wird sie in horizontaler Richtung fünf mal wiederholt, in vertikaler Richtung jedoch nur zweimal. Die linke untere Ecke des Bildes, das für die Textur verwendet wird, befindet sich auch auf dem dargestellten Quadrat links unten – zumindest so lange wie der Beobachter seine Position nicht verändert. Wie müßten die Koordinaten nun aussehen, 3810 wenn die Textur in horizontaler Richtung spiegelverkehrt auf das 3D-Objekt gemappt werden soll? Die X-oder richtiger U- Koordinaten müßten vertauscht werden:

UV 0 0 # 1

UV 0 2 # 2

3815 UV 5 2 # 3

UV 5 0 # 4

Jetzt befindet sich die linke untere Ecke des Bildes (also die mit den UV-Koordinaten 0,0) an der rechten unteren Ecke des Quadrats (also am Vertex Nummer eins).

3820

### 8.3 Farben und Reflektionsverhalten

Bei den verschiedenen Farbattributen wird es hingegen schon schwieriger. Zur Erinnerung: die Klasse Material verwendete Color3f-Objekte, um die Werte für Ambient, 3825 Diffuse und Specular Color festzulegen. Manche 3D-Modeler und dementsprechend ihre Dateiformate verwenden dafür jedoch nur einen einzigen Zahlenwert. Hier bleibt einem also nichts weiter übrig, als diesen Wert im Color3f-Objekt jeweils einmal für x, y und z zu verwenden (was hier für rot, grün und blau steht).

3830 Doch das nur als Anmerkung am Rande, für die Verwendung der Geometrieinformationen zum Erstellen komplexer Objekte ist das vorerst nicht weiter von Bedeutung.

### 8.4 Shape3D – das Universalgenie

3835 Liegen die für das Erstellen eines komplexeren 3D-Objektes mindestens nötigen Informationen – also die Koordinaten in Form der Vertices und die Polygoninformationen – nun vor, wird ein Konstrukt benötigt, das es uns erlaubt, diese zusammenzufügen.

3840 Es wird also ein Objekttyp benötigt, der an Stelle der bisher verwendeten Primitives tritt und der alle nötigen Informationen aufnehmen kann. Dieser Typ nennt sich Shape3D. Und um die Verwirrung, die bei ersten Experimenten ganz sicher entstehen wird, gleich





3885 Äquivalenzklassen zu mischen, so hat das eine `RestrictedAccessException` zur Folge. In Vorgriff auf die verschiedenen Geometry-Klassen seien hier die zusammengehörigen Äquivalenzklassen kurz aufgelistet:

- `PointArray`, `IndexedPointArray`
- `LineArray`, `IndexedLineArray`, `LineStripArray`, `IndexedLineStripArray`
- 3890 - `TriangleArray`, `IndexedTriangleArray`, `TriangleStripArray`, `IndexedTriangleStripArray`, `TriangleFanArray`, `IndexedTriangleFanArray`, `QuadArray`, `IndexedQuadArray`
- `CompressedGeometry`
- `Raster`
- `Text3D`

3895

Darüber hinaus bietet `Shape3D` einige Methoden mehr, von welchen sich ein Teil auch mit den im Folgenden noch zu beschreibenden Geometry-Objekten befasst:

```
java.util Enumeration getAllGeometries()
```

3900 So fern das `Shape3D`-Objekt bereits Geometriedaten hat, werden diese alle mit Hilfe einer Enumeration zurückgeliefert.

```
Appearance getAppearance()
```

```
void setAppearance(Appearance appearance)
```

3905 Diese Methoden beziehen sich auf die dem `Shape3D`-Objekt zugeordnete `Appearance`, sie liefern das aktuell verwendete `Appearance`-Objekt zurück bzw. erlauben es, eine neue `Appearance` für dieses zu definieren.

```
boolean getAppearanceOverrideEnable()
```

3910 

```
void setAppearanceOverrideEnable(boolean flag)
```

Das `appearanceOverrideEnable`-Flag sagt aus, ob die `Appearance` dieses `Shape3D`-Objektes unabhängig von der eventuell gesetzten

3915 **ALLOW\_APPEARANCE\_WRITE**-Capability von einer `AlternateAppearance` überschrieben werden kann. Voreingestellt ist dieses Flag `false`. Der aktuelle Wert dieses Flags wird von der `get`-Methode zurückgeliefert, während es die `set`-Methode erlaubt, es auf einen neuen Wert zu setzen.

```
Bounds getBounds()
```

Es wird das zu diesem Objekt gehörende `Bounds`-Objekt zurückgeliefert

3920

```
Bounds getCollisionBounds()
```

```
void setCollisionBounds(Bounds bounds)
```

Die Collision Bounds sind ein spezielles Bounds-Objekt, das nicht mit den realen Bounds identisch sein muß, die durch die Größe und Form der Geometrie des 3D-Objektes bestimmt werden. Statt dessen können die Collision Bounds größer sein. Mit diesen beiden Methoden ist es deswegen möglich, die aktuellen Collision Bounds zu ermitteln oder aber ein neues Bounds-Objekt zu definieren, das bei Kollisionen herangezogen werden soll.

3930 `Geometry getGeometry()`  
`Geometry getGeometry(int index)`

Es werden die Geometriedaten an Index 0 bzw. von der mittels `index` spezifizierten Position innerhalb der Shape3D-internen Geometriedatenliste zurückgeliefert.

3935 `int indexOfGeometry(Geometry geometry)`

Mit dieser Methode ist es möglich, die Position des Geometry-Objektes `geometry` innerhalb der Geometriedatenliste zu ermitteln. Existiert das als Parameter übergebene Objekt nicht innerhalb der Geometriedaten dieses Shape3D-Objektes, so ist der Rückgabewert -1.

`void insertGeometry(Geometry geometry, int index)`

Es wird eine neue Geometrie-Information zum Shape3D-Objekt hinzugefügt. Im Unterschied zu `addGeometry()` wird diese jedoch nicht angehängt, sondern an der mit `index` spezifizierten Position in die objektinterne Liste mit Geometriedaten eingefügt. Befinden sich an der Position `index` bereits Daten, so werden diese und alle nachfolgenden Geometriedaten in der Shape3D-internen Liste um eins nach hinten verschoben.

3950 `int numGeometries()`

Es wird die Gesamtanzahl von Geometriedaten zurückgeliefert, die sich in der Shape3D-internen Liste bereits befinden.

`void removeAllGeometries()`

3955 Diese Methode entfernt alle Geometriedaten, so dass dieses Shape3D-Objekt – so es live werden würde – in der Szene nicht sichtbar wäre.

`void removeGeometry(Geometry geometry)`

`void removeGeometry(int index)`

3960 Im Gegensatz zur vorhergehenden Methode werden hiermit nicht alle Geometriedaten, sondern nur ein spezifisches Geometry-Objekt entfernt. Dieses kann durch eine Referenz auf das Objekt `geometry` selbst oder aber durch Angabe seiner Position `index` in der Shape3D-internen Liste mit Geometriedaten spezifiziert werden.

```
3965 void setGeometry(Geometry geometry)
void setGeometry(Geometry geometry,int index)
```

Diese Methoden ersetzen Geometriedaten an entweder der Position 0 oder aber an der durch `index` festgelegten Stelle. Befinden sich dort bereits Geometriedaten, so werden diese durch `geometry` überschrieben.

```
3970 boolean intersect(SceneGraphPath path,PickRay pickRay,double[]
dist)
boolean intersect(SceneGraphPath path,PickShape pickShape)
boolean intersect(SceneGraphPath path,PickShape pickShape,double[]
3975 dist)
```

Die `intersect()`-Methoden sind allesamt für das Picking, also das Auffinden von Objekten innerhalb einer virtuellen 3D-Welt nötig. Das Picking und diese Methoden werden in den folgenden Abschnitten detailliert beschrieben.

3980 Ein `Shape3D` leitet sich folgendermaßen ab:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
3985 javax.media.j3d.Leaf
 javax.media.j3d.Shape3D
```

Wie zu sehen ist, ist `Node` eine der Basisklassen von `Shape3D`. Aus vorhergehenden Kapiteln ist bekannt, dass `Nodes` `Capabilities` besitzen, die vor dem Compilieren eventuell gesetzt werden müssen. Es liegt der Verdacht nahe, dass auch `Shape3D` eigene `Capabilities` definiert. Dieser Verdacht bestätigt sich recht schnell, es ist tatsächlich möglich, mit `setCapability()` und `setCapabilityIsFrequent()` das Verhalten in Bezug auf die dem `Shape3D` zugeordnete `Appearance` (**ALLOW\_APPEARANCE\_OVERRIDE\_READ**,  
3990 **ALLOW\_APPEARANCE\_OVERRIDE\_WRITE**, **ALLOW\_APPEARANCE\_READ**,  
3995 **ALLOW\_APPEARANCE\_WRITE**), auf die Geometriedaten (**ALLOW\_GEOMETRY\_READ**, **ALLOW\_GEOMETRY\_WRITE**) sowie auf die `Collision Bounds` (**ALLOW\_COLLISION\_BOUNDS\_READ**,  
**ALLOW\_COLLISION\_BOUNDS\_WRITE**) zu beeinflussen.

4000

### 8.4.1 IndexedTriangleArray

Als nächstes ist es erforderlich, die Geometriedaten hinzuzufügen. Wie im vorhergehenden Abschnitt zu sehen war, muß das mit Hilfe einer Klasse vom Typ  
4005 „Geometry“ geschehen. Eine Ableitung davon ist das „IndexedTriangleArray“, dass – wie der Name schon sagt – Polygone vom Typ „Dreieck“ erwartet.

Zuerst muß dieses spezielle Triangle-Array wie immer konstruiert werden:

```
4010 IndexedTriangleArray(int vertexCount, int vertexFormat, int
indexCount)
```

Der erste Parameter ist sicher klar. `vertexCount` erwartet die Anzahl der für das Objekt benötigten Vertices. Für das Beispiel vom Anfang dieses Kapitels wäre hier 4 anzugeben.

4015

Beim Parameter `vertexFormat` werden wiederum Flags erwartet, die per ODER verknüpft werden können und die angeben, welche Geometriedaten in diesem `TriangleArray` genau enthalten sein werden:

4020 **COORDINATES** ist dabei eigentlich obligatorisch. Es gibt an, dass Koordinaten in Form der Vertices übergeben werden sollen. Da es ohne die nicht geht, ist dieses Flag immer zu setzen.

4025 **NORMALS** gibt an, dass Normal-Koordinaten vorhanden sind und demzufolge ebenfalls übergeben werden sollen. Die so genannten Normals wurden bisher immer nur kurz angerissen. An dieser Stelle so viel dazu, dass sie unter anderem auch dafür benötigt werden, damit ein Objekt in der Lage ist, Licht zu reflektieren. Ohne diese Normals würden die Objekte allerdings nicht schwarz erscheinen sondern unabhängig von einer eventuell vorhandenen (farbigen) Beleuchtung immer nur gleichmäßig in ihre eigene Farbe zu sehen sein, völlig ohne vom Lichteinfall verursachte Farb- und Helligkeitsverläufe.

4030 Einige 3D-Grafikformate liefern die fertig berechneten Normal-Koordinaten gleich mit. In dem Fall ist mit Hilfe dieses Flags zu spezifizieren, dass diese Koordinaten dem Array ebenfalls hinzugefügt werden sollen. Sind sie nicht vorhanden, können sie allerdings auch selbst erzeugt werden. Das Verfahren dazu wird weiter unten beschrieben.

4035 Die Flags **COLOR\_3** und **COLOR\_4** geben an, dass für jeden Vertex auch Farbinformationen vorhanden sind. Diese Farbinformationen würden demzufolge eventuell bereits vorhandene Angaben aus dem Material-Objekt der `Appearance` überschreiben. Zwei verschiedene Flags sind hier deshalb von Nöten, da die Farben in Arrays von `Color3f`- oder von `Color4f`-Objekten übergeben werden können. Es gibt noch einige Varianten mehr, diese sind für ein erstes Verständnis der Mechanismen zur Erzeugung komplexer 3D-Objekte jedoch wirklich nicht von Belang.

4040 **TEXTURE\_COORDINATE\_2**, **TEXTURE\_COORDINATE\_3** oder **TEXTURE\_COORDINATE\_4** geben an, dass auch Texturkoordinaten gesetzt werden sollen. Dabei gelten diese Flags für zwei-, drei- oder vierdimensionale Texturen. Die zugehörigen Koordinaten werden demzufolge in der Regel auch mit Arrays aus `TexCoord2f`-, `TexCoord3f`- oder `TexCoord4f`-Objekten übergeben.

4050 Der letzte Parameter, `indexCount`, legt schließlich fest, wie viele auf die Vertices verweisenden Indices enthalten sein werden, um das gewünschte 3D-Objekt zu konstruieren. Für obiges Beispiel wären das 6, da sich dieses Objekt aus zwei Dreiecken zusammensetzt, für die logischerweise zweimal drei Indices benötigt werden.

Damit sollte es möglich sein, das benötigte TriangeArray-Objekt zu konstruieren. Anschließend müssen die eigentlichen Daten nur noch hinzugefügt werden. Die dafür verwendeten Methoden stammen überwiegend aus der Basisklasse GeometryArray und sind deswegen in allen anderen, ebenfalls von dieser Klasse abgeleiteten und weiter unten näher betrachteten Geometrie-Klassen ebenfalls zu finden:

```
void setCoordinates(int index,float[] coordinates)
void setCoordinates(int index,double[] coordinates)
4060 void setCoordinates(int index,Point3f[] coordinates)
void setCoordinates(int index,Point3d[] coordinates)
void setCoordinates(int index,float[] coordinates,int start,int
length)
void setCoordinates(int index,double[] coordinates,int start,int
4065 length)
void setCoordinates(int index,Point3f[] coordinates,int start,int
length)
void setCoordinates(int index,Point3d[] coordinates,int start,int
length)
```

4070 Diese Methoden dienen dazu, die Vertexkoordinaten hinzuzufügen. Als Parameter wird dabei der `index` erwartet, ab dem beginnen die Koordinatenwerte eingefügt werden sollen. `coordinates` ist dann das Array, das diese Vertexkoordinaten enthält. Wie zu sehen ist, ist es möglich, die Koordinaten wahlweise mit Hilfe mehrerer unterschiedlicher Datentypen zu übergeben. Das wären zum einen Arrays aus `float` oder `double`, bei denen immer drei aufeinanderfolgende Werte einen zusammengehörenden Satz

4075 Raumkoordinaten bilden. Alternativ können auch Arrays aus `Point3f`- oder `Point3d`-Objekten benutzt werden. Hier enthält dann immer genau ein Arrayelemente genau ein Koordinatentriplett aus X-, Y- und Z-Koordinate. Damit wären die Vertices für dieses GeometryArray festgelegt.

4080 Doch auch die möglichen letzten beiden Parameter sollen nicht verschwiegen werden. Während `index` die Zielposition im GeometryArray festlegt, gibt `start` die Quellposition innerhalb des übergebenen Arrays an, ab der Elemente zu `index` kopiert werden sollen. Der letzte Parameter, `length`, legt fest, wie viele der Koordinaten aus dem Array übergeben werden sollen. Diese Parameter sind ähnlich denen der Java-Methoden zur

```
void setCoordinateIndices(int index,int[] coordinateIndices)
```

4090 Diese Methode entstammt der Basisklasse IndexedGeometryArray, die sich selber natürlich ebenfalls auch von GeometryArray ableitet. Sie dient dazu, die Verweise auf die Vertices hinzuzufügen, von denen jeweils drei aufeinanderfolgende Array-Elemente immer genau ein Dreieck spezifizieren. Das klingt jetzt sicher furchtbar kompliziert, was es aber nicht ist. Am Anfang des Kapitels wurde genau diese indirekte Methode bereits

4095 beschrieben: Die Indices geben eigentlich nur an, welche zuvor angegebenen Vertexkoordinaten jeweils ein Dreieck bilden. Der Parameter `index` ist bereits aus der vorhergehend beschriebenen Methode bekannt,

er gibt hier ebenfalls an, ab welcher Position die Index-Informationen hinzugefügt werden sollen. Das Array, das als zweiter Parameter übergeben wird, enthält wie zu erwarten die Verweise auf die Koordinaten. Hier ist darauf zu achten, dass keine Indices enthalten sind, die auf Koordinaten verweisen, die nicht existieren. Wurden als z.B. 1000 Koordinaten übergeben, so darf die größte Indexnummer die 999 sein, da die Indices bei 0 beginnend gezählt werden. Sind größere Werte enthalten, kann das zu Exceptions führen, die jedoch erst bei der folgenden Bearbeitung des GeometryArrays geworfen werden und leider nicht immer direkt auf die wahre Ursache des Problems hinweisen.

```
void setNormals(int index,float[] normals,)
void setNormals(int index,Vector3f[] normals)
void setNormals(int index,float[] normals,int start,int length)
4110 void setNormals(int index,Vector3f[] normals,int start,int length)
und
void setNormalIndices(int index,int[] normalIndices)
```

stehen in direkten Zusammenhang mit dem Flag **NORMALS**. Die Methoden selbst funktionieren in exakt der gleichen Weise wie die vorhergehend beschriebenen.

4115 `setNormals()` dient dazu, die Normal-Koordinaten selber zu übergeben, während mit `setNormalIndices()` die Verweise auf die Koordinaten, die dann die jeweiligen Dreiecke beschreiben, zu übergeben. Auch hier darf wiederum keine Diskrepanz zwischen der Anzahl der übergebenen Normal-Koordinaten und den Index-Nummern entstehen.

```
4120
void setColors(int index,byte[] colors)
void setColors(int index,Color3f[] colors)
void setColors(int index,Color3b[] colors)
void setColors(int index,byte[] colors,int start, int length)
4125 void setColors(int index,Color3f[] colors,int start, int length)
void setColors(int index,Color3b[] colors,int start, int length)
```

Diese Methoden beziehen sich auf das mögliche Flag **COLOR\_3** und übergeben Arrays, die die Farbwerte enthalten. Da die Parameter exakt die gleiche Bedeutung haben, wie in den vorhergehend beschriebenen Methoden, soll hier nicht erneut darauf eingegangen werden. Zu beachten ist, dass bei der Verwendung eines Byte-Arrays jeweils drei aufeinanderfolgende Bytes einen Farbwert bilden, wobei je eines immer für die Farbanteile r (rot), g (grün) und b (blau) steht.

4130

```
void setColors(int index,byte[] colors)
4135 void setColors(int index,Color4f[] colors)
void setColors(int index,Color4b[] colors)
void setColors(int index,byte[] colors,int start, int length)
void setColors(int index,Color4f[] colors,int start, int length)
void setColors(int index,Color4b[] colors,int start, int length)
```

4140        Wie an einigen der Datentypen für die Farbe zu sehen, beziehen sich diese Methoden auf das Flag **COLOR\_4**. Bei identischer Funktionsweise ist lediglich bei der Verwendung eines Byte-Array zu beachten, dass sich eine Farbe jetzt aus jeweils vier Bytes zusammensetzt. Neben den bekannten Farbanteilen r, g und b kommt hier noch ein Alpha-Wert hinzu, der eine Transparenzinformation beinhaltet.

4145  
`void setColorIndices(int index,int[] colorIndices)`

      Diese Methode ist für beide Möglichkeiten von Farbdatentypen wiederum gleich. Sie spezifiziert in alt bekannter Weise wiederum die Indices, die mit Verweis auf die Farbwerte die Dreieck eindeutig beschreiben.

4150  
`void setTextureCoordinates(int texCoordSet,int index,float[] texCoords)`  
`void setTextureCoordinates(int texCoordSet,int index,TexCoord2f[] texCoords)`

4155 `void setTextureCoordinates(int texCoordSet,int index,float[] texCoords,int start,int length)`  
`void setTextureCoordinates(int texCoordSet,int index,TexCoord2f[] texCoords,int start,int length)`

      Wie zu erwarten beziehen sich diese Methoden auf das zuvor bereits erwähnte Flag **TEXTURE\_COORDINATE\_2**. Neben den bereits bekannten Parametern findet sich hier ein neuer. Mit `texCoordSet` wird festgelegt, für welches Set von Texturkoordinaten die anschließend übergebenen Koordinaten gelten. Wie der Name bereits sagt, können innerhalb eines Geometry-Objektes mehrere Texturkoordinaten-Sets festgelegt und verwendet werden. Da das zu den fortgeschrittenen Techniken zählt, genügt es in der Regel, nur einen Satz Texturkoordinaten zu verwenden und für diesen Parameter 0 zu übergeben.

4165 Eine zweidimensionale Textur dürfte ebenfalls der Regelfall sein. Diese bezieht sich auf das, was normalerweise für die Texturierung von Objekten verwendet wird: ein Bild. Dieses hat mit Breite und Höhe genau zwei Dimensionen und benötigt demzufolge auch nur zwei Koordinatenwerte.

4170 Wird ein Array aus floats übergeben, so wird nach dem gleichen Prinzip wie in den vorhergehend beschriebenen Methoden verfahren. Je zwei aufeinanderfolgende floats bilden immer ein Koordinatenpaar bestehend aus X- bzw. U- und Y- bzw. V-Koordinate.

4175 `void setTextureCoordinates(int texCoordSet,int index,float[] texCoords)`  
`void setTextureCoordinates(int texCoordSet,int index,TexCoord3f[] texCoords)`  
`void setTextureCoordinates(int texCoordSet,int index,float[] texCoords,int start,int length)`  
4180 `void setTextureCoordinates(int texCoordSet,int index,TexCoord3f[] texCoords,int start,int length)`

      Hierzu ist sicher nichts weiter zu sagen, da die Methoden nach exakt dem gleichen Prinzip Hand zu haben sind wie die vorangegangenen. Diese gelten lediglich für das Flag



4185 **TEXTURE\_COORDINATE\_3** und damit für dreidimensionale Texturen. Auch die Verwendung eines float-Arrays bietet nichts neues, außer, dass hier immer drei aufeinanderfolgende Werte ein zusammengehörendes Koordinatentriplett bilden.

```
void setTextureCoordinates(int texCoordSet,int index,float[]
4190 texCoords)
```

```
void setTextureCoordinates(int texCoordSet,int index,TexCoord4f[]
texCoords)
```

```
void setTextureCoordinates(int texCoordSet,int index,float[]
texCoords,int start,int length)
```

```
4195 void setTextureCoordinates(int texCoordSet,int index,TexCoord4f[]
texCoords,int start,int length)
```

Um nicht alles bereits gesagt noch einmal wieder zu käuen hier nur der Hinweis auf die wiederum identische Funktionalität der Methoden und auf die Tatsache, dass diese für vierdimensionale Texturen gelten und damit im Zusammenhang mit dem Flag

4200 **TEXTURE\_COORDINATE\_4** stehen.

```
void setTextureCoordinateIndices(int texCoordSet,int index,int[]
texCoordIndices)
```

4205 Diese Methode ergibt sich zwangsläufig aus dem bisher Bekannten. Sie legt wiederum mit Hilfe der Indices fest, welche Texturkoordinatenwerte jeweils ein Dreieck bilden sollen. Auch hier findet sich wieder der zusätzliche Parameter `texCoordSet`, der das Texturkoordinaten-Set spezifiziert, für das diese Indices gelten sollen. Da die Index-Arrays nur ein Verweis auf die eigentlichen Koordinatenwerte sind, sind sie von deren Dimensionalität nicht abhängig. Somit ist diese Methode für die Flags

4210 **TEXTURE\_COORDINATE\_2**, **TEXTURE\_COORDINATE\_3** und **TEXTURE\_COORDINATE\_4** in gleicher Weise zu verwenden.

Damit sind alle wichtigen Methoden bekannt, die benötigt werden, um die Geometriedaten an ein `IndexedGeometryArray`-Objekt zu übergeben. Auch wenn die Ableitungsverhältnisse dieser Klasse bereits erwähnt wurden, soll auf einen vollständigen Überblick darauf an dieser Stelle keinesfalls verzichtet werden:

4215

```
java.lang.Object
```

```
 javax.media.j3d.SceneGraphObject
```

```
4220 javax.media.j3d.NodeComponent
```

```
 javax.media.j3d.Geometry
```

```
 javax.media.j3d.GeometryArray
```

```
 javax.media.j3d.IndexedGeometryArray
```

```
 javax.media.j3d.IndexedTriangleArray
```

4225

Wie zu sehen ist, findet sich hier die bereits erwähnte Verwandtschaften mit der Klasse `GeometryArray` (von der die `set...Coordinates()`-Methoden geerbt wurden) sowie

mit `IndexedGeometryArray` (die für die Existenz der `set...CoordinateIndices()`-Methoden verantwortlich zeichnet). Weiter oben im „Stammbaum“ findet sich aber auch  
4230 eine Klasse „`SceneGraphObject`“, die – wie bereits kennengelernte und ebenfalls von ihr Abgeleitete Node-Objekte beweisen – für die Capability Bits sowie die zugehörigen Methoden grundlegend verantwortlich ist. Demzufolge besitzt das `IndexedTriangleArray` zumindest einen Basissatz an Capabilities. Zu diesen kommt noch ein ganzer Schwung  
4235 weitere Capabilities hinzu, die von `GeometryArray` und `IndexedGeometryArray` geerbt wurden. Wie zu erwarten beziehen sich diese Capabilities auf die Fähigkeit, auch nach einem `compile()` bzw. wenn der zugehörige `SceneGraph` live ist, die verschiedenen Koordinaten und deren Indices lesen und schreiben zu können. Sollte das also nötig sein, so müssen die Fähigkeiten mit Hilfe ein- oder mehrfacher Aufrufe von `setCapability()` als weiterhin vorhanden angegeben werden.

4240

## 8.4.2 Das Shape3D-Beispielprogramm

Nach so viel grauer Theorie und ganzen Bergen trockener Konstruktoren- und Methodenbeschreibungen wird es jetzt mehr als Zeit, das eben gelernte praktisch  
4245 einzusetzen und es vor allem auf seinen Wahrheitsgehalt hin zu überprüfen. Denn irgend welche Behauptungen aufstellen kann schließlich jeder und so möchte sich auch der Autor nur an der Praxistauglichkeit des beschriebenen messen lassen.

Das Beispiel soll dabei auf dem eben beschriebenen `IndexedTriangleArray` beruhen und  
4250 Texturkoordinaten, Normals und Vertex-Farben vorerst außen vor lassen. Da diese in praktisch identischer Weise zu handhaben sind wie die im folgenden Beispiel ausschließlich verwendeten Vertices und die zugehörigen Indices, dürfte es keine große Schwierigkeit darstellen, das Beispielprogramm um die zusätzlichen Informationen und Funktionalitäten zu erweitern.

4255

Als Basis für das neue Beispiel soll der Code aus dem Abschnitt über die Appearance dienen. Hier ist bereits alles nötige enthalten: Es wird ein Universum erzeugt, diesem wird ein wenig Umgebungslicht hinzugefügt und das dort verwendete Primitive vom Typ  
4260 `Sphere` erhält ein wenig Farbe sowie ein plastisches Aussehen. Die vorzunehmenden Änderungen beziehen sich also auf diese `Sphere`. Statt ihrer soll nun ein `Shape3D`-Objekt innerhalb einer eigenen Methode `createShape3D()` erzeugt werden. Die Objekteigenschaften werden diesem `Shape3D`-Objekt anschließend mittels `setAppearance()` hinzugefügt und es wird in den `SceneGraph`en mittels `addChild()` eingehängt. Die Methode `createSceneGraph()` sieht anschließend dann also so aus:

4265

```
(1) BranchGroup createSceneGraph()
(2) {
(3) BranchGroup RootBG=new BranchGroup();
(4) TransformGroup CObjTG=new TransformGroup();
4270 (5) Transform3D CObjT3D=new Transform3D();
(6) Appearance CObjAppearance=new Appearance();
(7) AmbientLight ALgt=new AmbientLight(new Color3f(1f,1f,1f));
```

```

(8) DirectionalLight DLgt=new DirectionalLight(new Color3f(1f,1f,1f),new
 Vector3f(0.5f,0.5f,-1f));
4275 (9) BoundingSphere BigBounds=new BoundingSphere(new Point3d(),100000);
 (10)
 (11) Shape3D ComplexObj;
 (12)
 (13) ALgt.setInfluencingBounds(BigBounds);
4280 (14) DLgt.setInfluencingBounds(BigBounds);
 (15) CObjT3D.setTranslation(new Vector3f(0f,0f,-1.5f));
 (16) CObjT3D.setRotation(new AxisAngle4f(0f,0f,1f,
 (float)Math.toRadians(120)));
 (17) CObjT3D.setScale(1.8);
4285 (18) CObjTG.setTransform(CObjT3D);
 (19) CObjAppearance.setMaterial(new Material(new Color3f(0f,0f,1f),new
 Color3f(0f,0f,0f),new Color3f(1f,0f,0f),new Color3f(1f,1f,1f),100f));
 (20)
 (21) ComplexObj=createShape3D();
4290 (22) ComplexObj.setAppearance(CObjAppearance);
 (23) CObjTG.addChild(ComplexObj);
 (24)
 (25) RootBG.addChild(CObjTG);
 (26) RootBG.addChild(ALgt);
4295 (27) RootBG.addChild(DLgt);
 (28) RootBG.compile();
 (29) return RootBG;
 (30) }

```

4300 Neues findet sich in Zeile 11 mit der Definition des Shape3D-Objektes und in Zeile 21 mit der gleich näher zu betrachtenden Methode `createShape3D()`, die das Shape3D-Objekt erzeugt samt der nötigen Geometriedaten erzeugt und diese auch hinzufügt. In Zeile 22 schließlich wird eine Appearance für das Shape3D-Objekt gesetzt bevor dieses in Zeile 23 dann in den SceneGraphen eingeklinkt wird.

4305

Bei näherer Betrachtung fällt auch auf, dass das 3D-Objekt in den Zeilen 15 und 16 ein wenig verschoben und rotiert und in Zeile 17 schließlich sogar etwas vergrößert wird. Das dient hier jedoch lediglich einer besseren Darstellung des 3D-Objektes. Schließlich ist der Code für den KeyBehavior in diesem Code nicht mehr enthalten, so dass es nicht mehr

4310 möglich ist, ein paar Schritte näher an das Objekt heranzutreten.

Die wirklich neuen Funktionalitäten finden sich schließlich in der Methode `createShape3D()`. Die nötigen Geometriedaten sind hart codiert in Form von nicht gerade kleinen Arrays enthalten, weswegen sie hier etwas gekürzt dargestellt werden soll.

4315 Die Zeiten von seitenfüllenden Programmlistings, die mühsam abgetippt werden müssen sind ja schließlich und glücklicherweise endgültig vorbei:

```

(1) Shape3D createShape3D()
(2) {
4320 (3) Shape3D S3D=new Shape3D();
(4) IndexedTriangleArray TriArr=new
 IndexedTriangleArray(13,IndexedTriangleArray.COORDINATES,66);
(5) Point3f[] CoordArr=new Point3f[13];
(6) int[] Vind=new int[66];
4325 (7)
(8) CoordArr[0]=new Point3f(-0.02f, -0.000394f, -0.02f);
(9) ...
(10) CoordArr[12]=new Point3f(0.06f, -0.749808f, -0.06f);
(11)
4330 (12) Vind[0]=12; Vind[1]=4; Vind[2]= 11;
(13) ...
(14) Vind[63]= 9; Vind[64]= 12; Vind[65]= 8;
(15)
(16) TriArr.setCoordinates(0,CoordArr);
4335 (17) TriArr.setCoordinateIndices(0,Vind);
(18) S3D.addGeometry(TriArr);
(19) return S3D;
(20) }

```

- 4340 In Zeile 3 wird die Variable für das Shape3D-Objekt definiert und das Objekt auch gleich erzeugt. In Zeile 4 geschieht das gleiche mit dem IndexedTriangleArray. Wie an den übergebenen Parameter zu sehen, sind 13 Vertices (also 13 Koordinatentriplets) sowie 66 Indices zu erwarten. Da es sich um Dreiecke handelt, sagt dieser Wert also klar aus, dass offenbar 22 Dreiecke benötigt werden, um das gewünschte 3D-Objekt darzustellen.
- 4345 Weiterhin wird mit dem Flag **COORDINATES** festgelegt, dass nur die Vertexkoordinaten und keine weiteren Informationen wie z.B. für Normals, Texturen oder Farben verwendet werden sollen.

- 4350 In den beiden folgenden Zeilen 5 und 6 werden schließlich die Arrays angelegt. Das Koordinatenarray für die Vertices hat dabei die zu erwartende Länge von 13. Dieses wird in den (gekürzt dargestellten) Zeilen 8 bis 10 mit Objekten vom Typ Point3f gefüllt, die ebenfalls die nötigen Koordinatenwerte mitbringen.

- 4355 Das Index-Array Vind wird in Zeile 6 mit einer Länge von 66 erzeugt und in den (ebenfalls gekürzt dargestellten) Zeilen 12 bis 14 mit Indexwerten gefüllt, die auf die Vertices oder genauer auf die Arraypositionen der Vertices des zuvor erzeugten Arrays CoordArr verweisen.

Damit wären die Geometriedaten komplett vorhanden und müssen nur noch dem

4360 IndexedGeometryArray-Objekt `TriArr` übergeben werden. Das geschieht in den Zeilen  
16 und 17 mittels der Methoden `setCoordinates()` (bei der das `Point3f`-  
Vertexkoordinatenarray übergeben wird) bzw. `setCoordinateIndices()` (hier wird das  
Array mit den darauf verweisenden Indexwerten übergeben). In beiden Fällen ist der Wert  
4365 für Parameter `index` gleich 0, schließlich sollen ja nicht nur Teilinformationen übergeben  
werden, sondern die vollständigen Geometriedaten.

In Zeile 18 schließlich wird das `IndexedTriangleArray` mittels `addGeometry()` dem  
`Shape3D`-Objekt `s3D` übergeben und selbiges zur weiteren Bearbeitung als Returnwert an  
die aufrufende Methode zurückgeliefert.

4370

Wird das Beispielprogramm nun kompiliert und ausgeführt, so ist das erste mal ein etwas  
komplexeres 3D-Objekt zu sehen, das nicht auf der Klasse `Primitive` basiert. Es ist zwar  
nicht überragend komplex, aber der hier dargestellte, dreidimensionale und viereckige  
Pfeil sollte für eine Demonstration der Fähigkeiten, die `Shape3D` und `GeometryArray`  
4375 bieten, genügen.

### 8.4.3 TriangleArray

Mit dem `IndexedTriangleArray` wurde dieses mal eigentlich eine ziemlich komplexe Klasse  
4380 zuerst beschrieben. Das macht in dem Fall aber durchaus Sinn, da die allermeisten  
Grafikformate nach dem Prinzip der indirekten Polygone aufgebaut sind. Dass heißt, man  
wird öfter Vertices und Indices finden, die die dazugehörigen Polygone beschreiben, als  
alles andere. Aus gutem Grund: Indices sind in der Regel nur vier Byte lang, während ein  
Vertex – je nach dem, ob er sich aus floats oder aus doubles zusammensetzt – 12 oder 24  
4385 Byte lang ist. Wird der selbe Vertex also nur oft genug als Eckpunkt für verschiedene  
Polygone benötigt, gleicht sich der Nachteil, dass diese vier Bytes eigentlich zusätzlich  
nötig sind, ziemlich schnell aus. Die Methode der Indizierung kann also einiges an Daten  
sparen.

4390 Das Gegenstück dazu findet sich im einfachen `TriangleArray`. Hier wird nicht mit Indices  
auf die Koordinatenwerte verwiesen. Demzufolge setzt sich ein Dreieck immer direkt aus  
drei aufeinanderfolgenden Koordinatentriplets zusammen.

Der Konstruktor kennt demzufolge auch keinen Parameter `indexCount`:

4395

```
TriangleArray(int vertexCount,int vertexFormat)
```

Mit `vertexCount` wird in bekannter Weise die Anzahl der Vertices festgelegt,  
`vertexFormat` legt wieder fest, welche Geometriedaten in dem Objekt enthalten sein  
4400 sollen, es kommen dabei die gleichen Flags wie auch beim `IndexedTriangleArray` zum  
Einsatz.

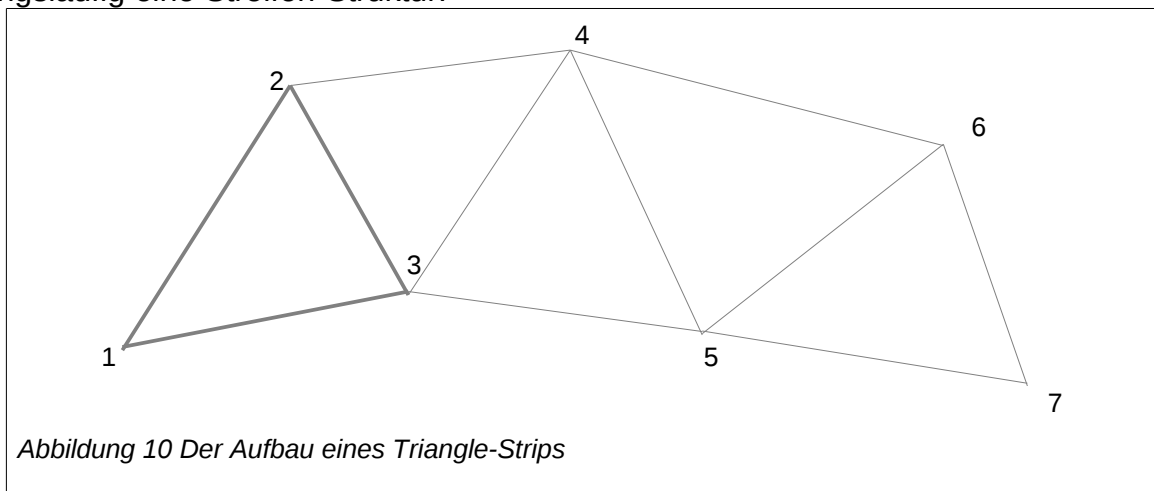
Des weiteren stellt das `TriangleArray` die gleichen Methoden `setCoordinates()` zum setzen der Koordinaten zur Verfügung wie bereits oben beschrieben. Das Gleiche gilt für  
4405 Normals (`setNormals()`), Texturkoordinaten (`setTextureCoordinates()`) und  
Farben (`setColors()`).

Die `set...Indices()`-Methoden sind in dieser Klasse nicht enthalten, da die Dreiecke  
4410 schließlich direkt über die Vertices definiert werden und nicht mehr indirekt über die  
Indices.

## 8.4.4 `IndexedTriangleStripArray` und `TriangleStripArray`

Die beiden Klassen `IndexedTriangleStripArray` und `TriangleStripArray` sind praktisch  
4415 genau so zu verwenden wie `IndexedTriangleArray` und `TriangleArray`. Der Unterschied zu  
diesen besteht jedoch in der Art der internen Verwendung der Daten und damit auch, wie  
viele aufeinander folgende Vertices bzw. Indices ein Dreieck bilden.

Bei den vorhergehend beschriebenen `TriangleArrays` konnten die Dreiecke beliebig  
4420 angeordnet sein, immer je drei Datensätze (drei Vertices beim `TriangleArray` und drei  
Indices beim `IndexedTriangleArray`) bildeten genau ein Dreieck. Das läßt jedoch Raum für  
Optimierungen. Wie der Name dieser neuen beiden Klassen bereits sagt, sind sie  
Dreiecke dieses mal streifenförmig angeordnet. Das heißt, die ersten drei Indices/Vertices  
4425 definieren das erste Dreieck, jedes weitere Dreieck besteht dann aber aus nur noch einem  
neuen Index/Vertex sowie den beiden vorhergehenden. Da die Dreiecke auf diese Art  
immer eine gemeinsame Seite mit dem vorhergehenden Dreieck haben, entsteht  
zwangsläufig eine Streifen-Struktur.



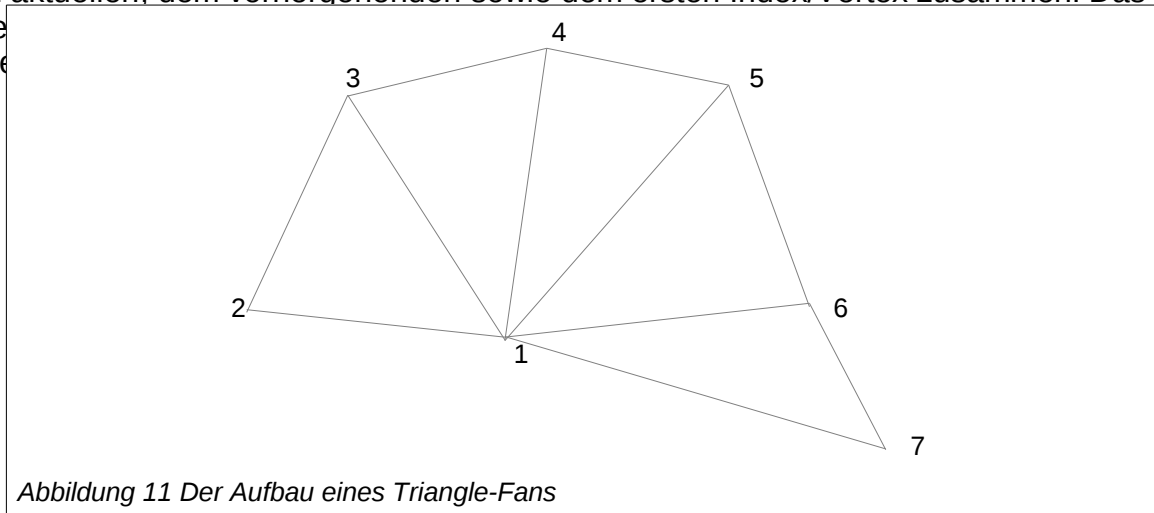
4430 Was ist nun der Vorteil dieser Methode? Nun zum einen kann Speicherplatz gespart  
werden, da abgesehen vom ersten Dreieck jedes weitere nur noch einen Index bzw. einen  
Vertex benötigt, statt drei. Zum anderen kommt diese Struktur dem Renderer entgegen,  
d.h. die Grafikhardware wird durch einfachere Zeichenoperationen entlastet, was sich in  
höheren Frameraten niederschlägt.

4435

Da die zur Verfügung gestellten Methoden mit denen der Klassen `IndexedTriangleArray` und `TriangleArray` identisch sind, soll auf ein stupides Wiederholen der Beschreibung verzichtet werden

## 4440 8.4.5 `IndexedTriangleFanArray` und `TriangleFanArray`

Zwei weitere in ihrem Aufbau optimierte Klassen sind das `IndexedTriangleFanArray` und das `TriangleFanArray`. Hier sieht der Aufbau so aus, dass die ersten drei Indices/Vertices wieder das erste Dreieck festlegen. Jedes weitere Dreieck benötigt wiederum nur einen weiteren Index bzw. einen weiteren Vertex. Nur dieses mal setzt dieses sich dann aus dem aktuellen, dem vorhergehenden sowie dem ersten Index/Vertex zusammen. Das Ergebnis sieht dann so aus:



4450

## 8.4.6 `IndexedQuadArray` und `QuadArray`

Zwei weitere wichtige Klassen für Geometriedaten sind das `IndexedQuadArray` sowie das `QuadArray`. Im Beispiel am Anfang dieses Kapitels wurde ein einfaches Objekt dargestellt, das sich wahlweise aus zwei Dreiecken oder aber auch aus einem Viereck zusammensetzen ließ.

Genau das erlauben die Klassen `IndexedQuadArray` sowie `QuadArray`. Sie bieten die Möglichkeit, Vierecke, also die so genannten „Quads“ zu zeichnen. Dass das möglich ist, erfordert jedoch einige Randbedingungen. So darf dieses Quad beispielsweise nicht konkav sein. Des weiteren muß es in einer Ebene liegen, es darf also keinen „Knick“ haben der von einem Eckpunkt zum gegenüberliegenden verläuft. Sind diese Bedingungen erfüllt, steht diesem Polygon nichts mehr im Wege. Mit den Quads ist bei der aktuellen Version von Java 3D allerdings Schluss. Polygone mit mehr als vier Eckpunkten werden nicht direkt unterstützt. Findet sich innerhalb eines Modells also ein Konstrukt in der Art

polygon 7 12 4 22 10 7 6 9

4470 wie es z.B. im RWX-Format vorkommen kann, so muß dieses Siebeneck manuell  
beispielsweise in Dreiecke zerlegt werden. Das ist allerdings eine recht einfache Aufgabe  
und das Ergebnis geradezu prädestiniert für ein IndexedTriangleFanArray.

4475 Der Vorteil von Quads ist eigentlich die höhere Effizienz, weil eben nur noch eine Fläche  
(also ein Viereck) gezeichnet werden muß, statt zwei (also zwei Dreiecke, die zusammen  
ein Viereck bilden). „Eigentlich“ deswegen, weil es nicht immer unterstützt wird. OpenGL  
kennt Quads und zeichnet diese mit der zu erwartenden höheren Effizienz. DirectX kann  
dies jedoch nicht. Das heißt, wenn Java 3D für DirectX zum Einsatz kommt, so können  
4480 IndexedQuadArrays und QuadArrays natürlich verwendet werden, sie werden dank des  
eingeschränkten Funktionsumfanges von DirectX anschließend nur wieder in Dreiecke  
zerlegt.

4485 Das soll jedoch kein Grund sein, Quads nicht zu verwenden. Schließlich ist Java 3D eine  
plattformunabhängige API. Applikationen können unter Windows und damit auch unter  
dem ineffizienteren DirectX ausgeführt werden, das muß aber nicht sein. Für den Fall,  
dass die Applikation auch auf anderen Betriebssystemen und damit auf anderen  
Grafikschichten zum Einsatz kommt, sollten also nach Möglichkeit die QuadArrays  
verwendet werden.

## 4490 8.4.7 Sonstige GeometryArrays

Mit den oben beschriebenen GeometryArrays und IndexedGeometryArrays, die in der  
Lage sind, Dreiecke und Vierecke für eine Darstellung innerhalb eines Shape3D zu  
speichern, sind die wichtigsten Klassen für diesen Zweck beschrieben worden.

4495 Daneben gibt es noch einige weitere GeometryArrays, die hier nur kurz erwähnt werden  
sollen. Sie werden nach dem gleichen Prinzip verwendet, wie alle zuvor bereits  
beschriebenen, so dass es an dieser Stelle sicher genügt, nur kurz deren Sinn und  
Funktion zu erläutern.

4500 Das **IndexedLineArray** sowie das **LineArray** stellen Linien dar. Dabei bilden immer zwei  
aufeinander folgende Index- bzw. Vertex-Paare die Endpunkte genau einer Linie. Mit  
diesen Objekten lassen sich also Gitterstrukturen ähnlich einem Drahtgittermodell  
erstellen (diese sind jedoch nicht zu verwechseln mit der Drahtgitterdarstellung, die mit  
4505 Hilfe der PolygonAttributes in einem Appearance-Objekt möglich ist).

**IndexedLineStripArray** und **LineStripArray** erweitern diese beiden Klassen um die  
bekannte Strip-Darstellung. Das heißt, hier bilden die beiden ersten Indices/Vertices die  
erste Linie, während sich alle weiteren Linien dann nur noch aus dem vorhergehenden  
4510 Index/Vertex sowie einem weiteren zusammensetzen. Das Ergebnis ist zwangsläufig eine  
längere, durchgehende, aber nicht unbedingt gerade Linie.



4515 IndexedPointArray sowie PointArray gehen wiederum noch einen Schritt weiter (zurück)  
und dienen nur noch der Darstellung einzelner Punkte. Für ein IndexedPointArray ergibt  
sich hier kein Fortschritt mehr in Bezug auf den Speicherverbrauch, da hier je ein Index  
genau einem Vertex zugeordnet sein dürfte. Sind mehrere Indices dem gleichen Vertex  
zugeordnet, so bedeutet dies nur, das der gleiche Punkt mehrfach gezeichnet wird. Die  
Daten sind dann von vorne herein also schon alles andere als optimal. Da jedoch die  
indizierte, indirekte Darstellung von Geometriedaten auch für Punkte möglich sein muß,  
4520 ist die Klasse IndexedPointArray natürlich trotzdem nicht sinnlos.

### 8.4.8 OrientedShape3D

4525 Nach dem die Möglichkeiten, verschiedenste Geometriedaten zu verwenden in den  
vergangenen Abschnitten fast schon exzessiv beschrieben wurden, soll nun ein Blick auf  
eine andere, ebenfalls sehr interessante Klasse geworfen werden. Sie leitet sich vom  
bereits bekannten Shape3D-Node ab, der ja die Geometriedaten innerhalb der  
GeometryArrays aufnimmt, ihnen eine Appearance zuordnet und dann in einen  
SceneGraph eingebunden werden kann. Bei einem Shape3D-Objekt war es nun so, dass  
4530 seine Position und Lage innerhalb einer virtuellen Welt eindeutig beschrieben wurde. Das  
kann durch ein oder mehrere TransformGroup/Transform3D-Kombinationen geschehen,  
die eine Position und Rotation festlegen können. Diese galten dann für das betroffene  
Shape3D-Objekt, so lange wie an der Transformation nichts geändert wurde.

4535 Das heißt, wenn man sich um ein solches Objekt herumbewegen würde, so könnte man  
das gleiche beobachten, wie in der realen Welt auch: Man sieht die anderen Seiten des  
3D-Objektes.

4540 Nicht so jedoch, wenn die Geometriedaten einem OrientedShape3D-Objekt zugeordnet  
wurden. Dieses hat eine für die reale Welt verstörende Eigenschaft: es wendet sich nie  
vom Betrachter ab, es zeigt ihm immer seine Vorderseite, die durch die lokale Z-Achse  
des 3D-Objektes definiert wird. Das heißt, ein Beobachter kann ein solches  
OrientedShape3D-Objekt umrunden und umkreisen wie er mag, er wird nie dessen Seiten  
oder Rückseite zu sehen bekommen. Das Ergebnis wäre ein endloses Spiel, vergleichbar  
4545 dem Hund, der versucht, seinen eigenen Schwanz zu fangen (Simpsons-Fans mögen  
sicher den Vergleich mit Homer lieber, der sich einst am Fußboden liegend um die eigene  
Achse drehte, um lesen zu können, was auf seinem Hinterkopf steht).

4550 Die Klasse OrientedShape3D bietet im Vergleich zum Shape3D einige Möglichkeiten und  
Optionen mehr an:

```
OrientedShape3D(Geometry geometry, Appearance appearance, int mode,
Vector3f axis, boolean constantScaleEnable, double scale)
```

4555 Die ersten beiden Parameter sind schon von der Klasse Shape3D her bekannt. Hier  
werden die Geometriedaten sowie ein Appearance-Objekt übergeben, die wieder für alle

wesentlichen Eigenschaften des darzustellenden 3D-Objektes zuständig sind. Der erste neue Parameter, `mode`, legt fest, wie sich das `OrientedShape3D`-Objekt bewegen soll, damit es immer dem Betrachter zugewandt bleibt. Wird hier **ROTATE\_ABOUT\_AXIS** angegeben, so rotiert das 3D-Objekt um eine festgelegte Achse. Bei **ROTATE\_ABOUT\_POINT** hingegen erfolgt die Rotation um einen bestimmten Punkt mit einer zusätzlichen Rotation um die Y-Achse des 3D-Objektes auf die Y-Achse des Beobachters auszurichten. Um welche Achse rotiert werden soll, wird dann wiederum mit dem Parameter `axis` angegeben. Das hier zu übergebende `Vector3f`-Objekt spezifiziert einen Strahl, um den das `OrientedShape3D`-Objekt rotiert. Des weiteren kann für den Parameter `mode` auch **ROTATE\_NONE** angegeben werden, dann rotiert das `OrientedShape3D`-Objekt gar nicht und verhält sich damit wie ein ganz normales `Shape3D`-Objekt.

Wird mit `constantScaleEnable` der vorletzte Parameter auf `true` gesetzt, so ist es damit möglich, Transformationen oder besser Skalierungen oberhalb dieses Objektes im zugehörigen `SceneGraph` unwirksam werden zu lassen. Statt dessen kommt dann der letzte Parameter des Konstruktors zum Einsatz und das 3D-Objekt wird nicht wie gewohnt auf eine Größe skaliert, die sich aus den übergeordneten Transformationen ergibt, sondern ausschließlich auf die mittels `scale` festgelegte Größe.

Soll der Modus **ROTATE\_ABOUT\_POINT** verwendet werden, so macht es Sinn, einen anderen Konstruktor zu verwenden:

```
4580 OrientedShape3D(Geometry geometry, Appearance appearance, int mode,
Point3f point)
```

Hier bietet der letzte Parameter die Möglichkeit, den Punkt anzugeben, um den unter Zuhilfenahme der X- und Y-Achse rotiert werden soll, da der `Vector3f axis` aus dem vorhergehend beschriebenen Konstruktor für diesen Modus schließlich keinen Sinn gehabt hätte. Statt dessen kann aber auch eine der folgenden Methoden zum Einsatz kommen, die es erlauben, die Koordinaten des Punktes, um den rotiert werden soll separat zu setzen:

```
void setRotationPoint(float x, float y, float z)
4590 void setRotationPoint(Point3f point)
```

Neben diesen kennt diese Klasse noch die folgenden Methoden:

```
void getAlignmentAxis(Vector3f axis)
4595 void setAlignmentAxis(float x, float y, float z)
void setAlignmentAxis(Vector3f axis)
```

Diese Methoden sind für den Alignment-Modus **ROTATE\_ABOUT\_AXIS** wichtig. Sie erlauben es, den aktuellen Alignment-Axis-Wert zu ermitteln oder aber einen neuen festzulegen. Dieser gibt an, um welche Achse das `OrientedShape3D`-Objekt bei einer Positionsänderung des Betrachters rotiert werden soll. Die spezifizierte Achse darf dabei

nicht parallel zur Z-Achse sein ( (0,0,z) für alle Werte von z).

```
int getAlignmentMode()
```

```
void setAlignmentMode(int mode)
```

4605 Der Alignment-Modus selbst wird mit diesen Methoden verwaltet. Sie liefern den aktuell verwendeten Wert zurück bzw. erlauben es, einen neuen zu setzen. Dabei finden die bereits erwähnten Konstanten **ROTATE\_ABOUT\_AXIS**, **ROTATE\_ABOUT\_POINT** und **ROTATE\_NONE** Verwendung.

```
4610 boolean getConstantScaleEnable()
```

```
void setConstantScaleEnable(boolean constantScaleEnable)
```

Auch diese Funktionalität ist vom Konstruktor her bereits bekannt. Das `constantScaleEnable` flag sagt aus, ob ein eigener Scale-Faktor exklusiv verwendet werden soll (`true`) oder ob die normale Verhaltensweise verwendet werden soll, bei der die Scale-Faktoren der übergeordneten Transformationen den Gesamt-Scalefaktor des `OrientedShape3D`-Nodes festlegen (`false`). Mit diesen Methoden läßt sich der aktuelle Status dieses Flags ermitteln beziehungsweise es ist möglich, dieses auf einen neuen Wert zu setzen.

```
4620 double getScale()
```

```
void setScale(double scale)
```

Ist dieses Flag `constantScaleEnable true`, also wird ein eigener Scalefaktor verwendet, so wird dessen aktueller Wert von `getScale()` zurückgeliefert, während dieser Faktor mit `setScale()` verändert werden kann.

4625

```
void getRotationPoint(Point3f point)
```

```
void setRotationPoint(float x, float y, float z)
```

```
void setRotationPoint(Point3f point)
```

Diese Methoden bilden das Gegenstück zu den Methoden `get/setAxisAlignment()` für den Alignment-Modus **ROTATE\_ABOUT\_POINT**. Sie behandeln den Punkt im virtuellen 3D-Raum, um den sich das `OrientedShape3D`-Objekt drehen soll, um immer dem Betrachter zugewandt zu sein. Mit ihnen ist es möglich, die aktuellen Koordinaten dieses Punktes zu ermitteln oder aber neue Koordinaten für ihn festzulegen.

4635

Da `OrientedShape3D` keine sonderlich exotischen Capabilities kennt, sondern nur einige, die das setzen und lesen der verschiedenen eben besprochenen Daten, Eigenschaften und Modes erlauben oder als häufig verwendet markieren, soll abschließend nur noch ein Blick auf die Ableitung dieser Klasse geworfen werden, die – wie zu erwarten – eine direkte Unterklasse des ebenfalls schon beschriebenen `Shape3Ds` ist:

4640

```
java.lang.Object
```

```

 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
4645 javax.media.j3d.Leaf
 javax.media.j3d.Shape3D
 javax.media.j3d.OrientedShape3D

```

### 8.4.8.1 Billboard

4650

Auch wenn die Klasse Billboard eigentlich zu den Behaviors gehört, soll sie doch an dieser Stelle erwähnt werden. Wenn schon nicht durch ihre Ableitung so läßt sie sich in diesem Fall durch ihre Eigenschaften thematisch sehr gut direkt hinter dem OrientedShape3D einordnen. Der Billboard Behavior macht nämlich ziemlich genau das gleiche, wie das OrientedShape3D – er sorgt dafür das ein 3D-Objekt oder ein Teil-SceneGraph seine Z-Achse immer auf die Position des Views ausrichtet. Was ist nun der Unterschied zum OrientedShape3D? Nun einer wurde bereits genannt: Da es sich um einen Behavior handelt, der – wie sich gleich zeigen wird – in alt bekannter Weise auf eine TransformGroup wirkt, ist es möglich, statt nur zusammengehörender Geometriedaten (die dann ein 3D-Objekt ergeben) einen ganzen SceneGraphen vom Billboard Behavior kontrollieren zu lassen. Der zweite Unterschied ist, dass sich ein Billboard-Objekt besser kontrollieren und manipulieren läßt, als ein OrientedShape3D, da es sich hier um aktiven (Java-)Code handelt.

4665 Ein Blick auf zwei der Konstruktoren zeigt deutliche Ähnlichkeiten mit dem OrientedShape3D:

```

Billboard(TransformGroup tg, int mode, Point3f point)
Billboard(TransformGroup tg, int mode, Vector3f axis)

```

4670

Der erste Parameter bezieht sich wie bei anderen Behaviors und Interpolatoren bereits gesehen wieder auf die TransformGroup, die beeinflusst werden soll. Dementsprechend müssen für diese wieder die Capabilities zum Lesen (**ALLOW\_TRANSFORM\_READ**) und Schreiben (**ALLOW\_TRANSFORM\_WRITE**) der zugehörigen Transformation gesetzt sein. Der zweite Parameter, `mode`, gibt an, wie sich der Billboard Behavior verhalten soll. Im Modus **ROTATE\_ABOUT\_AXIS** rotiert dieser nur um eine Achse, so dass die zugehörigen 3D-Objekte z.B. bei einer Änderung der Höhe des Views nicht mit ausgerichtet werden. In diesem Modus ist beim zweiten angegebenen Konstruktor mittels des Parameters `axis` der Strahl anzugeben, um den rotiert werden soll. Wird statt dessen der Modus **ROTATE\_ABOUT\_POINT** gewählt, so ist zweckmäßigerweise der erste Konstruktor zu verwenden, da hier mittels `point` angegeben werden kann, wo sich der Punkt befindet, um den in allen drei Achsen rotiert werden soll.

4685 Die Klasse bietet verschiedene, dem OrientedShape3D recht ähnliche Methoden an, allerdings finden sich auch einige, denen man die Verwandtschaft dieser Klasse mit den Behaviors ansieht:

```

void getAlignmentAxis(Vector3f axis)
void setAlignmentAxis(float x, float y, float z)
4690 void setAlignmentAxis(Vector3f axis)

```

In dem Fall, dass der Modus **ROTATE\_ABOUT\_AXIS** gewählt wurde, ist es mit diesen Methoden möglich, die Achse zu ermitteln oder aber neu festzulegen, um die sich der Behavior drehen soll. Diese Achse darf dabei ebenfalls nicht parallel zur Z-Achse sein (0,0,z) für alle Werte von z).

```

4695
int getAlignmentMode()
void setAlignmentMode(int mode)

```

Es wird entweder der aktuelle Alignment-Modus zurückgeliefert oder aber es kann ein neuer Modus gesetzt werden. Wie oben schon beschrieben kommen hier die Konstanten **ROTATE\_ABOUT\_AXIS** und **ROTATE\_ABOUT\_POINT** zum Einsatz.

```

void getRotationPoint(Point3f point)
void setRotationPoint(float x, float y, float z)
void setRotationPoint(Point3f point)

```

4705 Wird der Modus **ROTATE\_ABOUT\_POINT** verwendet, so ist es mit diesen Methoden möglich, die Position des Punktes, um den rotiert wird zu ermitteln oder aber neue Koordinaten für ihn festzulegen.

```

TransformGroup getTarget()
4710 void setTarget(TransformGroup tg)

```

Im Unterschied zum OrientedShape3D benötigt der Billboard-Behavior eine TransformGroup, deren Transformation verändert wird, um das gewünschte Verhalten zu erreichen. Dieses „Target“ wird mit Hilfe dieser beiden Methoden ermittelt oder neu festgelegt.

4715 Da bereits erwähnt wurde, dass Billboard ein Behavior ist, sind auch die Ableitungsverhältnisse dieser Klasse wenig überraschend:

```

java.lang.Object
4720 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Behavior
 javax.media.j3d.Billboard

```

4725

## 8.4.9 Normals

Bereits mehrfach wurden in den vergangenen Abschnitten die so genannten Normals angesprochen. Diese haben für den Aufbau einer Szene eine elementare Bedeutung, da ohne sie 3D-Objekte nicht in der Lage wären, Licht in einer realistisch erscheinenden Art und Weise zu reflektieren. Wie in den Abschnitten über die Geometriedaten gesehen, gibt es Möglichkeiten, bereits existierende Normals einem GeometryArray hinzuzufügen. Allerdings ist es leider so, dass diese nicht immer von Anfang an zur Verfügung stehen. Sei es, weil sie beim Exportieren eines 3D-Modelles nicht mitgespeichert wurden oder aber auch, weil es in einem 3D-Grafikformat gar nicht vorgesehen ist, diese Normals überhaupt mitzuliefern. In diesem Fall ist es notwendig, die Normals selber zu generieren.

Um das zu tun, ist eine Klasse aus dem Paket `com.sun.j3d.utils.geometry` nötig, der `NormalGenerator`. Wie der Name bereits sagt, ist das ein Hilfsmittel, um Normals zu erzeugen. Doch zuerst ein Blick auf die aus dem vorhergehenden Beispiel bekannte Methode `createShape3D()` die für diese Neuerung wiederum ein wenig abgeändert werden muß und auch hier wieder in verkürzter Form dargestellt werden soll:

```
(1)Shape3D createShape3D()
4745 (2) {
 (3) Shape3D S3D=new Shape3D();
 (4) Point3f[] CoordArr=new Point3f[8];
 (5) int[] Vind=new int[24];
 (6) GeometryInfo GInfo;
4750 (7) NormalGenerator NormGen=new NormalGenerator();
 (8) Stripifier GStrip=new Stripifier();
 (9)
 (10) CoordArr[0]=new Point3f(-1f,1f,-1f);
 (11) ...
4755 (12) CoordArr[7]=new Point3f(-0.5f,-1f,0.5f);
 (13)
 (14) Vind[0]=3; Vind[1]=2; Vind[2]=1; Vind[3]=0;
 (15) ...
 (16) Vind[20]=7; Vind[21]=3; Vind[22]=0; Vind[23]=4;
4760 (17)
 (18) GInfo=new GeometryInfo(GeometryInfo.TRIANGLE_ARRAY);
 (19) GInfo.setCoordinates(CoordArr);
 (20) GInfo.setCoordinateIndices(Vind);
 (21) GInfo.compact();
4765 (22) NormGen.generateNormals(GInfo);
 (23) GStrip.stripify(GInfo);
 (24) S3D.addGeometry(GInfo.getGeometryArray());
 (25) return S3D;
```

```
(26) }
```

4770

In den Zeilen 3 bis 8 werden wieder verschiedene Variablen deklariert und auch einige Objekte gleich erzeugt, die in der Folge benötigt werden. Das Vorhandensein des Stripifiers mag hier verwirren. In der Tat ist er für die Erzeugung von Normals nicht wirklich notwendig, da sich diese Klasse aber in der gleichen Package befindet und nicht weiter kompliziert ist, wird sie an dieser Stelle der Einfachheit halber gleich mit vorgestellt.

4775

Was jedoch auffällt, ist, dass jetzt nicht mehr der Weg über ein GeometryArray gegangen wird, um dem Shape3D die Geometriedaten zukommen zu lassen. Vielmehr wird hier eine neue Klasse GeometryInfo eingesetzt.

4780

#### 8.4.9.1 GeometryInfo

Wie praktisch alle in den `com.sun.j3d`-Packages enthaltenen Klassen ist auch GeometryInfo leider nicht wirklich detailliert spezifiziert worden. Es erschließt sich aber schnell, wie diese Klasse zu verwenden ist, da nur ein Konstruktor existiert, der demzufolge in Zeile 18 auch verwendet wird:

4785

```
GeometryInfo(int mode)
```

4790

Wie leicht zu erkennen ist, muß auch hier ein Modus angegeben werden. Dieser Bezieht sich auf die Art der Geometriedaten, die anschließend von diesem GeometryInfo-Objekt verarbeitet werden sollen. Es stehen als mögliche Werte **QUAD\_ARRAY** (wenn Quads verwendet werden sollen), **TRIANGLE\_ARRAY** (wenn die Geometriedaten Dreiecke beschreiben), **TRIANGLE\_STRIP\_ARRAY** (wenn die Dreiecke in Form eines Strips angeordnet sind, d.h. wenn ein neues Dreieck immer zwei Vertices mit dem vorhergehenden gemeinsam hat), **TRIANGLE\_FAN\_ARRAY** (wenn diese Dreiecke als Fan, also als Fächer angeordnet sind und damit immer einen gemeinsamen Start-Vertex verwenden) zur Auswahl. Interessanterweise existiert für die Klasse GeometryInfo ein weiterer Verarbeitungsmodus, der mit der Konstanten **POLYGON\_ARRAY** angegeben wird. Hier können also auch Geometriedaten übergeben werden, die echte Polygone mit mehr als vier Eckpunkten enthalten. Intern werden diese allerdings auch wieder in Dreiecke zerlegt, da Java 3D wie bereits erwähnt maximal Quads verarbeitet.

4795

4800

4805

GeometryInfo und die anderen Klassen, die ein Objekt dieses Typs benötigen setzen sogar im Falle des Modus **QUAD\_ARRAY** intern nur Dreiecke ein, die aber vor ihrer weiteren Verwendung wieder in Quads konvertiert werden, so fern das nach einer Umstrukturierung der Daten noch möglich und sinnvoll ist.

Drei der Methoden, die die Klasse GeometryInfo zur Verfügung stellt, werden dann in den Zeilen 19 bis 21 verwendet:

4810

```
setCoordinates(double[])
setCoordinates(float[])
```

```
setCoordinates(Point3de[])
```

```
setCoordinates(Point3f[])
```

- 4815 Diese Methode wird verwendet, um dem GeometryInfo die nötigen Vertexinformationen hinzuzufügen.

```
setCoordinateIndices(int[])
```

- 4820 Diese Methode wird verwendet, um die Indices, die zu den Vertexdaten gehören, festzulegen. Werden keine Indices übergeben, geht GeometryInfo offenbar davon aus, dass keine indizierten Daten zum Einsatz kommen und die Vertices die Polygone somit direkt beschreiben.

```
compact()
```

- 4825 Hier werden keinerlei weitere Parameter benötigt. Diese Methode komprimiert oder besser noch optimiert die Geometriedaten innerhalb des GeometryInfo-Objektes. Das heißt genauer, dass unbenutzte Daten (also z.B. Vertices, denen kein Index zugeordnet ist) entfernt werden.

- 4830 Die Klasse GeometryInfo kennt noch einige weitere Methoden, an Hand deren Namen sich ihre Funktion bereits erschließt. Ganz ähnliche Methoden sind bereits von den GeometryArrays her bekannt. Dort erfüllten diese den gleichen Zweck. Der Vollständigkeit halber sollen diese Methoden hier noch einmal aufgeführt werden:

4835 

```
setColors(Color3b[])
```

```
setColors(Color3f[])
```

```
setColors(Color4b[])
```

```
setColors(Color4f[])
```

```
setColors3(byte[])
```

4840 

```
setColors3(float[])
```

```
setColors4(byte[])
```

```
setColors4(float[])
```

Diese Methoden setzen per-Vertex-Farbinformationen, mit denen es möglich ist, unterschiedliche Farben für verschiedene Polygone zu definieren.

4845

```
setColorIndices(int[])
```

Wenn diese Farbinformationen indirekt über Indices angesprochen werden sollen, so ist es mit Hilfe dieser Methode möglich, die zugehörigen Indexwerte zu übergeben.

4850 

```
setNormals(float[])
```

```
setNormals(Vector3f[])
```

Hiermit ist es möglich, Normal-Daten zu übergeben, so diese bekannt sind. In





er konstruiert einen NormalGenerator mit Defaultwerten.

4895 NormalGenerator(double creaseAngle)

4900 Dieser Konstruktor ermöglicht es, einen Winkel in Radians anzugeben, der frei übersetzt auch als Knickwinkel bezeichnet werden könnte. Das hat mit einer weiteren Eigenschaft der Normals zu tun. Diese sind wie erwähnt zum einen dafür zuständig, dass Licht in einer realistisch wirkenden Art und Weise reflektiert werden kann. Genauer gesagt sorgen die Normals auch dafür, dass Flächen nicht nur in einer einheitlichen Farbe dargestellt werden können, sondern dass sie einen der Beleuchtung entsprechenden Helligkeitsverlauf bekommen (hier kommt das so genannte Gouraud-Shading zum Einsatz). Und an dieser Stelle kommt jener „Knickwinkel“ ins Spiel. Ist der Winkel zwischen zwei aneinander liegenden Polygonen kleiner als dieser Winkel, so werden sie als eine zusammenhängende Fläche behandelt und der Farbverlauf entsprechend berechnet.

4910 Was heißt das nun genau? Betrachtet man den Default-Wert dieses Knickwinkels von 0.76794487087750496 RAD (was etwa 43,9° entspricht), so heißt das, dass Flächen, die sich aus Polygonen zusammensetzen, zwischen denen der Winkel größer als 43,9° ist, wie bisher bekannt auch als separate, unterscheidbare Flächen dargestellt werden. Das bedeutet für den Pyramidenstumpf aus dem oberen Beispielpogramm, jede Seite ist klar durch eine Kante abgegrenzt.

4915 Was wäre nun aber, wenn das 3D-Objekt kein vierseitiger Pyramidenstumpf wäre, sondern ein Zehnseitiger? Dann wäre der Knickwinkel zwischen den Polygonflächen des Mantels mit 36° kleiner als der Default-Knickwinkel. Das sichtbare Ergebnis wäre ein Kegelstumpf, dessen Mantel den Eindruck hinterläßt, auch tatsächlich rund zu sein, da alle Polygone des Mantels auf Grund der erzeugten Normals als eine einzige, sich krümmende Fläche dargestellt werden, man könnte die Kanten also nicht mehr klar als solche erkennen! Und so lange man nicht die immer noch deutlich sichtbar Zehneckige Grundfläche sieht, wäre diese Illusion auch weitgehend perfekt.

4925 Damit bietet sich also eine sehr gute Möglichkeit, Objekte rund und glatt erscheinen zu lassen, ohne dafür sehr viele Polygone verwenden zu müssen. Gäbe es die Möglichkeit der Illusion glatter Flächen mittels dieses NormalGenerators nicht, wäre der einzige Ausweg nämlich der, sehr viele Polygone zu verwenden, was sich deutlich im Ressourcenverbrauch und damit in einbrechenden Frameraten niederschlagen würde.

4930 Doch zurück zu dem Gedankenexperiment mit dem überraschend erzeugten Kegelstumpf. Was wäre, wenn dieser nun trotzdem als kantiger, zehnsseitiger Pyramidenstumpf dargestellt werden soll? Nichts einfacher als das, der creaseAngle müßte nur auf einen Wert kleiner 36° gesetzt werden und die einzelnen Flächen der Polygone des Mantels wären wieder gut zu erkennen (dabei ist nicht zu vergessen: der creaseAngle erwartet den Winkel in Radians, nicht in Grad, es ist gegebenenfalls also mit Math.toRadians() umzurechnen).

Ob einzelne Seiten oder Flächen eines 3D-Modelles glatt (smooth) oder eben doch eher kantig (faceted) dargestellt werden sollen, wird innerhalb der meisten Dateien der verschiedenen 3D-Datenformate im übrigen angegeben. Abhängig von dieser Information

4940 genügt es in der Regel, den NormalGenerator entweder mit seinem Default-CreaseAngle zu verwenden oder aber mit einem Knickwinkel von 0°.

### 8.4.9.3 Stripifier

4945 Ein weiteres im Beispielprogramm verwendete Utility ist der Stripifier. Dieser hat – wie oben bereits erwähnt – eigentlich nichts mit der Erzeugung von Normals zu tun. Da er allerdings ebenfalls ein GeometryInfo-Objekt verwendet, liegt es nahe, diesen hier gleich mit anzuwenden und zu besprechen.

4950 Wie bereits bei den Geometrie-Klassen IndexedTriangleStripArray und TriangleStripArray gesehen, ist eine optimierte Anordnung der Polygone innerhalb von Geometriedaten möglich und zweckmäßig. Speziell die Anordnung als Strip spart zum einen Speicherplatz und kommt zum anderen der 3D-Hardware entgegen, da es dieser durch die optimierte Anordnung das Zeichnen erleichtert.

4955

Genau so eine optimierte Strip-Anordnung kann mit dem Stripifier erzeugt werden. Dieser versucht innerhalb der übergebenen Daten Möglichkeiten zu finden, die Dreiecke als Strip anzuordnen. Wurden Geometriedaten übergeben, die Quads enthalten, so kann der Stripifier ebenfalls angewendet werden, hier zerlegt er diese Quads jedoch zwangsläufig erst einmal in Dreiecke.

4960

Die Verwendung des Stripifiers ist denkbar einfach.

```
Stripifier()
```

4965 Dieser Konstruktor erzeugt ein Stripifier-Objekt, das keine statistischen Daten sammelt.

```
Stripifier(int stats)
```

4970 Für den einzigen Parameter dieses Konstruktors gibt es nur die Möglichkeit, eine 0 zu übergeben oder aber die ebenfalls in der Klasse Stripifier definierte Konstante **COLLECT\_STATS**. Damit ist es möglich, statistische Informationen über die Arbeit des Stripifiers zu sammeln und anschließend auszuwerten.

4975 Die in dieser Klasse vorhandenen Methoden sind recht übersichtlich, was aber nicht heißen soll, dass sie damit auch unwichtig wären:

```
void stripify(GeometryInfo geometryinfo)
```

4980 Diese Methode versucht, die Geometriedaten in `geometryinfo` zu optimieren und in Form von Strips anzuordnen. Wie in Zeile 23 des Beispielprogrammes zu sehen ist, wird damit die eigentliche Hauptarbeit getan.

```
StripifierStats getStripifierStats()
```

4985 Wurde beim Anlegen des Objektes mit **COLLECT\_STATS** dafür gesorgt, dass Statistikdaten gesammelt werden, so können diese Daten mit dieser Methode geholt werden. Das Ergebnis wird dabei in Form eines StripifierStats-Objektes zurückgeliefert.

#### 8.4.9.3.1 StripifierStats

4990 Da der Konstruktor der Klasse StripifierStats `protected` ist, ist es nicht möglich, diesen von außerhalb der Package `com.sun.j3d.utils.geometry` zu verwenden. Das wäre auch nicht wirklich sinnvoll, da nur der Stripifier diesen Konstruktor benötigt. Viel wichtiger und demzufolge auch von außen aus aufrufbar sind die verschiedenen Methoden dieser Klasse, die unter anderem die gesammelten statistischen Daten zurückliefern:

```
4995 void clearData()
```

Diese Methode setzt die Werte aller gesammelten Daten auf 0 zurück.

```
double getAvgNumVertsPerTri()
```

5000 Es wird die durchschnittliche Anzahl von Vertices zurückgeliefert, die zu einem Dreieck gehören. Diese Zahl ist um so höher, um so weniger Strips erzeugt werden konnten, weil dann mehr einzelne Dreiecke bzw. ineffizientere weil kurze Strips in den Geometriedaten erhalten bleiben.

```
double getAvgStripLength()
```

5005 Mit dieser Methode ist es möglich, die durchschnittliche Länge der Strips zu ermitteln.

```
int getMaxStripLength()
```

Hiermit wird die Länge des längsten erzeugten Strips zurückgeliefert.

5010

```
int getMinStripLength()
```

Im Gegensatz dazu liefert diese Methode die Länge des kürzesten erzeugten Strips zurück

```
5015 int getNumOrigTris()
```

Diese Methode liefert die Anzahl ursprünglich vorhandener Dreiecke zurück. Wurden Geometriedaten mit Quads verwendet, so ist dieser Wert durch zwei zu dividieren, da diese bekanntlich in Dreiecke zerlegt werden und zwei Dreiecke immer ein Quad ergeben (bzw. umgekehrt).

5020

```
int getNumOrigVerts()
```

Es wird die Anzahl ursprünglich vorhandener Vertices zurückgeliefert, so fern indizierte Geometriedaten zum Einsatz kamen.

5025 `int getNumStrips()`

Hiermit ist es möglich, die Gesamtanzahl erzeugter Strips in Erfahrung zu bringen.

`int getNumVerts()`

5030 Im Gegensatz zu `getNumOrigVerts()` liefert diese Methode die Anzahl an Vertices zurück, die nach der Bearbeitung der Geometriedaten in diesen enthalten sind.

`int[] getStripLengthCounts()`

5035 Diese Methode gibt detaillierte Auskunft über die Länge jedes erzeugten Strips. Es wird ein `int`-Array zurückgeliefert, das die Längeninformation für jeden Strip enthält. Die Länge dieses Arrays ist also identisch mit der Anzahl erzeugter Strips.

`long getTotalTime()`

5040 Um in Erfahrung zu bringen, wie lange die Bearbeitung der Daten insgesamt gedauert hat, sollte diese Methode verwendet werden. Der zurückgegebene Zahlenwert gibt die Zeitdauer in Millisekunden an.

`int getTotalTris()`

5045 Wie der Name es bereits verrät, liefert diese Methode die Gesamtzahl der in den Geometriedaten vorkommenden Dreiecke zurück. Wurden Quads verwendet, so ist auch hier wieder mit Hilfe des Faktors zwei umzurechnen.

## 9 Picking

5050 Ein weiteres wichtiges Gebiet innerhalb der Programmierung von dreidimensionalen  
virtuellen Welten befaßt sich damit, 3D-Objekte in einer Szene aufzufinden. Sei es durch  
den Eingriff von außen, weil z.B. ein User mit der Maus auf ein 3D-Objekt klickt, das  
anschließend selektiert, bewegt oder in anderer Weise manipuliert werden soll, oder aber,  
5055 weil es für eine Kollisionsvermeidung von bewegten Objekten innerhalb einer animierten  
Szene erforderlich ist. In beiden Fällen ist es nötig, dass in einer bestimmten Richtung  
(vorgegeben beispielsweise durch den Klick des Benutzers in das Fenster mit der 3D-  
Darstellung oder vorgegeben durch die Bewegungsrichtung eines animierten Objektes)  
überprüft werden muß, ob dort 3D-Objekte zu finden sind. Das kann bedeuten, dass das  
erste gefundene 3D-Objekt zurückgeliefert werden soll, weil höchstwahrscheinlich dieses  
5060 angeklickt wurde, oder das alle gefundenen Objekte innerhalb eines bestimmten,  
eventuell auch eingeschränkten Aktionsradius gefunden werden sollen. Für eine  
Kollisionsvermeidung wäre es beispielsweise schlichtweg unnötig und damit auch eine  
Ressourcenverschwendung, wenn immer alle 3D-Objekte ermittelt werden würden, auch  
wenn diese so weit entfernt sind, dass eine Kollision mit ihnen schon auf Grund dieser  
großen Entfernung gar nicht in Frage kommt.

5065

Da es in der Package `com.sun.j3d.util.picking` bereits passende Utilities für das  
Picking durch Mausclicks befinden, soll mit diesem dadurch deutlich einfacheren Weg  
begonnen werden.

### 5070 9.1 Dreidimensionaler Text

Gleichzeitig soll mit `Text3D` eine weitere Spezialklasse vorgestellt werden. Zu diesem  
Zweck muß das Beispielprogramm beginnend mit der Methode `createSceneGraph()`  
wieder etwas umgebaut werden, da hier das `Text3D`-Objekt erzeugt werden soll:

5075

```
(1) BranchGroup createSceneGraph()
(2) {
(3) BranchGroup RootBG=new BranchGroup();
(4) Appearance TextAppearance=new Appearance();
5080 (5) AmbientLight ALgt=new AmbientLight(new Color3f(1f,1f,1f));
(6) DirectionalLight DLgt=new DirectionalLight(new Color3f(1f,1f,1f),new
 Vector3f(0.5f,0.5f,-1f));
(7) BoundingSphere BigBounds=new BoundingSphere(new Point3d(),100000);
(8) TransformGroup TextTG=new TransformGroup();
5085 (9) Transform3D TextT3D=new Transform3D();
(10) Text3D T3D;
(11) Font3D TextFont;
(12) Shape3D TextShape;
(13) FontExtrusion FontExt;
```

```

5090 (14) int[] x=new int[3],y=new int[3];
 (15)
 (16) ALgt.setInfluencingBounds(BigBounds);
 (17) DLgt.setInfluencingBounds(BigBounds);
 (18) TextAppearance.setMaterial(new Material(new Color3f(0f,0f,1f),new
5095 Color3f(0f,0f,0f),new Color3f(1f,0f,0f),new Color3f(1f,1f,1f),100f));
 (19)
 (20) TextT3D.setScale(0.02);
 (21) TextTG.setTransform(TextT3D);
 (22) y[0]=2; y[1]=1; y[2]=0;
5100 (23) x[0]=0; x[1]=4; x[2]=5;
 (24) FontExt=new FontExtrusion(new Polygon(x,y,3),1);
 (25) TextFont=new Font3D(new Font("sans serif",0,24),0.01,FontExt);
 (26) T3D=new Text3D(TextFont,"Java 3D",new Point3f(0f,4f,-
 15f),Text3D.ALIGN_CENTER,Text3D.PATH_RIGHT);
5105 (27) T3D.setCapability(Text3D.ALLOW_INTERSECT);
 (28) TextShape=new Shape3D(T3D,TextAppearance);
 (29) TextTG.addChild(TextShape);
 (30)
 (31) RootBG.addChild(TextTG);
5110 (32) RootBG.addChild(ALgt);
 (33) RootBG.addChild(DLgt);
 (34) RootBG.compile();
 (35) return RootBG;
 (36) }

```

5115

In den Zeilen 10 bis 14 finden sich dieses mal gleich mehrere Deklarationen. Dieser Aufwand ist nötig, da bei der Erzeugung eines dreidimensionalen Textes aus einem sonst immer nur zweidimensionalen Font doch recht viele Informationen hinzugefügt werden müssen, die bis dato schlichtweg nicht vorhanden sind. Die ersten Schritte in diesem Zusammenhang finden sich in den Zeilen 20 und 21 sowie anschließend auch in Zeile 29. Es wird eine TransformGroup/Transform3D Kombination erzeugt, die das Text3D-Objekt beeinflussen soll. Hier geht es um eine Verkleinerung des Objektes, dass sonst in der Szene wirklich sehr groß erscheinen würde. Das passiert mit dem Aufruf von `setScale()` mit einem Skalierungsfaktor von 0.02 in Zeile 20.

5125

### 9.1.1 FontExtrusion

In den Zeilen 22 bis 24 wird ein FontExtrusion Objekt erzeugt. Wie der Name bereits andeutet, ist dieses ein Hilfs- oder Definitionsmittel, das es ermöglicht, den Font in die dritte Dimension zu extrudieren, also in die „Tiefe“ zu ziehen. Hier werden wieder alle wesentlichen Daten mit dem Konstruktor übergeben:

5130

```
FontExtrusion(java.awt.Shape extrusionShape, double
tessellationTolerance)
```

5135

Der erste benötigte Parameter beschreibt, wie der Text in der Tiefe geformt sein soll. Das passiert im Beispielprogramm in den Zeilen 22 bis 24 mit Hilfe einer `java.awt`-Klasse, dem Polygon das sich von der eigentlich geforderten AWT-Klasse `Shape` ableitet. Diesem Polygon werden drei xy-Koordinatenpaare übergeben, die die Form der Buchstaben in der neu zu gewinnenden Tiefe beschreiben. Der zweite Parameter wird bei der Umwandlung des Textes in Polygone benötigt. Diese `tessellationTolerance` beschreibt, wie stark Abweichungen der Vertices des späteren dreidimensionalen Objektes bei dieser Extrusion sein dürfen. Um so kleiner dieser Wert ist, um so größer wird die Genauigkeit und Präzision der erzeugten Objekte in Z-Richtung, um so höher wird aber auf Grund der deutlich größeren Menge an dafür aufgewendeten Polygonen auch wieder der Ressourcenverbrauch.

5140

5145

Die zur Verfügung gestellten Methoden beschränken sich hier auf set- und get-Funktionalitäten für die `extrusionShape` sowie eine get-Funktion für die `tessellationTolerance` (die also nur beim Erzeugen des Objektes beeinflusst werden kann):

5150

```
java.awt.Shape getExtrusionShape()
void setExtrusionShape(java.awt.Shape extrusionShape)
```

5155

Mit diesen Methoden ist es möglich, das für das Extrudieren aktuell verwendete `Shape` zu ermitteln oder aber ein neues festzulegen.

```
double getTessellationTolerance()
```

5160

Der Rückgabewert dieser Methode ist die aktuelle `tessellationTolerance` für die Extrusion des Fonts.

Auch die Ableitung der Klasse `FontExtrusion` ist denkbar simpel:

```
java.lang.Object
javax.media.j3d.FontExtrusion
```

5165

## 9.1.2 Font3D

5170

In Zeile 25 wird das eben erzeugte `ExtrusionShape`-Objekt verwendet, um ein `Font3D`-Objekt zu konstruieren. Das macht durchaus Sinn, schließlich muß ja ein bestimmter (wählbarer) Zeichensatz mit Hilfe der Informationen aus dem `ExtrusionShape` in die dritte Dimension erweitert werden. Auch in Zeile 25 spielt sich wieder alles wesentliche im Zusammenhang mit der Klasse `Font3D` bei der Erzeugung des zugehörigen Objektes ab:



5175 `Font3D(java.awt.Font font, double tessellationTolerance,  
FontExtrusion extrudePath)`

Der erste zu übergebende Parameter stammt auch hier aus dem Paket `java.awt`. Es wird der Zeichensatz festgelegt, der bei der Erzeugung des dreidimensionalen Fonts für die ersten beiden Dimensionen als Vorlage dienen soll. Der zweite Parameter ist bereits vom vorhergehend erzeugten `FontExtrusion`-Objekt bekannt, da er dort in ähnlicher Weise Verwendung findet. Auch hier legt er wieder fest, wie groß die Abweichungen von den Vertices der zu erzeugenden Polygone von der Vorlage sein dürfen. Bei der Klasse `Font3D` bezieht sich `tessellationTolerance` allerdings nur auf die erlaubte Abweichung in Breite und Höhe, da für die Tiefe schließlich bereits ein eigener Toleranzwert im `FontExtrusion`-Objekt definiert wurde. Auch hier gilt wieder: Um so kleiner dieser Wert ist, um so exakter und glatter wird der erzeugte 3D-Zeichensatz. Zusätzlich spielt hier allerdings auch der erzeugt `java.awt.Font` eine Rolle. Einer seiner Eigenschaften ist seine Größe. Und um so größer dieser Font ist, um so genauer wird auch er schon. Da heißt, auch die Größe des zweidimensionalen Fonts hat einen groben Einfluß auf die Exaktheit der späteren Darstellung. Als dritter und letzter Parameter wird schließlich das `FontExtrusion`-Objekt erwartet.

5195 Die Klasse `Font3D` bietet nach ihrer Instantiierung keinerlei spezifische Möglichkeiten mehr, sie zu verändern. Es werden lediglich Methoden zur Verfügung gestellt, die unter anderem auch die mit dem Konstruktor spezifizierten Werte zurückliefern:

```
void getBoundingBox(int glyphCode, BoundingBox bounds)
```

5200 Der in das zu übergebende `BoundingBox`-Objekt `bounds` kopierte Rückgabewert dieser Methode ist die Begrenzung eines bestimmten Buchstabens, der mittels `glyphCode` festgelegt wird. Diese Methode erlaubt es also, die Bounds und damit die maximalen Ausdehnungen bestimmter Buchstaben zu ermitteln.

```
java.awt.Font getFont()
```

5205 Der Rückgabewert dieser Methode ist das bei der Erzeugung des zugehörigen `Font3D`-Objektes übergebene `Font`-Objekt.

```
void getFontExtrusion(FontExtrusion extrudePath)
```

5210 Diese Methode liefert das verwendete `FontExtrusion` Objekt zurück, in dem dessen Werte in das als Parameter übergebene Objekt `extrudePath` kopiert werden.

```
double getTessellationTolerance()
```

5215 Mit der `tessellationTolerance` wird die letzte noch verbleibende `Font3D`-Eigenschaft von dieser Methode zurückgeliefert.

Die Klasse `Font3D` bringt keine eigenen Capability-konstanten mit, was bei der Verwandtschaft mit der Klasse `NodeComponent` hätte vermutet werden können:

```
java.lang.Object
5220 javax.media.j3d.SceneGraphObject
 javax.media.j3d.NodeComponent
 javax.media.j3d.Font3D
```

### 9.1.3 Text3D

5225

In den Zeilen 26 bis 28 finden sich nun die letzten Schritte, die nötig sind, um den dreidimensionalen Text in der Szene unterzubringen. Diese bestehen zum einen aus der Erzeugung der eigentlich benötigten Geometriedaten in Form eines Text3D-Objektes. Zum anderen müssen diese Geometriedaten anschließend zusammen mit einem Shape3D-Objekt dem Universum zugänglich gemacht werden. Auch bei der Klasse Text3D wird wieder alles Nötige mit dem Konstruktor erledigt:

5230

```
Text3D(Font3D font3D, java.lang.String string, Point3f position, int
alignment, int path)
```

5235

Als erster Parameter wird das einige Schritte zuvor erzeugte Font3D-Objekt übergeben, das alle nötigen Informationen enthält, also wie der Zeichensatz aussehen soll und wie er entlang der Z-Achse „tiefgezogen“ werden soll. Der Parameter `string` erwartet den Textstring, der letztendlich dargestellt werden soll.

5240

Mit dem Point3f-Objekt `position` wird festgelegt, an welcher Position der Text dargestellt werden soll.

Wo sich dieser Punkt dann relativ zum Textobjekt selber befindet, wird mittels `alignment` festgelegt: **ALIGN\_CENTER** zentriert den Text an der spezifizierten Position,

5245

**ALIGN\_FIRST** sorgt dafür, dass sich der erste Buchstabe dort befindet und **ALIGN\_LAST** wiederum legt fest, dass sich der letzte Buchstabe an den Koordinaten von `position` befindet.

Der letzte noch verbleibende Parameter, `path`, legt schließlich fest, wie der Text dargestellt werden soll. **PATH\_RIGHT** entspricht dabei der im mitteleuropäischen Raum üblichen Schreibweise von links nach rechts während **PATH\_LEFT** das Gegenstück dazu darstellt, die Buchstaben aus unserer Sichtweise also verkehrt herum anordnet.

5250

**PATH\_DOWN** wiederum spezifiziert eine Schreibweise von oben nach unten, wie es z.B. aus dem Chinesischen bekannt ist oder von vertikal angeordneten Beschriftungen an Häusern. Das Gegenstück dazu ist **PATH\_UP**, was dafür sorgt, dass die Buchstaben von unten nach oben angeordnet werden.

5255

Die Klasse Text3D bringt einige beachtenswerte Methoden mit, denen sich auch wieder verschiedene, ebenfalls Klassen-spezifische Capability-Konstanten zuordnen lassen:

```
int getAlignment()
5260 void setAlignment(int alignment)
```

Der aktuelle Alignment-Mode kann mit Hilfe dieser Methoden ermittelt oder neu gesetzt werden. Wie gerade beschrieben kommen dazu die Konstanten **ALIGN\_CENTER**, **ALIGN\_FIRST** und **ALIGN\_LAST** zum Einsatz.

5265 `void getBoundingBox(BoundingBox bounds)`

Mit dieser Methode ist es möglich, die BoundingBox zu ermitteln, die diesen Textstring umfaßt.

`float getCharacterSpacing()`

5270 `void setCharacterSpacing(float characterSpacing)`

Für Text3D-Objekte ist es möglich, einen zum verwendeten Font-Objekt zusätzlichen Abstand zwischen den einzelnen Buchstaben zu verwenden. Dieser Wert wird dabei zum bereits vorhandenen Abstand hinzugezählt. Der voreingestellte Abstand ist 0.0 während ein Wert von 1.0 der größten Breite der einzelnen Buchstaben entspricht. Diese beiden Methoden liefern den aktuellen Wert für das Character-spacing zurück bzw. erlauben es, einen neuen Wert festzulegen.

`Font3D getFont3D()`

`void setFont3D(Font3D font3d)`

5280 Das für diesen Text3D verwendete Font3D-Objekt wird von diesen Methoden zurückgeliefert bzw. es kann ein neues Font3D-Objekt zur Verwendung übergeben werden.

`int getPath()`

5285 `void setPath(int path)`

path legt bei der Klasse Text3D fest, wie der Text ausgerichtet sein soll und wie er verlaufen soll. Der aktuell verwendete Wert wird von der Methode getPath() zurückgeliefert während es mit setPath() möglich ist, eine neue Ausrichtung festzulegen. Dabei kommen die oben bereits erwähnten Konstanten **PATH\_RIGHT**, **PATH\_LEFT**, **PATH\_UP** und **PATH\_DOWN** zum Einsatz.

5290

`void getPosition(Point3f position)`

`void setPosition(Point3f position)`

5295 Diese Methoden kopieren die aktuelle Position des Text3D-Objektes innerhalb der Szene in bzw. aus dem als Parameter übergebenen Point3f-Objekt und ermöglichen es so, die aktuellen Koordinaten zu ermitteln oder aber eine neue Position festzulegen.

`java.lang.String getString()`

`void setString(java.lang.String string)`

5300 Bei den von diesen Methoden verwendeten Strings handelt es sich um den Text, der von diesem Text3D-Objekt in der Szene dargestellt wird. Sie liefern den aktuell verwendeten Textstring zurück beziehungsweise ermöglichen es, einen neuen String zu

definieren.

- 5305 Die Ableitungsverhältnisse der Klasse Text3D erklären das, was in Zeile 28 des Beispielprogrammes geschieht:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
5310 javax.media.j3d.NodeComponent
 javax.media.j3d.Geometry
 javax.media.j3d.Text3D
```

- 5315 Da diese sich nicht wie vielleicht zu erwarten von Primitive oder Shape3D ableitet, sondern von Geometry, ist es erforderlich, das erzeugte Text3D-Objekt T3D zusammen mit dem Appearance-Objekt TextAppearance einem Shape3D-Objekt zuzuordnen, um dieses dann dem SceneGraphen hinzuzufügen.

- 5320 Von der Klasse Geometry erbt Text3D auch die Capability **ALLOW\_INTERSECT**, die in Zeile 27 aktiviert wird. Das ist für das in Kürze folgende Picking notwendig, da diese Eigenschaft benötigt wird, um festzustellen, ob ein Picking-Ereignis wirklich die Geometrie des 3D-Objektes schneidet oder an ihm vorbeigeht. Damit soll nun dieser abschweifende Ausflug in die Welt dreidimensionaler Schriftzüge beendet werden um zum eigentlichen Sinn dieses Kapitels zurückzukehren.

5325

### 9.1.4 PickCanvas

- Am Beispielprogramm sind abseits des neu vorgestellten Objekttyps nämlich noch einige Änderungen mehr vorzunehmen. So müssen die Objekte RootBG sowie c (für das Canvas3D-Objekt) global verfügbar sein. Der Grund dafür findet sich in Kürze in der verwendeten Picking-Methodik. Des weiteren ist es nötig, den Code für einen MouseListener einzufügen, der auch dem Canvas3D c mit addMouseListener() hinzugefügt wird. Dieser ist logischerweise notwendig, da schließlich Mausklicks in den Canvas3D hinein registriert und ausgewertet werden sollen. Aus dem MouseListener
- 5335 heraus wird bei jedem mouseClicked() eine neu zu erstellende Methode aufgerufen:

```
(1)void handleObjectSelection(MouseEvent mousee)
(2) {
(3) Node PickedObject;
5340 (4) PickCanvas MyPick=new PickCanvas(c,RootBG);
(5) PickResult[] MyPickResult;
(6) int resCtr=0;
(7)
```

```

(8) MyPick.setTolerance(0f);
5345 (9) MyPick.setMode(PickTool.GEOMETRY);
(10) MyPick.setShapeLocation(mousee);
(11) MyPickResult=MyPick.pickAllSorted();
(12) if (MyPickResult==null)
(13) {
5350 (14) System.out.println("nichts");
(15) return;
(16) }
(17) while (resCtr<MyPickResult.length)
(18) {
5355 (19) PickedObject=MyPickResult[resCtr].getObject();
(20) System.out.println(PickedObject);
(21) resCtr++;
(22) }
(23) }

```

5360

In Zeile 4 wird eine Variable `MyPick` deklariert und das zugehörige `PickCanvas`-Objekt erzeugt. Dieses soll dazu dienen, bei einem Klick in den `Canvas3D` die zweidimensionalen, auf die Bildschirmauflösung und den `Canvas3D` bezogenen Mauskoordinaten in eine dreidimensionale Position und einen Richtungsvektor innerhalb der virtuellen 3D-Welt umzurechnen und die 3D-Objekte zu ermitteln, die sich in dieser Richtung befinden. Dazu benötigt die Klasse `PickCanvas` zwangsläufig das `Canvas3D`-Objekt (weswegen dieses jetzt global verfügbar sein muß), in das mit der Maus hineingeklickt wird sowie eine `BranchGroup`, unterhalb der nach Objekten gesucht werden soll, die durch diesen Klick „getroffen“ worden sein könnten.

5370

Doch zuvor sind noch einige weitere Werte für den `PickCanvas` festzulegen. Das geschieht in den Zeilen 8 und 9. `setTolerance()` legt einen Toleranzbereich für den Mausklick fest. Hier wurde 0 angegeben, was bedeutet, das Objekte wirklich exakt getroffen werden müssen, damit der `PickCanvas` diese auch registriert. Viel interessanter ist allerdings der Aufruf der Methode `setMode()`. Hier wird für den Modus die Konstante **GEOMETRY** übergeben, die festlegt, dass tatsächlich die Geometriedaten herangezogen werden sollen, um festzustellen, ob ein Objekt getroffen wurde. Das heißt, dass beispielsweise ein Klick in die Öffnung des „D“ im Schriftzug „Java 3D“ nicht als Treffer auf das Objekt gewertet wird. Anders, wenn der Modus **BOUNDS** verwendet wird, bei der die Geometriedaten nicht mehr beachtet werden, sondern nur noch die zu diesem 3D-Objekt gehörenden Bounds. Da hier die Öffnung des „D“ innerhalb dieser Bounds liegt, wird ein Klick an diese Stelle jetzt als Treffer gewertet. Diese deutlich ungenauere Methode hat den Vorteil, das sie auch wesentlich schneller und ressourcenschonender von statten geht.

5385

In Zeile 10 schließlich wird mit der Methode `setShapeLocation()` der vom `MouseListener` gelieferte `MouseEvent` an das `PickCanvas`-Objekt übergeben, so dass dieses endlich erfährt, wo hin eigentlich geklickt wurde. Anschließend ist es möglich, das

Ergebnis dieses „Picks“ zu ermitteln. Dazu wird im Beispiel die Methode  
5390 `pickAllSorted()` verwendet, die – wie der Name bereits sagt – die Objekte, für die  
dieser Klick ein Treffer war, in sortierter Reihenfolge zurückliefert. Das heißt, das am  
nächsten liegende Objekt findet sich im zurückgelieferten Array vom Typ `PickResult` ganz  
vorne, also an Indexposition 0. Wie mit den zurückgegebenen `PickResult`-Objekten zu  
5395 verfahren ist – wenn denn überhaupt ein Objekt getroffen wurde – ist innerhalb der  
Schleife zu sehen, die alle Elemente des Arrays abarbeitet. Mittels der Methode  
`getObject()` wird eine Referenz auf das 3D-Objekt aus dem `PickResult`-Objekt geholt,  
das durch den Mausklick getroffen wurde. In der recht einfachen Szene des  
Beispielprogrammes kann das natürlich nur das zum 3D-Text gehörende `Shape3D`-Objekt  
sein.

5400

Das beschriebene und gewünschte Verhalten läßt sich leicht überprüfen, wenn das  
Beispielprogramm compiliert und gestartet wird. Ein Klick auf den 3D-Text liefert den  
Namen des `Shape3D`-Textes zurück, geht der Klick daneben, passiert das  
dementsprechend nicht.

5405

Wesentliches Element zur Ermittlung des angeklickten 3D-Objektes ist hier – wie gesehen  
– die Klasse `PickCanvas`. Der verwendete Konstruktor wurde dabei im wesentlichen  
eigentlich bereits beschrieben:

5410 `PickCanvas(Canvas3D c, BranchGroup b)`

Es wird zum einen das `Canvas3D`-Objekt `c` benötigt, innerhalb dessen geklickt wird bzw.  
wurde sowie eine `BranchGroup` `b`, innerhalb deren Children nach dem angeklickten Objekt  
gesucht werden soll. Soll die komplette Szene durchsucht werden und nicht nur ein Teil-  
5415 `SceneGraph`, so sollte hier die gleiche `BranchGroup` angegeben werden, die auch schon  
zusammen mit der Methode `addBranchGraph()` verwendet wurde, um dem Universum  
alle Elemente hinzuzufügen.

Bei den Methoden der Klasse `PickCanvas` findet sich dann folgendes:

5420

`Canvas3D getCanvas()`

Diese Methode liefert den mit dem Konstruktor festgelegten `Canvas3D` zurück.

`float getTolerance()`

5425 `void setTolerance(float t)`

Diese Methoden liefern jeweils den aktuellen Toleranzwert für das Picking zurück  
bzw. setzen einen neuen Wert. Dieser Wert legt fest, wie exakt ein 3D-Objekt in der  
Szene getroffen werden muß, um dennoch gefunden zu werden. Klassen-intern wird  
dieser Wert verwendet, um ein `PickCylinder`-Objekt zu erzeugen, mit dem dann ermittelt  
5430 wird, ob es irgend welche Schnittpunkte mit 3D-Objekten gibt.

```
void setShapeLocation(int xpos,int ypos)
```

```
void setShapeLocation(int xpos,int ypos)
```

5435 Der Name dieser Methode mag verwirren, jedoch ist er vollkommen korrekt, wenn man weiß, dass das Picking intern mit Hilfe eines Objektes vom Typ PickShape (von der sich auch der bereits erwähnte PickCylinder ableitet) ausgeführt wird. Diese Methode übergibt nun die aktuellen Mauskoordinaten an das PickCanvas-Objekt, so dass dieses das benötigte PickShape-Objekt an der zugehörigen Stelle in der dreidimensionalen Welt setzen kann. Der Aufruf dieser Methode ist Voraussetzung dafür, dass es anschließend

5440 möglich ist, die betroffenen Objekte zu ermitteln. Da sich die dafür notwendigen Methoden in der Basisklasse PickTool befinden, sollen sie auch erst im folgenden Abschnitt besprochen werden.

Die Verwandtschaft mit dem PickTool zeigt sich auch hier:

5445

```
java.lang.Object
```

```
com.sun.j3d.utils.picking.PickTool
```

```
com.sun.j3d.utils.picking.PickCanvas
```

## 5450 9.1.5 PickTool

Der benötigte Konstruktor der Klasse PickTool sehen ein wenig anders aus als die im vorangegangenen Abschnitt:

5455 `PickTool(BranchGroup b)`

Es wird als Parameter lediglich die BranchGroup `b` erwartet, innerhalb der nach dem gewünschten Objekt gesucht werden soll. Ein Bezug auf ein Canvas3D oder ähnliches ist nicht zu finden, da diese Basisklasse als grundlegendes Element sich schließlich nicht auf

5460 einen speziellen Pick-Mechanismus festlegen sollte.

Bei den Methoden findet sich einiges interessantes:

```
BranchGroup getBranchGroup()
```

5465 Diese Methode liefert die mit dem Konstruktor übergebene BranchGroup zurück, innerhalb der das PickTool-Objekt nach „getroffenen 3D-Objekten suchen würde.“

```
int getMode()
```

```
void setMode(int mode)
```

5470 Diese Methoden liefern jeweils den aktuellen Modus zurück bzw. setzen ihn neu. Dabei kommen auch die bereits im vorhergehenden Abschnitt beschriebenen Modi

**GEOMETRY** und **BOUNDS** in Frage, die festlegen, ob sich das Picking nur auf die Bounds der 3D-Objekte oder aber wesentlich exakter auf deren echte Geometrie beziehen soll. Es sei noch erwähnt, dass für die Klasse `PickTool` (und damit natürlich auch für alle abgeleiteten Klassen) auch ein Modus **GEOMETRY\_INTERSECT\_INFO** existiert, der Momentan allerdings nicht weiter von Wichtigkeit ist.

```
PickShape getPickShape()
```

Diese Methode liefert das aktuell verwendete `PickShape`-Objekt zurück. Dieses wird verwendet, um eventuell vorhandene Schnittpunkte mit 3D-Objekten in der Szene zu ermitteln. Das heißt, damit wird die eigentliche Arbeit beim Picking erledigt. Gesetzt werden benötigte `PickShape`-Objekte mit einer der folgenden Methoden:

```
void setShape(PickShape ps, Point3d startPt)
```

Es wird ein benutzerdefiniertes `PickShape`-Objekt gesetzt, das zur Ermittlung von möglichen Schnittpunkten mit anderen 3D-Objekten der Szene verwendet wird. Hier bietet Java 3D mit z.B. `PickRay`, `PickSegment` und anderen mehrere Möglichkeiten, die in den folgenden Abschnitten näher beschrieben werden.

```
void setShapeConeRay(Point3d start, Vector3d dir, double angle)
```

Diese Methode definiert ein `PickShape`-Objekt, das hier von der Klasse `PickTool` selbst erzeugt wird. Bei diesem handelt es sich um einen Kegel, dessen Spitze (und damit Ausgangspunkt) durch den Parameter `start` festgelegt wird. Der `Vector3d dir` spezifiziert, in welche Richtung „gepickt“ werden soll, während `angle` den Öffnungswinkel dieses Kegels festlegt. Alle 3D-Objekte der Szene, die zu der bei der Konstruktion des `PickTool`-Objektes übergebenen `BranchGroup` gehören und sich räumlich mit dem so definierten, bis ins unendliche gehenden Kegel schneiden, können dadurch ermittelt werden.

In Vorgriff auf die detaillierte Beschreibung der `PickShape`-Klassen weiter unten sei erwähnt, dass alternativ dazu auch mittels `setShape()` ein `PickConeRay`-Objekt hätte übergeben werden können.

```
void setShapeConeSegment(Point3d start, Point3d end, double angle)
```

Im Gegensatz zur vorhergehenden Methode legt diese für das Picking einen endlichen Kegel fest. Dessen Start- und Endpunkt sowie seine Richtung wird mit Hilfe der Parameter `start` und `end` festgelegt, während `angle` wieder den Öffnungswinkel den Kegels angibt. Das `PickTool`-intern erzeugte `PickShape`-Objekt ist hier dementsprechend ein (ebenfalls weiter unten detailliert beschriebenes) `PickConeSegment`-Objekt.

```
void setShapeCylinderRay(Point3d start, Vector3d dir, double radius)
```

Diese Methode verwendet intern wieder ein unendliches `PickShape`-Objekt, den `PickCylinderRay`. Damit ist es möglich, ab einer Startposition `start` in einer bestimmten Richtung `dir` innerhalb eines festgelegten `radius` 3D-Objekte innerhalb des aktuell untersuchten `SceneGraphen` zu ermitteln. Die (gedachte) Form des Picking-Strahls ist dieses mal also ein ins unendliche verlaufender Zylinder.



```
void setShapeCylinderSegment(Point3d start,Point3d end,double
radius)
```

5520 Diese Methode stellt wiederum das endliche Gegenstück zur Vorangegangenen dar. Diesmal wird innerhalb eines zylindrischen Raumes, der sich von der Position `start` bis zur Position `end` erstreckt und den mit dem dritten Parameter übergebenen `radius` hat, nach betroffenen 3D-Objekten gesucht. Der Vollständigkeit halber sei wieder erwähnt, dass intern ein Objekt vom Typ `PickCylinderSegment` verwendet wird, welches wie alle anderen von `PickShape` angeleiteten Klassen in Kürze noch näher beschrieben wird.

5525

```
void setShapeRay(Point3d start,Vector3d dir)
```

5530 Diese Methode sucht innerhalb eines eindimensionalen Raumbereiches nach 3D-Objekten. Es handelt sich bei diesem Raumbereich um einen Strahl, der von der Position `start` beginnend in Richtung `dir` ins Unendliche verläuft und es ermöglicht, alle 3D-Objekte zu ermitteln, die diesen schneiden. Dieses Verhalten entspricht dem der Klasse `PickRay`.

```
void setShapeSegment(Point3d start,Point3d end)
```

5535 Die letzte der Methoden, die näher festlegen, in welcher Richtung und innerhalb welchen Raumes nach Objekten gesucht werden soll, ist wiederum das endliche Gegenstück zur vorhergehenden. Mittels der Positionen `start` und `end` wird festgelegt, entlang welchen endlichen Strahls nach 3D-Objekten gesucht werden soll. Das Pendant dazu ist die Klasse `PickSegment`.

5540 `Point3d getStartPosition()`

Egal welcher Raumbereich untersucht wird, allen möglichen `setShape...()`-Methoden ist eines Gemeinsam: Der Ausgangspunkt, ab dem gesucht werden soll. Dieser Punkt vom Typ `Point3d` kann mit Hilfe dieser Methode ermittelt werden.

5545 `PickResult[] pickAll()`

5550 Wurde ein `PickShape` spezifiziert, so ist eindeutig festgelegt worden, wo exakt nach 3D-Objekten gesucht werden soll. Das Ergebnis kann mit Hilfe mehrerer Methoden abgefragt werden. Rückgabewert ist dabei jeweils ein einzelnes Objekt oder ein Array von Objekten des Typs `PickResult`, das im Folgenden detailliert beschrieben wird.

5555 Die Methode `pickAll()` liefert ein Array aus diesen `PickResult`-Objekten zurück, das alle 3D-Objekte enthält, die innerhalb des angegebenen Raumbereiches gefunden wurden. Befand sich dort nichts, so ist der Rückgabewert `null`. Die Reihenfolge der 3D-Objekte ist dabei aus Sicht des Users willkürlich. Sie entspricht einer Anordnung, die durch die Strukturierung der Daten durch Java 3D festgelegt wird. Es kann bei der Verwendung dieser Methode also keinesfalls von einer bestimmten Reihenfolge ausgegangen werden, da diese auch bei der Verwendung der gleichen Version von Java 3D von Plattform zu Plattform verschieden sein könnte.

```
PickResult[] pickAllSorted()
```

5560 Hiermit wird ebenfalls ein Array aller betroffenen 3D-Objekte zurückgeliefert. Allerdings sind diese nach der Entfernung vom Ausgangspunkt des Pick-Ereignisses geordnet. An Position 0 im Array befindet sich demnach das Objekt, dass sich am nächsten befindet. Konnte innerhalb des gewünschten Raumbereiches nichts gefunden werden, so ist der Rückgabewert auch hier wieder `null`.

5565

```
PickResult pickAny()
```

Diese Methode liefert tatsächlich irgend einen Treffer zurück, so fern überhaupt ein 3D-Objekt gefunden werden konnte. Es handelt sich im Falle eines Treffers natürlich wieder um ein 3D-Objekt, das innerhalb des spezifizierten Raumbereiches liegt, jedoch  
5570 kann nicht vorhergesagt werden, welches Objekt das sein wird, wenn es mehrere gibt. Das hängt – wie bei `pickAll()` - wieder allein von der Java-3D-internen Datenverwaltung ab.

```
PickResult pickAny()
```

5575 Der Ergebniswert dieser Methode ist wieder eindeutig: sie liefert mit Hilfe des `PickResult`-Objektes dasjenige 3D-Objekt zurück, dass der Quelle des Pick-Ereignisses (bzw. anders gesagt: dem Ausgangspunkt des `PickShape`-Objektes) am nächsten ist.

Da für die Klasse `PickCanvas` bereits angegeben wurde, welche  
5580 Verwandtschaftsverhältnisse für diese herrschen, bietet die Ableitung der Klasse `PickTool` eigentlich nichts neues mehr:

```
java.lang.Object
```

```
com.sun.j3d.utils.picking.PickTool
```

5585

### 9.1.5.1 PickResult

Die Ergebnisse jeder Picking-Aktion, also der Suche nach 3D-Objekten innerhalb eines spezifizierbaren Raumbereiches, wurden – wie in den Beschreibungen der Methoden des  
5590 `PickTools` bereits gesehen – immer in Form eines `PickResult`-Objektes geliefert. Da es für eine weitere Verarbeitung aber nötig ist, zu wissen, welches 3D-Objekt innerhalb der Szene (also welches `Shape3D`, `OrientedShape3D`, `Primitive`) denn nun eigentlich ermittelt wurde, so muß es wohl auch einen Weg geben, diese Information aus dem `PickResult` zu extrahieren. Dieser Weg findet sich in den Methoden, die von der Klasse `PickResult` zur  
5595 Verfügung gestellt werden. Doch zuvor ein Blick auf einen der Konstruktoren, wie er beispielsweise von der Klasse `PickTool` verwendet wird und der bei eigenen Picking-Klassen ebenfalls genutzt werden könnte:

```
PickResult (SceneGraphPath sgp, PickShape ps)
```

5600

Der erste Parameter ist ein Objekt eines bisher unbekannten Typs. An dieser Stelle nur so viel: ein `SceneGraphPath` beschreibt ein Objekt innerhalb eines `SceneGraphs` in der

5605 Form, dass es den Weg von der Wurzel dieses SceneGraphen bis zu diesem Node spezifiziert. Und genau das ist hier ja gewünscht: Es soll ein bestimmter Node - nämlich das gefundene 3D-Objekt – im PickResult-Objekt gespeichert werden. Der zweite Parameter, `ps`, übergibt das PickShape-Objekt, das für das Picking verwendet wurde.

Viel interessanter sind hier jedoch die Methoden, die es unter anderem auch endlich ermöglichen, das eigentlich gewünschte 3D-Objekt zu ermitteln:

5610

```
Node getObject()
```

Es wird der Node aus dem SceneGraph zurückgeliefert, dessen Geometriedaten gefunden wurden. Im Falle des obigen Beispielprogrammes wäre das ein Shape3D-Objekt.

5615

```
SceneGraphPath getSceneGraphPath()
```

Wird diese Methode verwendet, so wird der Pfad zum gefundenen 3D-Objekt mit Hilfe eines SceneGraphPath-Objektes zurückgeliefert. Wurde ein anderer als der oben beschriebene Konstruktor verwendet, so ist der Rückgabewert statt dessen `null`.

5620

```
PickShape getPickShape()
```

Diese Methode liefert das PickShape-Objekt zurück, mit dem das Picking, das zu diesem PickResult geführt hat, ausgeführt wurde

5625

```
GeometryArray getGeometryArray()
```

```
GeometryArray[] getGeometryArrays()
```

Es wird das erste bzw. es werden alle GeometryArrays zurückgeliefert, die zu dem ermittelten 3D-Objekt gehören. Diese Methoden verhalten sich also ähnlich wie `getGeometry()` und `getAllGeometries()` der Klasse Shape3D.

5630

```
PickIntersection getClosestIntersection (Point3d pt)
```

5635 Diese Methode ist interessant, wenn der Abstand zum nächstmöglichen Schnittpunkt mit Geometriedaten des zugehörigen 3D-Objektes ermittelt werden soll. Sie liefert diese Information mit Hilfe eines PickIntersection-Objektes zurück, welches gleich im Anschluß beschrieben wird. Als Parameter erwartet diese Methode eine Positionsinformation `pt`, die angibt, bezugnehmend auf welchen Punkt im virtuellen Raum der nächstmögliche Schnittpunkt gefunden werden soll.

5640 Die Ableitungsverhältnisse der Klasse PickResult liefern dieses mal keine neuen Erkenntnisse:

```
java.lang.Object
```

```
com.sun.j3d.utils.picking.PickResult
```

#### 5645 9.1.5.1.1 PickIntersection

Wie im vorhergehenden Abschnitt gesehen, läßt es sich mit den Funktionalitäten der Klasse PickResult relativ einfach ermitteln, in welchem Abstand sich das PickShape-Objekt mit den Geometriedaten des ermittelten 3D-Objektes schneidet. „PickShape“ muß  
5650 hier allerdings eingeschränkt werden. Ein wirklich eindeutiger Abstand zu einem Schnittpunkt kann natürlich nur dann zurückgeliefert werden, wenn der Schnittpunkt auch tatsächlich nur ein Punkt ist und nicht ein Strahl oder gar eine Fläche. Das heißt, nur wenn ein eindimensionales PickShape-Objekt für das Picking verwendet wurde, kann ein eindeutiger und exakter Wert für den Abstand ermittelt und zurückgegeben werden. Wie  
5655 bereits erwähnt aber noch nicht detailliert besprochen, stellen die Klassen PickRay und PickSegment solche eindimensionale Picking-Objekte dar.

Der Konstruktor der Klasse PickShape ist von außerhalb der Package com.sun.j3d.utils.picking nicht zugänglich, weshalb hier auch nur die  
5660 verschiedenen Methoden, mit denen sich der Abstand ermitteln läßt, von Interesse sein sollen:

```
boolean geometryIsIndexed()
```

Es wird die Information zurückgeliefert, ob die Geometriedaten indiziert sind, also  
5665 ob die Polygone durch den indirekten Verweis auf die Vertices definiert werden. Diese Information sollte im allgemeinen jedoch nichts neues sein, wenn das gefundene 3D-Objekt und damit dessen Aufbau bereits bekannt sind.

```
Point3d getClosestVertexCoordinates()
```

5670 Es werden die Koordinaten des nächstgelegenen Vertex innerhalb dieses PickIntersection-Objektes zurückgeliefert.

```
int getClosestVertexIndex()
```

5675 Wurde beim zugehörigen Node der Weg der indirekten Geometriedaten verwendet, so kann mit dieser Methode der Indexwert des Vertex ermittelt werden, der innerhalb der Definition dieses PickIntersection-Objektes am nächsten liegt.

```
double getDistance()
```

5680 Wenn die Entfernung vom Startpunkt bis zum gewünschten Schnittpunkt mit den Geometriedaten bekannt und eindeutig ist (also wenn ein eindimensionales PickShape-Objekt verwendet wurde, dann gibt diese Methode den exakten Abstand zurück.

```
Color4f getPointColor()
```

5685 Diese Methode liefert die Farbe des Polygons an der Schnittstelle zurück. Wurden Geometriedaten verwendet, in denen keine Farbinformationen enthalten sind, so ist der

Rückgabewert `null`. Bei Farbinformationen vom Typ **COLOR\_3** wird der vierte Farbwert, also `w`, auf 1.0 festgesetzt.

```
Point3d getPointCoordinates()
```

5690        Im Gegensatz zu `getClosestVertexCoordinates()` liefert diese Methode die Koordinaten des exakten Schnittpunktes zurück und nicht nur einen Indexwert, der auf die Koordinaten verweist. Voraussetzung dafür ist auch hier wieder, dass ein eindimensionales `PickShape`-Objekt verwendet wurde, da sonst schließlich kein Schnittpunkt existiert, sondern eine Strecke bzw. eine Fläche, der dann kein einzelner, eindeutiger Punkt mehr zugeordnet werden kann.

```
Vector3f getPointNormal()
```

Es wird der Normal-Wert des Schnittpunktes zurückgeliefert. Enthält das zugehörige `GeometryArray` keine Normals, so ist der Rückgabewert `null`.

5700

```
TexCoord3f getPointTextureCoordinate(int index)
```

Diese Methode liefert die Texturkoordinaten für den Schnittpunkt zurück. Der Parameter `index` legt dabei fest, innerhalb welchen Texturkoordinaten-Sets die gewünschten Koordinaten ermittelt werden sollen. Enthält das `GeometryArray` keine Texturkoordinaten, so ist der Rückgabewert `null`. Enthält es nur zweidimensionale Koordinaten (**TEXTURE\_COORDINATE\_2**), so wird der zusätzliche Koordinatenwert `z` auf 0.0 gesetzt.

5705

```
int[] getPrimitiveColorIndices()
```

5710        Ist der Node, der die Geometriedaten beinhaltet nicht vom Typ `Shape3D` oder `OrientedShape3D` sondern ein `Primitive`, so ist es mit dieser Methode möglich, die Farbindices für das `Primitive` zu ermitteln.

```
Color4f[] getPrimitiveColors()
```

5715        Auch diese Methode ist für ein `Primitive` von Interesse, sie liefert die Farbwerte zurück oder aber `null`, wenn das `Primitive` keine Farbinformationen enthält.

```
int[] getPrimitiveCoordinateIndices()
```

5720        Es werden die Koordinaten-Indices des `Primitive`-Objektes zurückgeliefert, für das das zugehörige `PickIntersection`-Objekt gültig ist.

```
int[] getPrimitiveCoordinateIndices()
```

Im Gegensatz zur vorangegangenen Methode liefert diese die exakten Koordinaten und nicht nur die Indices auf diese zurück.

5725

```
int[] getPrimitiveNormalIndices()
```

Diese Methode dient dazu, die Normal-Indices für das zugehörige Primitive-Objekt zu ermitteln. Enthält dieses keine Normal-Informationen, so ist der Rückgabewert wiederum `null`.

5730

```
Vector3f[] getPrimitiveNormals()
```

Statt nur der Indices liefert diese Methode die Normal-Werte selber, so fern das Primitive-Objekt diese Informationen enthält.

5735 `int[] getPrimitiveTexCoordIndices(int index)`

Auch für Texturkoordinaten existiert für Primitives eine passende Methode. So fern überhaupt vorhanden, liefert diese die Indices auf die Texturkoordinaten für das mit `index` spezifizierte Koordinatenset zurück.

5740 `TexCoord3f[] getPrimitiveTexCoords(int index)`

Die Existenz dieser Methode lies sich fast zwangsläufig aus der vorangegangenen schließen. Diese liefert statt der Indices die Texturkoordinaten selber zurück. Um welches Texturkoordinatenset es dabei gehen soll, legt wiederum der Wert des Parameters `index` fest.

5745

```
int [] getPrimitiveVertexIndices()
```

Die letzte für Objekte vom Typ Primitive noch fehlende Methode liefert die Vertex-Indices für das Objekt zurück.

5750 Mit einem abschließenden, nicht weiter erläuterungsbedürftigen Blick auf die Verwandtschaftsverhältnisse der Klasse `PickIntersection` soll dieser Abschnitt dann auch beendet werden, da sie zwar interessant und wichtig, aber nicht so kompliziert ist, dass man darüber Romane schreiben müßte:

5755 `java.lang.Object`

```
com.sun.j3d.utils.picking.PickIntersection
```

### 9.1.5.2 PickShape

5760 Bei der Klasse `PickTool` wurden mehrfach verschiedene `PickShapes` erwähnt. Wie dort kurz beschrieben, dienen diese dazu, Raumbereiche zu definieren, innerhalb der das Picking, also die Suche nach darin enthaltenen 3D-Objekten, ausgeführt werden soll. Wie dieser Raumbereich nun aussehen soll, wie viele und welche Dimensionen er haben soll und ob er endlich sein oder gar unendlich weiter verlaufen soll, wird dabei durch die Art des `PickShapes` festgelegt. Die Klasse `PickShape` selbst ist schließlich nur die Basisklasse dieser Art von Objekten, die deswegen auch nur die grundlegendsten, wichtigsten Methoden mitliefert. Sowohl die Basisklasse `PickShape` als auch ihre Unterklassen können innerhalb eigener Programme und ganz unabhängig vom `PickTool`

5765

5770 verwendet werden, da sich die Funktionalitäten, die benötigt werden, um ein Picking auszuführen, sowieso in anderen Klassen finden: In der BranchGroup und im Shape3D.

Es erscheint logisch, dass es dadurch beispielsweise auch möglich ist, eine Kollisionsvermeidung ganz unabhängig von den vorgefertigten Klassen der Package `com.sun.j3d.utils.picking` zu realisieren und ohne Zuhilfenahme zusätzlicher Tools alle dazu nötigen Informationen zu ermitteln.

5775

Da die Klasse `PickShape` selber abstrakt ist und da sie selber keine eigenen Methoden mitbringt, findet sich alles interessante in den von ihr abgeleiteten Klassen. Letztendlich soll die Ableitung dieser Basisklasse jedoch nicht verschwiegen werden, sei sie auch noch so simpel:

5780

```
java.lang.Object
 javax.media.j3d.PickShape
```

#### 5785 9.1.5.2.1 PickRay

Diese Klasse spezifiziert einen Strahl mit einem definierten Ausgangspunkt und einem unendlich weit entfernten Endpunkt. Da ein Strahl immer nur eine Längeninformation (oder hier genauer nur eine Richtungsinformation) kennt, handelt es sich also um ein eindimensionales `PickShape`. Demzufolge muß es mit dem `PickRay` auch möglich sein, einen exakten Schnittpunkt sowie eine eindeutige Entfernung zu diesem zu ermitteln.

5790

```
PickRay(Point3d origin, Vector3d direction)
```

5795 Entsprechend diesen Eigenschaften erwartet der Konstruktor zum einen die Koordinaten des Ausgangspunktes `origin` und zum anderen die Richtung `direction`, in die der Strahl verlaufen soll.

```
void get(Point3d origin, Vector3d direction)
5800 void set(Point3d origin, Vector3d direction)
```

Die einzigen beiden Methoden dieser Klasse bearbeiten die gleichen Parameter wie der oben beschriebene Konstruktor, sie erlauben es, die aktuellen Daten, die dieses Objekt definieren, entweder zu ermitteln oder aber auf neue Werte zu setzen. Bei diesen Daten handelt es sich wieder um die Ausgangsposition `origin` des Strahls sowie die Richtung `direction`, in die er verläuft.

5805

Diese Klasse leitet sich direkt vom `PickShape` ab:

```
java.lang.Object
```

```
5810 javax.media.j3d.PickShape
 javax.media.j3d.PickRay
```

#### 9.1.5.2.2 PickSegment

5815 Das PickSegment ist das endliche Gegenstück zum PickRay, es ist also ein Strahl mit einem definierten Startpunkt und einem eben so definierten Endpunkt:

```
PickSegment(Point3d start,Point3d end)
```

5820 Demzufolge erwartet der Konstruktor auch die Koordinaten der beiden Enden dieses Strahls in Form der Point3d-Objekte `start` und `end`.

```
void get(Point3d start, Point3d end)
void set(Point3d start, Point3d end)
```

5825 Auch hier erlauben es die einzigen beiden Methoden, die diese Klasse zur Verfügung stellt, die Daten, die das Objekt definieren, zu ermitteln oder aber auf neue Werte zu setzen. Diese Daten sind - wie schon vom Konstruktor bekannt – der Startpunkt `start` des Strahls sowie sein Endpunkt `end`.

5830 Interessanterweise besteht keine direkte Verwandtschaft zum PickRay:

```
java.lang.Object
 javax.media.j3d.PickShape
 javax.media.j3d.PickSegment
```

5835

#### 9.1.5.2.3 PickCylinder

Hierbei handelt es sich um eine Basisklasse für PickShapes, die einen zylindrischen Raumbereich definieren. Da sie abstrakt ist, kann sie nicht direkt instantiiert werden.

5840 Nutzbare Ableitungen von der Klasse PickCylinder finden sich aber mit den folgenden beiden Klassen, die demzufolge auch die verschiedenen `get()`-Methoden des PickCylinders erben:

```
void getDirection(Vector3d direction)
```

5845 Diese Methode dient der Ermittlung der Richtung, in die der zugehörige PickCylinder verläuft.



```
void getOrigin(Point3d origin)
```

Die Koordinaten, die den Ausgangspunkt des zugehörigen Objektes darstellen,  
5850 werden von dieser Methode in das als Parameter übergebene Point3D-Objekt kopiert.

```
double getRadius()
```

Abweichend von den beiden anderen Methoden liefert `getRadius()` den Radius  
des `PickCylinders` als echten Returnwert zurück.

5855

#### – **PickCylinderRay**

Dieses `PickShape` spezifiziert einen unendlichen zylindrischen Raumbereich:

```
PickCylinderRay(Point3d origin, Vector3d direction, double radius)
```

5860

Die beiden ersten Parameter sind dabei vom `PickRay` bekannt, sie spezifizieren den  
Ausgangspunkt sowie die Richtung, in die der Zylinder verlaufen soll. Neu kommt hier  
der Parameter `radius` hinzu, der dieses `Pick`-Objekt dreidimensional macht. Er  
spezifiziert den Radius des Zylinders.

5865

```
void set(Point3d origin, Vector3d direction, double radius)
```

Interessanterweise bietet diese Klasse nur eine Methode, die es erlaubt, den  
Ausgangspunkt `origin`, die Richtung `direction`, in die der Zylinder verläuft sowie  
seinen `radius` auf neue Werte zu setzen.

5870

Die Klasse `PickCylinderRay` leitet sich wie angekündigt direkt vom `PickCylinder` ab:

```
java.lang.Object
 javax.media.j3d.PickShape
 javax.media.j3d.PickCylinder
 javax.media.j3d.PickCylinderRay
```

5875

#### – **PickCylinderSegment**

Und auch hier findet sich wieder ein endliches Gegenstück zur vorhergehend  
beschriebenen Klasse:

5880

```
PickCylinderSegment(Point3d origin, Point3d end, double radius)
```

In bekannter Weise wird dieses durch einen Start- und einen Endpunkt definiert, zu  
dem jetzt – auf Grund der zylindrischen Form – wieder ein Wert für den `radius`  
hinzukommt, welcher durch den letzten der drei Parameter übergeben wird.

5885

```
void getEnd(Point3d end)
```

Diese Methode kopiert die Koordinaten des `PickCylinderSegment`-Endpunktes in  
das als Parameter übergebene `Point3d`-Objekt.

5890

```
void set(Point3d origin, Point3d end, double radius)
```

Wie von den vorhergehend bereits beschriebenen `PickShapes` her bekannt, gibt es  
auch für dieses wieder eine `set()`-Methode, die es ermöglicht, neue Werte für den  
Ausgangspunkt `origin`, den Endpunkt `end` sowie den `radius` des

5895

PickCylinderSegment-Objektes zu übergeben.

Der Vollständigkeit halber auch hier wieder ein Blick auf die Ableitung dieser Klasse:

```
5900 java.lang.Object
 javax.media.j3d.PickShape
 javax.media.j3d.PickCylinder
 javax.media.j3d.PickCylinderSegment
```

#### 5905 9.1.5.2.4 PickCone

Mit PickCone ist eine weitere abstrakte Klasse zu besprechen, bei der wieder die beiden direkt von ihr abgeleiteten, je einmal endlichen und einmal unendlichen PickShape-Klassen von eigentlichem Interesse sind. PickCone ans sich definiert einen kegelförmigen Raumbereich, bei dem die Spitze des Kegels gleichzeitig den Ausgangspunkt des Pick-Objektes darstellt. Man könnte diesen Kegel also mit einem Scheinwerfer vergleichen, der in eine Szene hineinleuchtet und alle Objekte ermittelt, die in diesem Scheinwerferlicht auftauchen.

5915 Entsprechend den gemeinsamen Eigenschaften, die auch alle von PickCone abgeleiteten Klassen haben, gibt es für diese auch gemeinsam zu verwendende Methoden:

```
void getDirection(Vector3d direction)
```

5920 Auch ein PickCone hat eine Richtung, in den er verläuft. Der Richtungsvektor, der diese beschreibt, wird von dieser Methode in das als Parameter übergebene Objekt `direction` kopiert.

```
void getOrigin(Point3d origin)
```

5925 Eine weitere Eigenschaft, der Ursprungspunkt des PickCone, wird von dieser Methode behandelt, die es erlaubt dessen Koordinaten zu ermitteln.

```
double getSpreadAngle()
```

5930 Auch der aktuelle Öffnungswinkel kann aus einem PickCone-Objekt geholt werden. Diese Method liefert ihn als Returnwert in der Einheit Radians. Der Öffnungswinkel legt fest, wie spitz der Konus sein soll.

#### - PickConeRay

Als erstes soll wieder die unendliche der beiden PickCone-Klassen beschrieben werden:

```
5935 PickConeRay(Point3d origin, Vector3d direction, double spreadAngle)
```

Neben dem Ausgangspunkt `origin` des Kegels und der Richtung `direction`, in die er sich erstreckt, findet sich hier mit dem letzten Parameter `spreadAngle` dessen

5940 Öffnungswinkel. Um so kleiner dieser ist, um so spitzer wird der Kegel, umgekehrt wird er um so stumpfer, um so größer der Winkel wird.

```
void set(Point3d origin, Vector3d direction, double spreadAngle)
```

5945 Auch diese Klasse bietet wieder eine `set()`-Methode, die es erlaubt, mit dem Ausgangspunkt `origin`, der Richtung `direction`, in die dich das Objekt erstreckt und seinem `spreadAngle` sämtliche relevanten Eigenschaften auf neue Werte zu setzen.

Die Ableitung dieser Klasse bietet auch nichts überraschendes mehr, da ja bereits erwähnt wurde, dass sie sich direkt vom abstrakten `PickCone` herleitet:

```
5950 java.lang.Object
 javax.media.j3d.PickShape
 javax.media.j3d.PickCone
 javax.media.j3d.PickConeRay
```

#### – **PickConeSegment**

Nun fehlt, um die Gruppe der `PickCone`-Klassen abzuschließen, nur wieder die endliche Variante des Pick-Kegels:

```
5960 PickConeSegment(Point3d origin, Point3d end, double spreadAngle)
```

Da Java 3D sehr gut und konsequent strukturiert wurde, erschließen sich die Parameter eigentlich von selbst: es finden sich wieder Start- und Endpunkt `origin` und `end` des Kegels sowie der eben so nötige Öffnungswinkel `spreadAngle`.

```
5965 void getEnd(Point3d end)
```

Mittels dieser Methode läßt sich der Endpunkt des `PickConeSegment`-Objektes ermitteln, dessen Koordinaten werden in bekannter Weise in das übergebene `Point3d`-Objekt `end` kopiert.

```
5970 void set(Point3d origin, Point3d end, double spreadAngle)
```

Sinn und Funktionalität dieser Methode folgen wieder dem Prinzip, das von den vorangegangenen `PickShapes` her bereits bekannt ist. Sie erlaubt es, die Eigenschaften Ausgangspunkt `origin`, Endpunkt `end` sowie den Öffnungswinkel `spreadAngle` des `PickConeSegment`s auf neue Werte zu setzen.

5975

Auch hier findet sich wieder die direkte Verwandtschaft mit der Klasse `PickCone`:

```
5980 java.lang.Object
 javax.media.j3d.PickShape
 javax.media.j3d.PickCone
 javax.media.j3d.PickConeSegment
```

#### **9.1.5.2.5 PickPoint**

5985

Ein letztes Picking-Objekt ist der `PickPoint`. Damit kann eigentlich kein Raumbereich mehr definiert werden, da dieser Punkt keine Dimensionen besitzt. Dieses sehr spezielle Objekt findet sicher nur relativ wenige sinnvolle Anwendungsmöglichkeiten, da es im allgemeinen

5990 eher nicht zweckmäßig sein dürfte, nur einen ganz spezifischen Punkt zu untersuchen. Nichts desto trotz gibt es natürlich sinnvolle Nutzungsmöglichkeiten, die die Existenz dieser Klasse nötig machen.

```
PickPoint(Point3d location)
```

5995 Wie der Name der Klasse es bereits verrät, spezifiziert diese Klasse nur einen einzigen Punkt in der virtuellen 3D-Welt, der bei der Instantiierung eines solchen Objektes mit Hilfe des einzigen Parameters `location` in Form eines `Point3d`-Objektes festgelegt wird. Dieser Wert findet sich auch bei den beiden Methoden wieder:

```
6000 void get(Point3d location)
 void set(Point3d location)
```

Diese beiden Methoden erlauben es, die aktuellen Koordinaten des `PickPoint`-Objektes zu ermitteln oder aber neue Koordinaten dafür festzulegen.

6005 Der `PickPoint` leitet sich wieder in alt bekannter Weise ab:

```
java.lang.Object
 javax.media.j3d.PickShape
 javax.media.j3d.PickPoint
```

6010

### 9.1.5.3 Picking-Funktionalitäten der `BranchGroup`

Wie zu sehen war, bieten die `PickShapes` keinerlei Möglichkeit, zu ermitteln, ob sich in dem Raumbereich, den sie definieren, ein Objekt befindet oder nicht. Die dafür nötigen Methoden finden sich statt dessen in der Klasse `BranchGroup`. Das mag auf den ersten Blick unsinnig erscheinen, was es jedoch keinesfalls ist. Vielmehr entspricht das der objektorientierten Strukturierung, der auch die Java 3D API unterliegt und die hier recht konsequent realisiert wurde. Wie schon beim `PickTool` gesehen, wurde nicht nur der Raumbereich spezifiziert, in dem nach 3D-Objekten „gepickt“ werden soll, es wurde auch angegeben, innerhalb welches Teil-SceneGraphen nach den jeweiligen Objekten gesucht werden soll. Und genau das passiert jetzt implizit dadurch, dass sich die Methoden zur Auswertung in der Klasse `BranchGroup` wiederfinden. Es wird also immer nur in den Nodes unterhalb dieser `BranchGroup` nach getroffenen Objekten gesucht.

6025 Da `BranchGroups` am Anfang dieses Dokuments bereits beschrieben wurden, kann hier auf eine vollständige Erklärung eigentlich verzichtet werden. Es genügt ein Blick auf die zuvor verschwiegenen Methoden zur Ermittlung der Picking-Ergebnisse, wobei deren prinzipielle Wirkungsweise von den äquivalenten Methoden der Klasse `PickTool` her bekannt sein sollte. Allen diesen Methoden ist dabei gemeinsam, dass als einziger Parameter ein `PickShape`-Objekt `pickShape` übergeben werden muß, welches den

6030

Raumbereich festlegt, in dem das Picking erfolgen soll, also innerhalb dessen 3D-Objekte aufgefunden werden sollen:

```
SceneGraphPath[] pickAll(PickShape pickShape)
```

- 6035 Es wird ein Array aus SceneGraphPath-Objekten zurückgeliefert, das alle durch das Picking gefundenen Objekte enthält. Die Reihenfolge ist dabei beliebig und für den Benutzer nicht vorhersehbar. Es kann also auch nicht von einer bestimmten Reihenfolge ausgegangen werden, diese könnte auf einer anderen Plattform als der aktuell verwendeten völlig anders aussehen.
- 6040 Wurden keine Objekte gefunden, so wird `null` zurückgeliefert.

```
SceneGraphPath[] pickAllSorted(PickShape pickShape)
```

- 6045 Diese Methode liefert ebenfalls ein Array aus SceneGraphPath-Objekten zurück, das alle gefundenen Objekte beinhaltet. Der Unterschied zur Vorhergehenden ist jedoch, das diese jetzt geordnet sind. Um so näher ein solches Objekt sich am Ausgangspunkt des PickShapes befindet, um so weiter vorne in diesem Array befindet es sich. Dasjenige Objekte, das sich an der Position 0 in diesem Array befindet, entspricht dem, das auch von der Methode `pickClosest()` zurückgeliefert wird.
- 6050 Wurden keine Objekte gefunden, so ist der Rückgabewert dieser Methode ebenfalls `null`.

```
SceneGraphPath pickAny(PickShape pickShape)
```

- 6055 Der Rückgabewert dieser Methode entspricht dem, was man erhalten würde, wenn man den ersten Index des Arrays verwendet, das von der Methode `pickAll()` zurückgegeben wird. Es handelt sich somit um irgend eines der Objekte, die durch die BranchGroup und den festgelegten Raumbereich spezifiziert wurden.
- Wurde kein 3D-Objekt gefunden, so ist der Rückgabewert `null`.

```
SceneGraphPath pickClosest(PickShape pickShape)
```

- 6060 Wie bereits kurz angeschnitten liefert diese Methode mit dem Rückgabewert einen Verweis auf dasjenige 3D-Objekt zurück, das dem Ausgangspunkt des `pickShape` am nächsten ist, innerhalb des damit spezifizierten Raumbereiches liegt und Child der BranchGroup ist, die diese Methode anbietet.
- 6065 Auch hier wird `null` zurückgeliefert, wenn diese Bedingungen auf kein Objekt zutreffen.

#### 9.1.5.4 Das Picking-Ergebnis SceneGraphPath

- 6070 Bisher stellt sich der Ablauf einer komplett selbst geschriebenen Picking-Funktionalität unabhängig vom PickTool der Package `com.sun.j3d.utils.picking` folgendermaßen dar:

- es wird ein PickShape-Objekt eines für die Aufgabe zweckmäßigen Typs erzeugt
- es wird die Basis-BranchGroup ermittelt, unterhalb derer sich das gesuchte Objekt befinden muß (im einfachsten Fall wird die BranchGroup verwendet, die sich auf

oberster Ebene befindet und unterhalb derer sich die gesamte Szene verzweigt)

- 6075 - mit Hilfe dieser BranchGroup wird ermittelt, welches Objekt bzw. welche Objekte durch diese Picking-Operation gefunden wurden.

6080 Da das Ergebnis des letzten Schrittes jedoch ein SceneGraphPath-Objekt ist und für eine weitere Verwendung jedoch der eigentliche Node aus dem SceneGraphen benötigt werden würde, der durch das Picking ermittelt wurde, ist es notwendig, einen Weg zu finden, diesen Node (der z.B. ein Shape3D oder ein Primitive sein kann) zu ermitteln. Die Mittel und Wege dafür müssen sich geradezu zwangsläufig in der Klasse SceneGraphPath finden.

6085 Ein SceneGraphPath selbst ist eine Art Beschreibung, die den Weg von der Wurzel eines SceneGraphen bis zu einem bestimmten Endpunkt beinhaltet. Dieser Endpunkt ist in diesem speziellen Fall das gesuchte, per Picking ermittelte 3D-Objekt. Diese Beschreibung kann intern dabei im Extremfall nur aus dem Startpunkt und dem Endnode bestehen. Andererseits ist es auch möglich, dass sich Nodes, die sich zwischen diesen  
6090 beiden Punkten befinden ebenfalls enthalten sind. Das hängt ganz davon ab, wie und für welchen Zweck das jeweilige SceneGraphPath-Objekt erzeugt wurde. Für den Anwendungsfall „Picking“ genügt indes meist die Minimalversion aus Start- und Endnode. Die Klasse SceneGraphPath bietet unter anderem passende Methoden an, die einen Zugriff auf genau den gewünschten Node ermöglichen:

6095  
`Node getObject()`

Es wird der Node zurückgeliefert, der dem Endpunkt des SceneGraphPath entspricht, es handelt sich hier also um das mittels Picking gesuchte Objekt.

6100 `Node getNode(int index)`

Enthält das SceneGraphPath-Objekt mehrere Nodes, die sich auf dem Weg zum Endpunkt dieses Pfades befinden, so können diese mittels ihrer Position `index` ermittelt werden.

6105 Des weiteren bringt die Klasse SceneGraphPath noch die folgenden, ebenfalls erwähnenswerten Methoden mit:

`boolean equals(java.lang.Object o1)`  
`boolean equals(SceneGraphPath testPath)`

6110 Der Rückgabewert dieser Methoden ist `true`, wenn das aktuelle sowie das als Parameter übergebene SceneGraphPath-Objekt den gleichen Pfad inklusive der gleichen, eventuell vorhandenen Zwischennodes beschreiben. Sie bieten also die Möglichkeit, verschiedene SceneGraphPath-Objekte zu vergleichen.

6115 `boolean isSamePath(SceneGraphPath testPath)`

Auch diese Methode führt einen Vergleich mit einem zweiten SceneGraphPath-

Objekt aus. Im Gegensatz zu `equals()` liefert diese jedoch auch dann `true` zurück, wenn nur der Start- und Endpunkt identisch sind, die möglichen Nodes zwischen diesen beiden jedoch nicht.

6120

```
Transform3D getTransform()
```

```
void setTransform(Transform3D trans)
```

Zu einem `SceneGraphPath`-Objekt kann auch eine Transformation gehören. Mittels dieser Methoden ist es möglich, diese Transformation zu ermitteln oder aber eine neue festzulegen. Wurde das `SceneGraphPath`-Objekt als Ergebnis einer Picking- oder einer Kollisionsoperation erzeugt, so ist diese Transformation mit der Transformation von lokalen Koordinaten zu den Koordinaten der virtuellen Welt identisch, die zum Zeitpunkt des Pickings (oder der Kollision) gültig war.

6125

6130 `int hashCode()`

Der Returnwert dieser Methode ist ein Hash-Code, der basierend auf den Daten dieses Objektes erzeugt wurde.

```
int nodeCount()
```

6135 Der Rückgabewert dieser Methode ist die Gesamtanzahl Nodes, die in diesem Objekt vorkommen.

```
void set(SceneGraphPath newPath)
```

6140 Diese Methode ersetzt den aktuellen `SceneGraphPath` praktisch vollständig in dem die Daten des als Parameter übergebenen `newPath` übernommen werden und alle eventuell vorhandenen alten Werte ersetzen.

```
void setNode(int index, Node newNode)
```

6145 Der an der Position `index` bereits vorhandene Node wird von dieser Methode durch `newNode` ersetzt.

```
void setNodes(Node[] nodes)
```

Diese Methode setzt die Nodes innerhalb des als Parameter übergebenen Arrays als die Nodes dieses `SceneGraphPath`-Objektes.

6150

```
void setObject(Node object)
```

Das Gegenstück zu dieser Methode ist das bereits erwähnte `getObject()`. Während diese den durch dieses `SceneGraphPath`-Objekt beschriebenen Endpunkt zurückliefert, ist es mit `setObject()` möglich, einen neuen Endnode festzulegen.

6155

Kurz und knapp und als Abschluß der Beschreibung dieser Klasse wieder die Ableitung selbiger:

java.lang.Object

6160        **javax.media.j3d.SceneGraphPath**

### 9.1.5.5 Picking-Funktionalitäten des Shape3D

Wie im vorhergehenden Abschnitt beschrieben läßt sich mittels der Methode  
6165 `getObject()` der Node ermitteln, der durch das Picking gefunden wurde. Wenn es sich  
dabei um ein Objekt vom Typ `Shape3D` handelt, bietet dieses noch weitergehende  
Möglichkeiten der Analyse, die sich auf die Geometriedaten genau dieses `Shape3D`-  
Objektes bezieht. Das ist auch wieder der Grund, warum sich diese Picking-  
Funktionalitäten in Methoden der Klasse `Shape3D` finden:

6170

```
boolean intersect(SceneGraphPath path, PickRay pickRay, double[]
dist)
```

Es wird überprüft, ob es Schnittpunkte mit der Geometrie des `Shape3D`-Objektes  
gibt. Existieren diese, ist der Rückgabewert `true`, anderenfalls wird `false`  
6175 zurückgeliefert.

Der Parameter `path` legt dabei den `SceneGraphPath` fest, der zu diesem `Shape3D`-  
Objekt führt, er kann also einfach aus dem vorhergehenden Arbeitsschritt übernommen  
werden. Der zweite Parameter wiederum spezifiziert ein `PickRay`-Objekt, das angibt, in  
welcher Richtung nach den Schnittpunkten mit der Geometrie gesucht werden soll. Diese  
6180 Methode ist auf ein eindimensionales `PickRay`-Objekt festgelegt, weil nur dieses in der  
Lage ist, die Abstände zu den verschiedenen möglichen Schnittpunkten mit Hilfe des als  
letzten Parameter übergebenen Arrays vom Typ `double` zurück zu liefern. Das heißt, nach  
dem Ausführen dieser Methode enthält das Array `dist` alle Abstände vom  
Ausgangspunkt des `PickRay`-Objektes bis zu den einzelnen Schnittpunkten mit der  
6185 Geometrie des `Shape3D`-Objektes.

```
boolean intersect(SceneGraphPath path, PickShape pickShape)
```

Diese Methode überprüft, ob es Schnittpunkte der eigenen Geometrie mit dem  
angegebenen `pickShape`-Objekt gibt. Als erster Parameter wird auch hier wieder der  
6190 `SceneGraphPath` benötigt, in dem sich dieses Objekt (also `this`) befindet. Gibt es  
mindestens einen Schnittpunkt, so ist der Rückgabewert `true`.

```
boolean intersect(SceneGraphPath path, PickShape pickShape, double[]
dist)
```

6195 Es wird auch hier überprüft, ob es Schnittpunkte mit der Geometrie des `Shape3D`-  
Objektes gibt. Existieren diese, ist der Rückgabewert `true`, anderenfalls wird `false`  
zurückgeliefert.

Der Parameter `path` legt dabei wieder den `SceneGraphPath` fest, der zu diesem  
`Shape3D`-Objekt führt. Der zweite Parameter spezifiziert im Gegensatz zur ersten  
6200 Methode ein beliebiges `PickShape`-Objekt, das angibt, in welcher Richtung nach den  
Schnittpunkten mit der Geometrie gesucht werden soll. Da hier nicht explizit ein



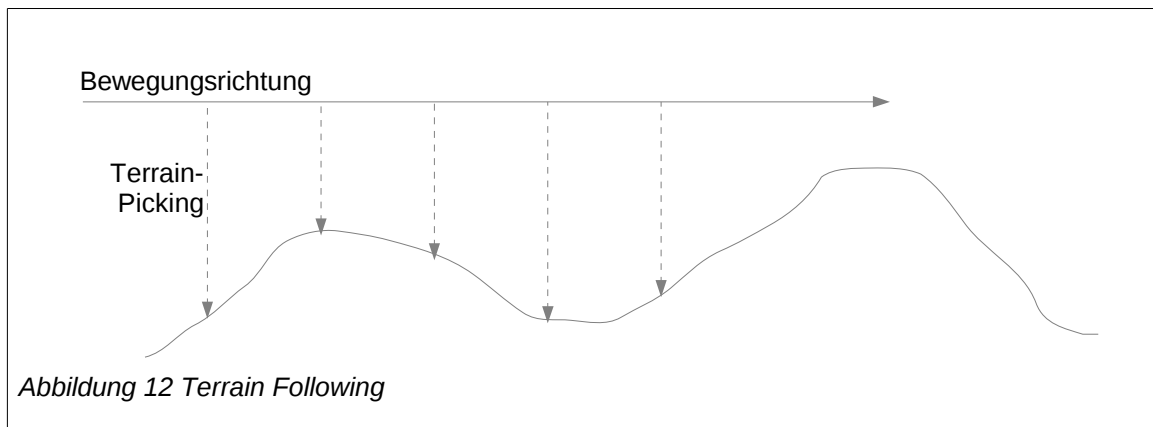
eindimensionales PickRay-Objekt gefordert wird, muß auch nicht unbedingt ein eindeutiger Abstand zu den Schnittpunkten existieren, der mit Hilfe des als letzten Parameter übergebenen Arrays vom Typ double zurückgeliefert werden könnte.  
6205 Deswegen enthält das Array `dist` nach dem Ausführen dieser Methode den jeweils kleinsten Abstand vom Ausgangspunkt des PickShape-Objektes bis zu den einzelnen Schnittpunkten bzw. -flächen mit der Geometrie des Shape3D-Objektes.

## 9.2 Terrain Following und Collision Prevention

6210 Mit den jetzt bekannten Möglichkeiten und Methoden, das Vorhandensein von 3D-Objekten in einer bestimmten Richtung und innerhalb eines bestimmten Raumbereiches festzustellen, ist es möglich, die ViewPlatform so zu steuern, dass sich eine realistisch wirkende Bewegung durch die virtuelle Welt ergibt.

6215 Wie sollte so eine Steuerung am Besten aussehen? Nun zum einen muß eine Bewegung realisiert werden, die z.B. mit Hilfe eines Behaviors Tastatureingaben umsetzt. Wie das zu realisieren wäre, wurde in vorangegangenen Abschnitten bereits näher beschrieben. Des weiteren ist es erforderlich, in Bewegungsrichtung zu überprüfen, ob sich die  
6220 ViewPlatform 3D-Objekten nähert, mit denen sie kollidieren würde, wenn die aktuelle Bewegung unverändert fortgesetzt werden würde. Diese so genannte Collision Prevention (also Kollisionsvermeidung) läßt sich beispielsweise mit einem (in Richtung des aktuellen Bewegungsvektors) nach vorne gerichteten PickCylinderSegment realisieren, der etwa so lang ist, wie der nächste Schritt. Finden sich innerhalb dieses PickCylinderSegments 3D-  
6225 Objekte, so kann davon ausgegangen werden, dass dieser nächste Schritt zu einer Kollision führt und deswegen besser vermieden werden sollte. Wird die Bewegung statt dessen nicht ausgeführt, bleibt die ViewPlatform an der letzten Position und damit kurz vor diesem Objekt stehen. Ist die Schrittweite nicht zu groß, wirkt das bereits so, als ob der Benutzer gegen dieses 3D-Objekt gelaufen ist.  
6230 Ist die jeweilige Schrittweite hingegen so groß, dass anschließend ein sichtbarer Abstand bestehen würde, so kann mit der Methode  
`intersect(SceneGraphPath,PickShape,double[])` zusätzlich der exakte Abstand zu diesem Objekt ermittelt werden. Aus dem Abstand, der sich an Indexposition 0 des übergebenen double-Arrays findet, kann dann die maximal zulässige Schrittweite ermittelt  
6235 werden, so dass es möglich ist, die ViewPlatform noch bis exakt vor das gefundene 3D-Objekt zu manövrieren, aber nicht in diese hinein.

Doch das ist unter Umständen nicht alles, was für eine realistische Bewegung benötigt wird. Man stelle sich einen ungleichmäßigen Untergrund vor, wie der von einer hügeligen  
6240 Landschaft. Mit dem bisherigen, lediglich vorwärts gerichteten Picking würde der Benutzer über Täler hinwegschweben und vor einem flachen Bergrücken stehenbleiben. Das wäre eine Verhaltensweise, wie sie in der realen Welt ohne die Zuhilfenahme berauschender Mittel sicher kaum zu beobachten ist. Es fehlt also eine Möglichkeit, der Form der Landschaft zu folgen. Diese deswegen auch Terrain Following (also  
6245 Landschaftsverfolgung) genannte Methode läßt sich mittels Picking ebenfalls recht einfach realisieren.



Dazu ist es lediglich nötig, aus einer gewissen Höhe von der Position der ViewPlatform  
 6250 aus ein nach unten gerichtetes PickShape-Objekt einzusetzen. Hier könnte es dann sogar  
 zweckmäßig sein, einen unendlichen PickShape wie z.B. einen PickRay zu verwenden, da  
 der Benutzer – abhängig von der Struktur der Landschaft - unter Umständen auch über  
 eine sehr tiefe Schlucht geraten könnte. Beim Terrain Following ist es ebenfalls  
 interessant zu wissen, wie groß der Abstand zu den Objekten ist, die den Untergrund  
 6255 bilden. Dazu kann wieder die bereits bekannte Shape3D-Methode `intersect()` benutzt  
 werden. Wird der Abstand für den nächsten Schritt kleiner, so ist die Höhe der  
 ViewPlatform (also die Y-Position) entsprechend zu vergrößern, da der Untergrund  
 offensichtlich ansteigt. Wird der Abstand nach unten hin größer, jedoch nicht so groß,  
 dass das nicht noch mit einem Schritt der simulierten Person zu bewältigen wäre, so fällt  
 6260 das Gelände offenbar ab, und die Y-Position der ViewPlatform ist entsprechend nach  
 unten zu verändern. Ist der Abstand jedoch plötzlich deutlich größer, so könnte es sein,  
 dass sich der User tatsächlich gerade über einer (virtuellen) Schlucht befindet. Dann  
 wären andere Maßnahmen zu ergreifen, denkbar wäre ein beschleunigter Fall nach unten,  
 wobei das Terrain Following für die Zeit dieses Absturzes dann eigentlich wieder wie eine  
 6265 Collision Prevention funktioniert, um nämlich den Zeitpunkt und die Position für den  
 Aufprall zu ermitteln.

## 10 Umweltbedingungen

- 6270 Nach dem in den vorangegangenen Kapiteln allerlei effektvolles und für die Details der Szene wichtiges beschrieben wurde, wird es langsam Zeit, sich im wahrsten Sinne des Wortes größeren Aufgaben zuzuwenden. Ein bisher eher vernachlässigter Aspekt der Java 3D Programmierung ist nämlich die Beeinflussung der globalen Umweltbedingungen in einer virtuellen Welt. Allerdings wurde ein dazu gehörender Teil mit der Beschreibung von AmbientLight und DirectionalLight bereits beschrieben. Ein anderer wichtiger Teil wird mit der Klasse BackgroundSound im kommenden Kapitel behandelt. Wie es sich am Namen dieses Nodes unschwer erkennen läßt, handelt es sich dabei um eine Möglichkeit, einer Szene ein Umgebungsgeräusch hinzuzufügen.
- 6275
- 6280 Der verbleibende aber nicht weniger interessante Rest an Einflußmöglichkeiten auf die virtuelle Umwelt findet sich nun in diesem Kapitel, das damit also gewissermaßen als Lumpensammler dient.

### 10.1 Hintergrund

- 6285 Ein Element der hier geschaffenen 3D-Welten sah bisher immer ziemlich langweilig aus: der Hintergrund. Er war in den vorhergehend verwendeten Beispielpogrammen immer nur in einem eintönigen, per Default voreingestellten schwarz zu sehen. Das soll jetzt geändert werden. Zuvor ein paar Überlegungen über die nötigen Ansichten, die der ungläubige Leser auch gerne in der realen Welt überprüfen kann, am besten auf einem großen, freien Feld.
- 6290
- Bewegt man sich in der Realität, so ändert sich die eigene Position im Bezug auf Objekte, die relativ nahe sind. Geht man auf diese zu, kommen sie zwangsläufig näher und erscheinen größer. Dieses Verhalten ist in der bisherigen 3D-Welt bereits realisiert und kann mit einem Programm, das den KeyNavigatorBehavior verwendet, überprüft werden.
- 6295
- Ganz anders sieht das in der Realität jedoch für sehr weit entfernte „Objekte“ wie den Horizont aus – der zusammen mit dem Himmel gewissermaßen den Hintergrund bildet. Egal ob man sich auf diesen zu bewegt oder von ihm weg, der Abstand scheint immer gleich zu bleiben (zu mindest so lange, wie keine technischen Hilfsmittel eingesetzt werden, die die Geschwindigkeit des Beobachters nennenswert erhöhen). Und genau so ein Verhalten wird nun auch für den Hintergrund der virtuellen Java 3D Welt benötigt.
- 6300
- 6305 Der erste Gedanke, der sicher aufkommt, wäre es, einfach eine wirklich sehr große Kugel zu erzeugen und diese innen mit zum Beispiel einer Wolkentextur zu versehen. Ein entsprechender Versuch dürfte jedoch kläglich scheitern, wenn die Kugel eine gewisse Größe überschreitet. Sie ist dann – aus gutem Grund - schlichtweg nicht mehr sichtbar. Die Ursache für dieses Problem ist die so genannte Back Clip Distance, also eine
- 6310 Entfernung, hinter der alle Objekte, die dort noch kommen würden, nicht mehr dargestellt werden. Der Sinn dahinter wird detailliert in einem der folgenden Kapitel erklärt, hier kurz nur so viel, dass diese Back Clip Distance auch deswegen notwendig ist, um auch bei

sehr komplexen Szenen mit sehr vielen Objekten eine brauchbare Performance zu gewährleisten. 3D-Objekte, die jenseits dieser Entfernung liegen, würden sowieso nur noch ganz klein dargestellt werden können, sie würden jedoch genau so viel Ressourcen benötigen, als wenn sie in voller Größe im Vordergrund sichtbar sind. Die Back Clip Distance verhindert dies, in dem diese weit entfernten Objekte aus der Darstellung in der Szene entfernt werden – und was von der Hardware nicht mehr gezeichnet werden muß, verbraucht auch keine Grafik-Rechenleistung mehr.

6320

Für die Darstellung des Hintergrundes muß es also einen anderen Weg geben.

### 10.1.1 Der Background – Farbe und Geometrie

6325 Und diesen Weg gibt es tatsächlich in Form des Background-Nodes. Wie der Name es bereits sagt, definiert er global für eine gesamte Szene das Aussehen des Hintergrundes. Dieser Node-Typ ist dabei der einzige, der wirklich für jede Position in einem virtuellen Universum die gleiche Auswirkung hat, da für diesen – im Gegensatz zu Ambient-/DirectionalLight oder Fog – keine begrenzten Einflussbereiche mittels der Methode `setInfluencingBounds()` definiert werden können.

6330

Ein Blick auf einige der Konstruktoren dieses Node-Typs zeigt, dass es mit ihm möglich ist, alle Varianten der Beeinflussung des Hintergrundes zu realisieren:

6335 `Background(BranchGroup branch)`

Der Parameter dieses Konstruktors mag verwirren, er bietet jedoch genau das, was mit den Überlegungen im vorhergehenden Abschnitt bereits angedacht wurde: Eine Möglichkeit, geometrische Formen (wie also eben jene Kugel, die innen texturiert sein könnte) als Hintergrund zu verwenden. Eigentlich ermöglicht der einzige Parameter vom Typ `BranchGroup` sogar noch mehr: er erlaubt es, ganze `SceneGraphen` als Hintergrund zu verwenden. Einziger Unterschied zu den herkömmlichen, bisher verwendeten `SceneGraphen` ist, dass sie in der Tiefe anders wirken. Diese werden immer so dargestellt, als ob sie unendlich weit weg wären oder anders gesagt, als ob sie auf das Innere einer sehr weit entfernten Kugel projiziert werden würden.

6345

Ein paar Einschränkungen existieren jedoch für `SceneGraphen`, die für den Hintergrund eingesetzt werden. Diese dürfen – aus eben dem Grund, weil ihre Darstellung immer so erfolgt, als seien sie maximal weit entfernt – lediglich Nodes vom Typ `Shape3D` (aber nicht `OrientedShape3D`), `Morph`, `Light` und `Fog` beinhalten.

6350

`Background(Color3f color)`

`Background(float r, float g, float b)`

6355 Eine andere Möglichkeit, den Hintergrund zu gestalten, wäre es, ihm einfach eine einheitliche Farbe zu geben. Das ist mit diesen Konstruktoren machbar. Die Farbe wird dabei mittels des Parameters `color` festgelegt oder aber mit Hilfe der drei einzelnen

Farbanteile *r*, *g* und *b* für rot, grün und blau.

```
Background(ImageComponent2D image)
```

6360

Der letzte hier beschriebene Konstruktor erwartet ein `ImageComponent2D`-Objekt als Parameter. Dieses ist bereits von den Texturen her bekannt, es legt also ein Bild fest, das für den Hintergrund verwendet werden soll. Das Verhalten des `Background`-Nodes mit einem statischen Bild ist anders, als beim Einsatz von geometrischen Formen wie der

6365

Kugel, die innen eine Textur haben können. Während sich letztere so verhält, wie man es aus der Realität gewohnt ist, d.h. Dass sich der sichtbare Teil des Hintergrundes verändert, wenn sich der Beobachter um seine Y-Achse dreht, bleibt ein solches Hintergrundbild immer an der gleichen Position, es erzeugt also keinen Effekt, wie er aus der Realität bekannt ist.

6370

Das so erzeugte `Background`-Objekt ist anschließend wie jeder andere Node auch in eine `BranchGroup` einzufügen, die live ist bzw. durch das spätere Hinzufügen zum aktuellen Universe live werden wird.

6375

```
BranchGroup getGeometry()
```

```
void setGeometry(BranchGroup branch)
```

Passend zum ersten der beschriebenen Konstruktoren ist es mit diesen Methoden möglich, den aktuellen SceneGraphen des `Background`-Objektes zu ermitteln bzw. diesen neu zu setzen.

6380

```
void getColor(Color3f color)
```

```
void setColor(Color3f color)
```

```
void setColor(float r, float g, float b)
```

6385

Diese Methoden korrespondieren mit dem Konstruktor, der eine gleichmäßige Hintergrundfarbe festlegt. Die erste Methode kopiert die Farbinformationen in das zu übergebende `Color3f`-Objekt während es mit den anderen beiden möglich ist, diese neu zu setzen.

```
ImageComponent2D getImage()
```

6390

```
void setImage(ImageComponent2D image)
```

Passend zum letzten der beschriebenen Konstruktoren ermöglichen es diese Methoden, das aktuelle `ImageComponent2D`-Objekt des Hintergrundes zu ermitteln bzw. ein neues zu setzen.

6395

```
int getImageScaleMode()
```

```
void setImageScaleMode(int imageScaleMode)
```

Es wird der aktuelle Modus ermittelt bzw. ein neuer gesetzt, der das Aussehen des statischen Hintergrundbildes beeinflusst. **SCALE\_FIT\_ALL** legt fest, dass das Bild so

6400 skaliert wird, das es sich über das komplette Fenster erstreckt. Weichen die  
 Seitenverhältnisse des Images von denen des Fensters ab, wird das Bild verzerrt, es ist  
 bei Verwendung dieses Modes also immer das komplette Hintergrundbild zu sehen.  
 Mit der Konstanten **SCALE\_FIT\_MAX** wird das Bild so skaliert, dass die  
 Seitenverhältnisse erhalten bleiben, aber trotzdem das gesamte Fenster ausgefüllt ist. Bei  
 6405 abweichenden Seitenverhältnissen werden Teile des Images dann in X- oder in Y-  
 Richtung abgeschnitten.  
 Das Gegenstück dazu ist **SCALE\_FIT\_MIN**, hier wird so auf das Fenster mit der 3D-  
 Darstellung skaliert, das immer das gesamte Bild unverzerrt zu sehen ist. Das kann also  
 unter Umständen dazu führen, dass nicht das gesamte Fenster abgedeckt ist.  
**SCALE\_NONE** unterbindet jegliche Größenveränderung am Hintergrundbild, während  
 6410 **SCALE\_NONE\_CENTER** dieses zusätzlich in der Mitte des Canvas3D zentriert.  
 Die verbleibende Konstante **SCALE\_REPEAT** sorgt wieder für eine vollständige  
 Abdeckung des Hintergrundes. Ist das gesetzte Image hierfür nicht groß genug, wird es in  
 horizontaler und / oder vertikaler Richtung nötigenfalls wiederholt dargestellt.

6415 Wie bereits erwähnt, handelt es sich beim Background um einen weiteren Abkömmling  
 der Klasse Node:

```

java.lang.Object
 javax.media.j3d.SceneGraphObject
6420 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Background

```

6425 Deswegen finden sich hier auch diverse Capabilities, die wieder gesetzt werden müssen,  
 wenn die korrespondierenden Eigenschaften nach dem Compilieren mittels `compile()`  
 der übergeordneten BranchGroup oder wenn diese live ist noch modifizierbar sein sollen.  
 Entsprechende Konstanten existieren für das Lesen und Setzen der Hintergrundfarbe  
 (**ALLOW\_COLOR\_READ** und **ALLOW\_COLOR\_WRITE**), das Holen und neu Definieren  
 der BranchGroup, die die geometrischen Informationen für den Hintergrund enthält  
 6430 (**ALLOW\_GEOMETRY\_READ** und **ALLOW\_GEOMETRY\_WRITE**), die get- und die set-  
 Methode für das ImageComponent2D-Objekt des statischen Hintergrundbildes  
 (**ALLOW\_IMAGE\_READ** und **ALLOW\_IMAGE\_SCALE\_MODE\_READ**) sowie für das  
 ändern des Skalierungsmodus (**ALLOW\_IMAGE\_SCALE\_MODE\_WRITE**) bzw. das  
 zugehörige statische Hintergrundbild selber (**ALLOW\_IMAGE\_WRITE**).

6435

### 10.1.2 Ein Sternenhimmel

Da es mit dem jetzigen Wissensstand ein leichtes ist, eine texturierte Sphere zu erzeugen  
 (bei der mittels der Flags **GENERATE\_NORMALS\_INWARD** und  
 6440 **GENERATE\_TEXTURE\_COORDS** dafür gesorgt wird, dass die Textur in ihrem Inneren  
 sichtbar wird), soll das Beispielpogramm eine etwas kompliziertere Geometrie für den  
 Hintergrund verwenden. Speziell dreht es dabei um ein häufig anzutreffendes  
 Gestaltungsmittel: einen Sternenhintergrund. Da für diesen in der hier gezeigten Lösung

keine Textur verwendet wird, ist es zusätzlich möglich durch Einsatz selbiger zusammen  
6445 mit einer Sphere zusätzlich z.B. Galaxien am Himmel darzustellen.

Um zu überprüfen, ob der Hintergrund sich auch wirklich so verhält wie erwartet, muß  
zuerst eine Navigationsmöglichkeit geschaffen werden:

```
6450 (1) public BranchGroup createSceneGraph()
 (2) {
 (3) BranchGroup RootBG=new BranchGroup();
 (4) TransformGroup ViewTG=new TransformGroup();
 (5) Appearance ConeAppearance=new Appearance();
6455 (6) DirectionalLight DLgt=new DirectionalLight(new
 Color3f(0.8f,0.8f,0.8f),new Vector3f(-0.5f,-1f,-0.5f));
 (7) AmbientLight ALgt=new AmbientLight(new Color3f(0.8f,0.8f,0.8f));
 (8) BoundingSphere BigBounds=new BoundingSphere(new Point3d(),100000);
 (9)
6460 (10) ALgt.setInfluencingBounds(BigBounds);
 (11) DLgt.setInfluencingBounds(BigBounds);
 (12) RootBG.addChild(ALgt);
 (13) RootBG.addChild(DLgt);
 (14) ConeAppearance.setMaterial(new Material(new Color3f(0.9f,0.5f,0.5f),new
6465 Color3f(0f,0f,0f),new Color3f(0.3f,0.7f,0.7f),new
 Color3f(0.8f,0.8f,0.8f),1f));
 (15) RootBG.addChild(new
 Cone(0.5f,1.5f,Cone.GENERATE_NORMALS,40,1,ConeAppearance));
 (16) ViewTG=u.getViewingPlatform().getViewPlatformTransform();
6470 (17) MouseRotate MouseCtrl=new MouseRotate(ViewTG);
 (18) MouseCtrl.setSchedulingBounds(BigBounds);
 (19) RootBG.addChild(MouseCtrl);
 (20) MouseZoom MouseCtrl2=new MouseZoom(ViewTG);
 (21) MouseCtrl2.setSchedulingBounds(BigBounds);
6475 (22) RootBG.addChild(MouseCtrl2);
 (23)
 (24) createStarBackground(RootBG);
 (25)
 (26) RootBG.compile();
6480 (27) return RootBG;
 (28) }
```

Das geschieht mit den bereits bekannten Behaviors MouseRotate (zur Veränderung der  
Lage bei gedrückter linker Maustaste) und MouseZoom (zur Veränderung der Position in  
6485 Z-Richtung bei gedrückter mittlerer Maustaste). Diese werden in den Zeilen 17 bis 22 in  
bekannter Weise erzeugt und dem SceneGraphen hinzugefügt, damit sie anschließend

live und damit aktiv werden können. Ein Unterschied zeigt sich jedoch in Zeile 16: Statt wie bisher die TransformGroup eines 3D-Objektes der Szene zu verändern, wird die TransformGroup der ViewPlatform verwendet. Das heißt, dass die Veränderungen durch die beiden Behaviors jetzt direkten Einfluss auf Position und Lage des Beobachters haben.

In Zeile 24 findet sich ein neuer Methodenaufruf. Innerhalb von createStarBackground() wird nun dafür gesorgt, dass einige Sterne am virtuellen Himmel zu sehen sind:

```
(1) void createStarBackground(BranchGroup bg)
(2) {
(3) java.util.Random rand = new java.util.Random();
6500 (4) float mag;
(5) BranchGroup BGBranch=new BranchGroup();
(6) Background BG=new Background();
(7)
(8) PointArray starfield = new PointArray(15000, PointArray.COORDINATES|
6505 PointArray.COLOR_3);
(9) float[] point = new float[3];
(10) float[] brightness = new float[3];
(11) for (int i = 0; i < 15000; i++)
(12) {
6510 (13) point[0] = (rand.nextInt(2) == 0) ? rand.nextFloat() * -1.0f
 :rand.nextFloat();
(14) point[1] = (rand.nextInt(2) == 0) ? rand.nextFloat() * -1.0f
 :rand.nextFloat();
(15) point[2] = (rand.nextInt(2) == 0) ? rand.nextFloat() * -1.0f
6515 :rand.nextFloat();
(16) starfield.setCoordinate(i, point);
(17) mag=(rand.nextFloat()+0.5f)/1.5f;
(18) brightness[0]=mag;
(19) brightness[1]=mag;
6520 (20) brightness[2]=mag;
(21) starfield.setColor(i, brightness);
(22) }
(23) Shape3D StarShape=new Shape3D(starfield);
(24) StarShape.setAppearance(new Appearance());
6525 (25) StarShape.getAppearance().setPointAttributes(new
 PointAttributes(1f,true));
(26) BGBranch.addChild(StarShape);
(27) BG.setGeometry(BGBranch);
(28) BG.setApplicationBounds(new BoundingSphere(new Point3d(),10.0));
6530 (29) bg.addChild(BG);
```



( 30 )     }

6535 In den Zeilen 3 bis 6 werden einige Variablen deklariert und die dazugehörigen Objekte auch gleich erzeugt. So auch das Background-Objekt, das hier mit einem weiteren Konstruktor angelegt wird, der keine Parameter erwartet. Das erfordert natürlich, das später mit einer der set-Methoden näheres spezifiziert wird.

6540 In den folgenden Codezeilen wird nun ein (interessanterweise dreidimensionaler) Sternenhimmel erzeugt, der in Form eines PointArrays bestehend aus 15000 Punkten einem Shape3D-Objekt zugeordnet werden soll. Dieses GeometryArray wird in Zeile 8 angelegt, wobei mit Hilfe der Flags **COORDINATES** und **COLOR\_3** angegeben wird, dass neben den Koordinaten der Punkte auch Farbinformationen für diese enthalten sein werden. In der folgenden for-Schleife, die für jeden Satz Koordinaten und Farbinformationen einmal durchlaufen wird, werden nun zum einen die Koordinaten jedes Punktes selbst als auch die Farbinformation erzeugt. Die X-, Y- und die Z-Koordinate eines jeden Sterns wird in den Zeilen 13 bis 15 per Zufallsfunktion ermittelt und mit Hilfe der Methode `setCoordinate()` für jeden Vertex einzeln dem PointArray hinzugefügt. Ebenso zufällig wird der Farbwert in den Zeilen 16 bis 20 ermittelt. Da die Sterne nur in den grauen bzw. weißen Farbtönen erscheinen sollen, werden alle drei Farbelemente, die für rot, grün und blau stehen auf den gleichen Zahlenwert gesetzt und mittels `setColor()` für jeden Vertex separat hinzugefügt. Der Konstrukt in Zeile 17 sorgt im Übrigen dafür, dass die Sterne eine Mindesthelligkeit haben. Da es Verschwendung wäre, Ressourcen für nicht sichtbare (weil schwarze Sterne) oder aber für nur sehr schwer zu erkennende, dunkelgraue Stern aufzuwenden, macht es Sinn, für deren Farbe einen Schwellwert festzulegen.

6555

6560 In den Zeilen 23 und 24 findet sich auch nichts wesentlich neues mehr, hier wird ein Shape3D-Objekt angelegt und diesem ein Appearance-Objekt zugeordnet. Da die gewünschte Appearance des Hintergrundes genau dem entspricht, was mit dem Defaultkonstruktor angelegt wird (also keine besonderen Materialeffekte, keine Beeinflussung durch Lichtquellen etc.) genügt hier diese recht direkte Vorgehensweise. In Zeile 25 schließlich werden lediglich die in dieser Appearance enthaltenen PointAttributes modifiziert. Da es bei der Verwendung des PointArrays auf manchem Plattformen (speziell unter Windows und DirectX) unter Umständen schwer reproduzierbare Effekte geben kann, empfiehlt es sich, im Falle, dass Geometriedaten in Form eines PointArrays zum Einsatz kommen, immer explizit ein PointAttributes-Objekt anzulegen.

6565

6570 Ab Zeile 27 wird es dann erstmalig richtig spannend: Das Shape3D-Objekt, das die Geometriedaten des dreidimensionalen Sternfeldes enthält, wird einer BranchGroup hinzugefügt, die gleich anschließend dem Background-Objekt mittels `setGeometry()` übergeben wird. Damit wird festgelegt, dass dieses Sternfeld für den Hintergrund verwendet werden soll, wo es damit zum Sternenhimmel wird. Abschließend wird der Background-Node nur noch der BranchGroup hinzugefügt, die der Methode `createStarBackground()` als Parameter übergeben wurde und welche letztendlich im Universum live werden wird.

6575

Nach dem Compilieren und Ausführen dieses Programmes zeigt sich der gewünschte

Effekt. Navigiert man mit der Maus (bei gedrückter linker bzw. mittlerer Maustaste) ein wenig durch die Szene, so zeigt sich für den Hintergrund exakt das gewollte Verhalten.  
6580 Allerdings sollte man sich nicht vom dargestellten Kegel irritieren lassen: nicht dieser bewegt sich, sondern die ViewPlatform. Wer das nicht glaubt, möge einen Blick in den Code werfen: der Kegel ist direkt mit der obersten BranchGroup an das Universum gebunden, es findet sich keine TransformGroup, die dessen Position verändern könnte.

### 6585 10.1.2.1 PointAttributes

Mit diesem Beispielprogramm wurde ebenfalls ein neues Attributes-Objekt für die Klasse Appearance vorgestellt, die PointAttributes. Da diese nur dann sinnvoll verwendet werden können, wenn ein PointArray (oder natürlich auch ein IndexedPointArray) für die  
6590 Geometriedaten des zugehörigen Shape3D-Objektes zum Einsatz kommt, sollen sie hier in diesem Zusammenhang beschrieben werden. Neben dem Defaultkonstruktor, der keine Parameter erwartet, ist dieser sicher der interessantere:

```
PointAttributes(float pointSize, boolean pointAntialiasing)
```

6595

Er erzeugt ein neues PointAttributes-Objekt, das mit dem ersten Parameter `pointSize` festlegt, wie groß die dargestellten Punkte sein sollen. Da diese bei Werten größer 1 je nach Anwendungszweck ziemlich unschön aussehen können, ist es möglich, sie durch Setzen des zweiten Parameters auf `true` bei der Darstellung glätten zu lassen. Dieses  
6600 Antialiasing kann sich aber auch für kleinere Punkte lohnen. Hier kann es z.B. passieren, dass sich ein solcher Punkt nicht direkt 1:1 einem Pixel des Canvas3D zuordnen läßt. Die Darstellung muß dann auf mehrere Pixel verteilt werden, was bei aktiviertem Antialiasing deutlich besser aussieht.

```
6605 boolean getPointAntialiasingEnable()
void setPointAntialiasingEnable(boolean state)
```

Diese Methoden ermöglichen es, zu ermitteln, ob dieses eben beschriebene Antialiasing aktiviert ist bzw. erlauben es, dieses zu aktivieren (`state=true`) oder zu deaktivieren (`state=false`)

6610

```
float getPointSize()
void setPointSize(float pointSize)
```

Ähnliches ist für die Größe der Punkte möglich: Mittels dieser beiden Methoden kann der aktuelle Wert ermittelt bzw. ein neuer Wert gesetzt werden.

6615

Die PointAttributes leiten sich ähnlich ab wie die anderen Attributes-Klassen:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
```

6620                   javafx.media.j3d.NodeComponent  
                      **javafx.media.j3d.PointAttributes**

Das Verhalten bei den Capabilities zeigt bei den PointAttributes genau so große Ähnlichkeiten. Die kann mit dem alt bekannten Aufruf der Methoden `setCapability()` und/oder `setCapabilityIsFrequent()` festgelegt werden, ob der Antialiasing-Modus nach dem Compilieren noch gelesen (**ALLOW\_ANTIALIASING\_READ**) oder verändert werden darf (**ALLOW\_ANTIALIASING\_WRITE**). Gleiches gilt für die Größe der Punkte und damit für die Verwendbarkeit der Methoden `getPointSize()` (**ALLOW\_SIZE\_READ**) oder `setPointSize()` (**ALLOW\_SIZE\_WRITE**) nach dem Aufruf von `compile()` in der übergeordneten BranchGroup oder aber wenn diese live ist.

## 10.2 Nebel

Ein Gestaltungselement, mit dem sich besonders stimmungsvolle Szenen erzeugen lassen, ist Nebel. Auch hier bietet Java 3D interessante Möglichkeiten, einen solchen zu erzeugen. Genauer gesagt, gibt es mit `LinearFog` und `ExponentialFog` gleich zwei Klassen für Nebel, die sich beide von der abstrakten Basisklasse `Fog` ableiten. Zu diesem Nebel sei gesagt, dass es sich dabei nicht um einen volumetrischen Nebel handelt, also einen Nebel, wie er aus der Realität bekannt ist. Bei diesem würde ein Raumbereich mit Partikeln gefüllt sein, der die Sicht mit zunehmender Entfernung verschlechtert. Java 3D bzw. die darunterliegende Grafikhardware und -schnittstelle arbeiten anders: Hier werden die 3D-Objekte mit zunehmender Entfernung einfach mehr und mehr mit der Farbe des Nebels überlagert. Das ist keine schlechte Methode, um Nebel zu simulieren, sie erfordert es lediglich, dass der Hintergrund die gleiche Farbe hat, wie der Nebel, damit der Effekt möglichst echt wirkt.

Wie das aussehen würde, wenn Die Hintergrundfarbe und die Farbe des Nebels nicht zusammenpassen, können Sie durch die Veränderung der Hintergrundfarbe im kommenden Beispielpogramm gerne einmal ausprobieren. Es wäre sogar vorteilhaft, einmal zu sehen, wie so etwas aussieht, damit es Ihnen später einmal möglich ist, den sich dadurch ergebenden Effekt einzuordnen und die Fehlerursache zu finden, wenn die Farbe von Nebel und Hintergrund irrtümlicherweise einmal nicht zusammenpassen.

Ein mit einem der Fog-Nodes erzeugter Nebel wird innerhalb einer virtuellen Welt immer dann sichtbar, wenn

1. ein Einflußbereich mittels der Methode `setInfluencingBounds()` definiert wurde.
2. der Fog-Node live ist (er also zum Universum hinzugefügt wurde)

Da `Fog` eine abstrakte Klasse ist, finden sich hier keine Konstruktoren, die benutzbar wären, jedoch enthält sie einige Methoden, die zwangsläufig von den abgeleiteten Klassen geerbt werden:

```
void addScope(Group scope)
```

6665 Der Scope ist eine interessante Funktionalität der Java 3D Fogs und ist vergleichbar mit dem bei der Klasse Light verwendeten Scope. Wird mit dieser Methode eine (Branch)Group übergeben, so wirkt der Nebel ausschließlich auf die Nodes unterhalb dieser Group. Wird für ein Fog-Objekt jedoch kein Scope gesetzt, so wirkt der Nebel global, d.h. innerhalb des ganzen Universums (oder exakter innerhalb des Universums im Rahmen der für den Nebel festgelegten Influencing Bounds).

6670 Normalerweise wird hier sicher immer ein BranchGroup-Objekt übergeben, da dieses in der Lage ist, auf weitere Groups und Nodes zu verzweigen. Da der geforderte Parameter jedoch ein Objekt vom übergeordneten Typ Group ist, ist es natürlich auch möglich, ganz andere Group-Objekte wie eine TransformGroup, einen Switch, eine SharedGroup oder anderes zu übergeben. Einschränkungen gelten beim Hinzufügen einer neuen Group  
6675 lediglich in Bezug auf deren Status. Ist sie nämlich schon Teil eines SceneGraphen, der live ist, so hat das eine `RestrictedAccessException` zur Folge. In diesem Fall ist es also immer erforderlich, zuerst die Group aus dem Universum zu entfernen, damit sie nicht mehr live ist, sie dann mittels einer der set-Methoden als Scope zu definieren um sie anschließend gegebenenfalls wieder zum aktuellen Universum hinzuzufügen.

6680

```
Group getScope(int index)
```

Mit dieser Methode ist es möglich, diejenige Group zu ermitteln, die für den Scope an Position `index` der klasseninternen Scopeliste gesetzt wurde.

6685 `java.util.Enumeration` getAllScopes()

Hiermit werden alle Scopes in Form einer Enumeration zurückgeliefert, die für dieses Fog-Objekt gelten.

```
int indexOfScope(Group scope)
```

6690 Diese Methode ist das Gegenstück zur Vorangegangenen: sie liefert den Indexwert für die übergebene Group, d.h. sie sagt aus, an welcher Position das Group-Objekt in der internen Scopeliste eingeordnet wurde. Kann das übergebene Objekt `scope` in dieser nicht gefunden werden, so ist der Rückgabewert -1.

6695 `void` insertScope(Group scope, int index)

Es wird ein Group-Objekt an der Position `index` in die Liste der Fog-intern verwalteten Group-Objekte eingefügt, die für dieses Fog-Objekt angeben, auf welche Nodes und 3D-Objekte es wirken soll.

6700 `int` numScopes()

Diese Methode liefert die Gesamtzahl von Group-Objekten zurück, die aktuell für dieses Fog-Objekt definieren, auf welche 3D-Objekte der Nebel wirken soll. Ist der Rückgabewert 0, so heißt das, dass kein spezifischer Scope definiert wurde und der Nebel somit global auf alle 3D-Objekte des Universums innerhalb der spezifizierten Influencing  
6705 Bounds wirkt.

```
void removeAllScopes()
```

Es werden alle Group-Objekte entfernt, so dass anschließend kein spezieller Scope mehr definiert ist und das Fog-Objekt somit wieder auf alle 3D-Objekte des zugehörigen Universe wirkt.

```
void removeScope(Group scope)
```

```
void removeScope(int index)
```

Diese Methoden entfernen jeweils ein Group-Objekt aus der Liste der Scopes. Welche Group entfernt werden soll, wird dabei durch den einzigen zu übergebenden Parameter festgelegt, der entweder das Group-Objekt selbst spezifiziert, das entfernt werden soll, oder aber die Position `index`, an der sich die zu entfernende Group aktuell befindet.

```
void getColor(Color3f color)
```

```
void setColor(Color3f color)
```

```
void setColor(float r, float g, float b)
```

Diese Methoden erlauben es, den aktuellen Wert für die Farbe des Nebels zu ermitteln bzw. ihn neu zu setzen. Bei der Verwendung von Color3f-Objekten als Parameter wird dabei verfahren, wie auch schon von anderen Java3D-Klassen her bekannt: Die Farbinformationen werden aus diesem bzw. in dieses Color3f-Objekt kopiert, es wird also nicht als Referenz verwendet.

```
Bounds getInfluencingBounds()
```

```
void setInfluencingBounds(Bounds region)
```

Auch für die Klasse Fog ist es notwendig, einen Einflußbereich festzulegen, der mit diesen Klassen ermittelt bzw. geholt werden kann. Dieser funktioniert wie bereits von den Lichtobjekten her bekannt: Nur innerhalb dieses Einflußbereiches ist der Nebel wirksam. Für den Fall der Klasse Fog wird dieser Einflußbereich zusätzlich zu einem eventuell gesetzten Scope verwendet.

Wie sich zeigt, gehört die Klasse Fog ebenfalls zu den Nodes:

```
java.lang.Object
```

```
6740 javax.media.j3d.SceneGraphObject
```

```
 javax.media.j3d.Node
```

```
 javax.media.j3d.Leaf
```

```
 javax.media.j3d.Fog
```

Und als Abkömmling der Klasse Node bzw. SceneGraphObject kennt Fog nicht nur die von dort geerbten Capability-Bits, sondern definiert auch eigene. Diese beeinflussen die Nutzbarkeit der verschiedenen Fog-Eigenschaften wie dessen Farbe (mittels

**ALLOW\_COLOR\_READ** und **ALLOW\_COLOR\_WRITE**), den Influencing Bounds (über **ALLOW\_INFLUENCING\_BOUNDS\_READ** und **ALLOW\_INFLUENCING\_BOUNDS\_WRITE**) sowie die Möglichkeit, die für den Scope verwendeten Groups zu ermitteln (**ALLOW\_SCOPE\_READ**) bzw. den Scope zu verändern (**ALLOW\_SCOPE\_WRITE**).

### 10.2.1 ExponentialFog

6755

Nach so viel grauer Theorie soll nun ein wenig Licht in das Dunkel der praktischen Anwendung von Fog-Nodes gebracht werden – auch wenn das paradox scheint, da der Nebel-Effekt ja eher für eine Verschleierung sorgt. Es soll hier mit der Klasse `ExponentialFog` begonnen werden. Diese bietet zum einen einen etwas leichteren

6760 Einstieg und zum anderen einen realistischer wirkenden Nebel-Effekt. Wie es der Name bereits andeutet, ist es mit der Klasse `ExponentialFog` möglich, einen Nebel zu erzeugen, der mit wachsender Entfernung zum Beobachter exponentiell zunimmt.

An dieser Stelle ist es wieder einmal notwendig, das Beispielprogramm abzuändern. Zum

6765 einen wird dem Kegel aus den vorangegangenen Abschnitten eine andere Farbe gegeben, damit er sich deutlicher vom Nebel abhebt, welcher in klassischem weiß dargestellt werden soll (die Darstellung giftig-gelben Nebels aus Industrieabgasen oder anderer ungesunder Variationen soll an dieser Stelle vermieden werden). Die Materialeigenschaften der `ConeAppearance` werden deswegen so geändert, dass das

6770 Ergebnis ein blauer Kegel ist:

```
ConeAppearance.setMaterial(new Material(new
Color3f(0.1f,0.1f,1.0f),new Color3f(0f,0f,0f),new
Color3f(0.4f,0.4f,1.0f),new Color3f(0.5f,0.9f,0.9f),100f));
```

6775

Erhalten bleiben die Navigationsmöglichkeiten mit der Maus, speziell der `MouseZoom Behavior`, der es ermöglichen wird, den mit wachsender Entfernung zunehmenden Nebel zu sehen. Die neuen Elemente befinden sich jetzt in einer Methode `createEnvironment()`, die aus `createSceneGraph()` heraus unter Angabe des

6780 `Root-BranchGroup`-Objektes aufgerufen werden soll:

```
(1)void createEnvironment(BranchGroup bg)
(2) {
(3) Background BG=new Background();
6785 (4) ExponentialFog Nebel;
(5)
(6) BG.setColor(new Color3f(1.0f,1.0f,1.0f));
(7) BG.setApplicationBounds(new BoundingSphere(new Point3d(),10.0));
(8) bg.addChild(BG);
6790 (9)
(10) Nebel=new ExponentialFog(new Color3f(1.0f,1.0f,1.0f),10.0f);
```

```

(11) Nebel.setInfluencingBounds(BigBounds);
(12) bg.addChild(Nebel);
(13) }

```

6795

Nach der Deklaration einiger Variablen, bei der in Zeile 3 auch gleich ein Default-Background-Objekt angelegt wird, ist es der nächste Schritt, dass der Hintergrund selbigens einförmig weiß wird, damit er farblich zum Nebel paßt. Das passiert in Zeile 6 unter Zuhilfenahme der bereits bekannten Methode `setColor()`. Die Erzeugung des Hintergrundes geht hier also wesentlich einfacher vom stattdessen, mit dem Hinzufügen des Background-Objektes zur übergebenen BranchGroup `bg`, die später live werden und damit den Hintergrund anzeigen wird, ist dafür bereits alles erledigt. Anschließend wird in Zeile 10 das Fog-Objekt konstruiert:

6800

```

6805 ExponentialFog(Color3f color, float density)

```

Der erste Parameter `color` dürfte klar sein, er legt fest, welche Farbe der Nebel haben soll. Mit dem Zweiten ist es möglich, die Formel zu beeinflussen, nach der der Nebel berechnet wird. Kurz gesagt, bedeutet das, um so größer der Wert ist, der für `density` gesetzt wird, um so dicker wird der Nebel. Etwas ausführlicher läßt sich das in Form der Formel ausdrücken, mit deren Hilfe der Faktor `f` berechnet wird, der angibt wie stark ein 3D-Objekt abhängig von der Dichte **density** und der Entfernung `z` zum Beobachter zunehmen soll:

6810

6815 Wie bereits erwähnt, sind auch für den Nebel Influencing Bounds anzugeben, was in Zeile 11 passiert. In Zeile 12 wird dann noch die letzte Bedingung erfüllt, um den Nebel aktiv werden zu lassen – das ExponentialFog-Objekt wird einer BranchGroup hinzugefügt, die später live werden wird.

6820 Nach dem Compilieren und Ausführen des Beispielprogrammes kann man ein wenig mit dem Nebel spielen. Wird die Maus bei gedrückter mittlerer Maustaste bewegt, so nähert bzw. entfernt man sich dank des MouseZoom Behaviors vom dargestellten Kegel und sieht diesen auf diese Art aus dem Nebel auftauchen bzw. ihn im Nebel verschwinden. An dieser Stelle kann dann auch ein wenig mit den Möglichkeiten des `density`-Wertes und  
6825 speziell auch mit der Variante einer anderen, nicht zum Nebel passenden Hintergrundfarbe experimentiert werden.

Da die Dichte des Nebels eine spezifische Eigenschaft der Klasse ExponentialFog ist, fügt diese den geerbten Methoden noch folgendes hinzu:

6830

```

float getDensity()
void setDensity(float density)

```

Diese beiden Methoden erlauben es, den aktuellen Wert für die Dichte des Nebels zu ermitteln bzw. ihn auf einen neuen Wert zu setzen.

6835

Wie bereits erwähnt, ist diese Klasse ein direkter Abkömmling von Fog:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
6840 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Fog
 javax.media.j3d.ExponentialFog
```

6845 Ähnlich wie bereits bei den Methoden gesehen, erweitert diese Klasse die Capabilities um die Möglichkeit, das Lesen des aktuellen Wertes für die Dichte des Nebels (**ALLOW\_DENSITY\_READ**) und das Setzen eines neuen Density-Wertes (**ALLOW\_DENSITY\_WRITE**) auch nach dem Compilieren eines übergeordneten BranchGroup-Objektes oder wenn dieses live geworden ist, weiterhin zu erlauben.

6850

### 10.2.2 LinearFog

Der zweite mögliche Fog-Node simuliert eine andere Art von Nebel. Wie es der Name bereits verrät, handelt es sich dabei um einen linearen Nebel, der demzufolge also direkt  
6855 proportional zur Entfernung zunimmt. Deswegen kommt hier auch kein Density-Wert zum Einsatz, der festlegt, wie dicht der Nebel sein soll. Vielmehr wird für diesen angegeben, bei welcher Entfernung der Nebel einsetzt und ab welcher Entfernung er die 3D-Objekte vollständig zu bedecken hat. Einer der möglichen Konstruktoren erwartet dementsprechend auch zwei Werte für die Entfernung:

6860

```
LinearFog(Color3f color,double frontDistance,double backDistance)
```

Der erste hier zu übergebende Parameter ist bereits bekannt, er legt wieder die Farbe des Nebels fest. Die beiden folgenden spezifizieren dann die nötigen Entfernungswerte, wobei  
6865 `frontDistance` angibt, ab welcher Entfernung vom Beobachter der Nebel einsetzen soll zu wirken, `backDistance` sagt dementsprechend aus, ab welcher Entfernung dann von den 3D-Objekten der Szene nichts mehr zu sehen sein soll. Sinnvollerweise sollte für einen realistischen Nebeleffekt der Wert für die `frontDistance` nicht all zu groß und `backDistance` hingegen deutlich größer als `frontDistance` sein. Der Faktor `f`, der die  
6870 Stärke des Nebels angibt, berechnet sich im Falle dieser Klasse abhängig von den beiden Entfernungswerten übrigens folgendermaßen:

$$f = \frac{\text{backDistance} - z}{\text{backDistance} - \text{frontDistance}}$$

6875 Die Basisklasse Fog wird bei den Methoden ebenfalls um die für den LinearFog-Node



notwendigen Entfernungswerte erweitert:

```
double getBackDistance()
void setBackDistance(double backDistance)
```

6880        So ermöglichen es diese Methoden, den Wert für die Entfernung, ab der der Nebel die 3D-Objekte vollständig verdecken soll zu ermitteln bzw. ihn neu zu setzen.

```
double getFrontDistance()
void setFrontDistance(double frontDistance)
```

6885        Dementsprechend gibt es auch Methoden, die den Anfangs-Abstand für den Nebel ermitteln bzw. ihn neu setzen.

Die Ableitung von der Klasse Fog liefert dieses Bild, das eigentlich auch zu erwarten war:

```
6890 java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Fog
6895 javax.media.j3d.LinearFog
```

Bei den Capabilities werden fast zwangsläufig wieder zusätzliche Werte benötigt, die sich auf die set- und get-Funktionen für die FrontDistance, also die Entfernung, ab der der Nebel einsetzt und die BackDistance und somit auf den Punkt, ab dem der Nebel alles

6900 **ALLOW\_DISTANCE\_READ** für das Lesen der Entfernungswerte und  
**ALLOW\_DISTANCE\_WRITE** für das Schreiben.

## 11 Klangvolles – Sound in der virtuellen Welt

6905

Ein wichtiges Element besonders in Virtual Reality Simulationen oder in Spielen sind Soundeffekte. Die Anwendungsmöglichkeiten hierfür sind vielfältig, sie beginnen bei einem einfachen Hintergrundgeräusch, gehen über Geräusche, die von ganz alltäglichen Objekten wie von Springbrunnen oder Bächen ausgehen, und enden noch lange nicht bei Motorengeräuschen von Autos oder gar dem Phaserfeuer von Raumschiffen (auch wenn letzteres im luftleeren Raum realistischerweise eigentlich nicht zu hören sein dürfte ist es nach wie vor sehr beliebt und wirkt leider auch recht interessant, wenn dieser falsche Effekt zum Einsatz kommt).

6910

6915

Da in diesem Kapitel das Hauptaugenmerk eben nicht auf das in einer virtuellen Java 3D Welt Sichtbare liegt, ist das eine gute Gelegenheit, um ein wenig mehr mit den grafischen Möglichkeiten zu spielen, die in den vorangegangenen Kapiteln beschrieben wurden. Schließlich lenken diese jetzt nicht vom eigentlichen Objekt der Betrachtungen ab.

6920

Wirft man auf der Suche nach Möglichkeiten, einer Java 3D Szene Soundeffekte hinzuzufügen einen Blick in die offizielle Dokumentation, so findet sich ein Node-Typ, der hierfür gut geeignet zu sein scheint: die Klasse Sound.

### 11.1 Die Verwendung der Basisklasse Sound

6925

Sound ist wiederum nur eine abstrakte Basisklasse, die sich deswegen nicht direkt sondern nur in Form der von ihr abgeleiteten Klassen verwenden läßt. Da die Klasse Sound jedoch bereits viele sehr wesentliche Eigenschaften und Funktionalitäten beinhaltet, ist es durchaus sinnvoll, nicht nur diese Klasse sondern ausnahmsweise auch einmal einige ihrer eben so abstrakten und deswegen nicht direkt verwendbaren Konstruktoren zu betrachten:

6930

```
Sound(MediaContainer soundData,float initialGain,int
loopCount,boolean release,boolean continuous,boolean enable,Bounds
region,float priority)
```

6935

Hier werden offenbar schon sehr viele Werte eingestellt, allerdings fällt auf, dass 1. kein Pfad, URL oder ähnliches zu der abzuspielenden Sounddatei übergeben wird 2. aber statt dessen ein Parameter vom Typ MediaContainer erwartet wird. In der Tat ist es so, dass sich dieser MediaContainer um die Sounddaten kümmert.

6940

#### 11.1.1 MediaContainer

Dieser MediaContainer ist wie bereits erwähnt unter anderem für die Quelle zuständig, aus der der Sound bezogen werden soll. Des weiteren bearbeitet er die Sounddaten (oder

6945

genauer die Audio-Fileformate) selber und er steuert zusätzlich auch ein eventuell notwendiges Caching. Wozu dieses notwendig sein kann, zeigt sich bei den möglichen Quellen, von denen der MediaContainer die Sounddaten holen kann. Unter anderem kann die Angabe über den Speicherort der Sounddatei auch mit Hilfe einer URL gemacht werden. Das heißt, der MediaContainer lädt die Sounddaten in diesem Fall auch von irgend wo aus dem Netzwerk – und abhängig von der Verbindungsqualität und -geschwindigkeit kann hier ein zusätzliches Caching durchaus sinnvoll sein, um Aussetzer im Sound zu vermeiden.

Die verschiedenen möglichen Konstruktoren gewähren denn auch einen Einblick in die Möglichkeiten, die es für mögliche Herkunftsorte der Sounddaten gibt:

```
MediaContainer(java.io.InputStream stream)
```

Dieser Konstruktor ist interessant, da er offenbar einen InputStream erwartet, der auf beispielsweise eine bereits geöffnete Datei oder auf eine bestehende Remote-Verbindung zum Soundfile verweist. Bei näherer Betrachtung erschließt sich aber schnell der Gedanke dahinter: Da Sounddateien unter Umständen auch sehr groß werden können, wäre es absolut nicht zweckmäßig, diese vollständig und auf einmal zu übergeben, also beispielsweise innerhalb eines dann eventuell sehr großen Bytearrays. Der hier beschrittene Weg ist wesentlich effizienter: Es wird nur ein Stream übergeben aus dem sich der MediaContainer die benötigten Daten selbständig und Stück für Stück nacheinander immer dann abholt, wenn er sie benötigt.

```
MediaContainer(java.net.URL url)
```

Die Existenz dieser Möglichkeit wurde bereits erwähnt: es wird ein URL-Objekt übergeben, das festlegt, von welcher (Internet-)Adresse die Sounddaten heruntergeladen werden sollen.

6975

```
MediaContainer(java.lang.String path)
```

Bei diesem Konstruktor ist darauf zu achten, dass hier zwar kein URL-Objekt als Parameter erwartet wird, sondern ein URL in Form eines Strings. Das heißt für Dateien, die lokal gespeichert sind, dass diese beispielsweise mittels „file:/home/user/sound.au“ geladen werden müssen, die reine Pfadangabe „/home/user/sound.au“ funktioniert nicht und würde dazu führen, dass eine `SoundException` geworfen wird.

Bei den Methoden, die von der Klasse MediaContainer zur Verfügung gestellt werden, finden sich unter anderem auch Möglichkeiten, die sich auf das bereits erwähnte Caching der Daten, also deren Zwischenspeicherung beziehen. Auch ist es mit Hilfe verschiedener Methoden möglich, Objekte zu setzen, die wieder angeben, wo die Sounddaten her bezogen werden sollen. Dabei ist zu beachten, dass immer nur einer der möglichen Objekttypen InputStream, URL oder String verwendet werden darf, werden diese durch den Aufruf mehrerer unterschiedlicher set-Methoden oder aber durch die Verwendung

einer set-Methode, deren als Parameter übergebenes Objekt nicht zu dem des verwendeten Konstruktors passt, so wird eine `IllegalArgumentException` geworfen.

```
boolean getCacheEnable()
```

```
6995 void setCacheEnable(boolean flag)
```

Diese Methoden ermöglichen es, den Status des internen Caching-Flags abzufragen oder auf einen anderen Wert zu setzen. Ist das Caching aktiviert (das Caching-Flag also `true`) so werden die Sounddaten zusätzlich zwischengespeichert um z.B. möglicherweise unterschiedlich hohe Übertragungsraten beim Datendownload und damit Unterbrechungen beim Abspielen des Sounds zu vermeiden.

```
7000
```

```
java.io.InputStream getInputStream()
```

```
void setInputStream(java.io.InputStream stream)
```

```
7005
```

Wird ein `InputStream`-Objekt verwendet, um auf die Sounddaten zu verweisen, so ist es mit Hilfe dieser Methoden möglich, den aktuell verwendeten `InputStream` zu ermitteln oder einen neuen festzulegen.

```
java.net.URL getURLObject()
```

```
void setURLObject(java.net.URL url)
```

```
7010
```

Kommt statt dessen ein `URL` in Form der gleichnamigen Objekttyps zum Einsatz, so bieten diese beiden Methoden die Möglichkeit, das aktuelle Objekt zu ermitteln oder ein neues festzulegen.

```
java.lang.String getURLString()
```

```
7015
```

```
void setURLString(java.lang.String path)
```

Ähnlich verhält es sich mit diesen Methoden. Sie erlauben es den aktuellen `URL` zu holen bzw. einen neuen zu setzen, wenn dieser in Form eines Strings verwendet wird.

```
7020
```

Bei der Verwandtschaft mit anderen Klassen zeigt sich für den `MediaContainer` folgendes Bild:

```
java.lang.Object
```

```
 javax.media.j3d.SceneGraphObject
```

```
 javax.media.j3d.NodeComponent
```

```
7025
```

```
 javax.media.j3d.MediaContainer
```

```
7030
```

Interessantes findet sich hingegen wieder bei den Capabilities, die auch für Objekte dieser Klasse gesetzt werden müssen, wenn die zugehörige Eigenschaft auch innerhalb von SceneGraphen, die live oder kompiliert sind, modifizierbar sein soll. So existiert die Möglichkeit, die verschiedenen get-Methoden für den `URL` (bzw. `InputStream`) weiterhin zu verwenden, wenn das Capability Bit **ALLOW\_URL\_READ** gesetzt wurde. Gleiches gilt

für **ALLOW\_URL\_WRITE**, allerdings bezieht sich das auf die set-Methoden. Das Ermitteln des Caching-Status und das Setzen des selben läßt sich mit separaten Capability Bits erlauben, hier sind dann die Konstanten **ALLOW\_CACHE\_READ** und **ALLOW\_CACHE\_WRITE** einzusetzen.

### 11.1.2 Sound

Doch nun zurück zur eigentlichen Beschreibung der Klasse Sound, die aus aktuellem Anlaß so jäh unterbrochen wurde. Nach dem klar ist, wo die Sounddaten herkommen, soll bei den Details zur Basisklasse jeglichen Sounds fortgesetzt werden.

```
Sound(MediaContainer soundData,float initialGain,int
loopCount,boolean release,boolean continuous,boolean enable,Bounds
region,float priority)
```

Der Parameter `soundData` wäre an dieser Stelle also ausführlich beschrieben. `initialGain` legt die Gesamtlautstärke des Sounds fest, es ist damit also möglich, beispielsweise Hintergrundgeräusche deutlich leiser zu gestalten, als andere Soundeffekte. Ein Wert von 1.0 entspricht dabei 100% der Lautstärke. Mittels `loopCount` kann festgelegt werden, ob und wie oft der Sound wiederholt werden soll. Wird hier -1 angegeben, so steht das für eine endlose Schleife.

Mittels des Parameters `release` wird angegeben, welches Verhalten der Sound-Node an den Tag legen soll, wenn er deaktiviert wird. Wird hier `true` angegeben, so führt das dazu, dass der Sound auf jeden Fall erst zu Ende gespielt wird, bevor die Stummschaltung bzw. Deaktivierung erfolgt. Im Falle von `false` hingegen wird das Abspielen des Sounds sofort und damit unter Umständen sehr abrupt abgebrochen.

Mit `continuous` wird wiederum festgelegt, wie mit den Sounddaten verfahren werden soll, während der Sound deaktiviert wurde, also ob er unhörbar weiter abgespielt werden soll oder nicht. Im Falle von `true`, passiert genau das, was bedeutet, das bei einem späteren erneuten Aktivieren des Sounds an der Stelle fortgesetzt werden würde, an der sich der Sound auch beim hörbaren Abspielen befunden hätte. Wird statt dessen für `continuous false` angegeben, so wird beim erneuten Aktivieren eines Soundnodes der Sound auf jeden Fall immer von Anfang an abgespielt.

Der Parameter `enable` wiederum legt fest, ob dieser Sound-Node aktiviert werden soll oder nicht. Auch der vorletzte Parameter sollte bekannt sein, er definiert wieder einen Einflußbereich, innerhalb dessen der Sound hörbar sein soll. Also gilt auch hier: Nur wenn sich die ViewPlatform innerhalb der hier anzugebenden Bounds befindet, kann der User den zugehörigen Sound hören. Auch an dieser Stelle hat das – wie schon bei den Lights und bei den Behaviors – Performancegründe.

Der letzte Parameter beruht wieder auf einer Sound-spezifischen Eigenschaft bzw. auf

Einschränkungen, die von der zugrundeliegenden Audiohardware und ihrer Treiber herrühren. So ist es nicht möglich, beliebig viele Sounds gleichzeitig abzuspielen. Eine gewisse Vorauswahl wird dabei schon durch die Scheduling Bounds (den Einflußbereich) getroffen (so diese auf sinnvolle und nicht wie in den Beispielprogrammen zu sehen utopisch hohe Werte gesetzt werden). Passiert es nun aber doch, dass für das Abspielen der verbleibenden Sound mehr Kanäle benötigt werden, als für das aktuell verwendete Audio-Device verfügbar sind, so gibt es Schwierigkeiten. Java 3D würde nichts anderes übrig bleiben, als einige Sounds stumm zu schalten (wobei sie allerdings unhörbar weiter abgespielt werden würden, die korrekte Abspielposition innerhalb des Soundstreams bliebe also erhalten). Da das aber unter Umständen die falschen weil für die Anwendung besonders wichtigen Sounds treffen kann, gibt es die Möglichkeit, mit Hilfe des Parameters `priority` festzulegen, welche Sound-Nodes wie wichtig sind. Um so höher der hier übergebene Zahlenwert, um so höher ist die Priorität beim Abspielen. Es werden also immer erst den Sounds Plätze auf dem Audiodevice zugewiesen, die die höchste Priorität haben. Sind dann noch weitere Kanäle möglich, werden die Sound-Nodes mit der nächst niedrigeren Priorität verwendet. Aber auch hier gilt: Sind für Sounds der gleichen Priorität trotzdem nicht genügend Kanäle vorhanden, werden einzelne Sounds stumm geschaltet und die Sounds unhörbar weiter abgespielt. Die Auswahl, welche Sound-Nodes das sind, erfolgt dabei rein zufällig bzw. auf Grundlage der Datenanordnung innerhalb von Java 3D.

Da die Klasse `Sound` recht viele Methoden und damit auch sehr viele dazu gehörende Capabilities besitzt, sollen die Capability-Konstanten hier erstmalig gleich zusammen mit den Methoden erwähnt werden, deren Verwendbarkeit sie nach einem `compile()` beeinflussen:

```
boolean getContinuousEnable()
void setContinuousEnable(boolean state)
```

Diese Methoden gehören zum Parameter `continuous` des oben beschriebenen Konstruktors. Es ist mit ihnen also möglich zu ermitteln, ob ein Sound stumm weiter

```
(boö state) ! void
```

er auch gewesen wäre, wenn er nicht deaktiviert worden wäre. Soll ein Sound gezielt von vorne abgespielt werden, so kann das mit einem erneuten Aufruf von `setEnabled(true)` erzwungen werden.

7125 Die zu diesen beiden Methoden gehörenden Capability-Konstanten sind **ALLOW\_ENABLE\_READ** sowie **ALLOW\_ENABLE\_WRITE**.

```
float getInitialGain()
```

```
void setInitialGain(float amplitude)
```

7130 „Initial Gain“ bezeichnet die Gesamtlautstärke des abzuspielenden Sounds, welche bei `amplitude`-Werten kleiner eins leiser werden als durch die Sounddatei vorgesehen und bei Werten größer eins lauter. Wird der Sound lauter abgespielt, damit also verstärkt, so ist zu beachten, dass das bei zu großen Verstärkungen zu Verzerrungen führt. Der aktuelle Wert für die Gesamtlautstärke lässt sich mit der ersten dieser beiden Methoden  
7135 ermitteln, mit der Zweiten ist es möglich, einen neuen Lautstärkewert festzulegen. Die zur `get`-Methode gehörende Capability-Konstante ist **ALLOW\_INITIAL\_GAIN\_READ**, für das Schreiben neuer Initial-Gain-Werte ist **ALLOW\_INITIAL\_GAIN\_WRITE** zu setzen.

```
int getLoop()
```

7140 `void setLoop(int loopCount)`

Diese beiden Methoden stehen im Zusammenhang mit dem Loop-Wert, also der Angabe, wie oft der Sound zu wiederholen ist. Der aktuelle Loop-Wert kann mit der ersten Methode ermittelt werden. Soll diese auch nach einem `compile()` verwendbar sein, ist beim Aufruf von `setCapability()` die Konstante **ALLOW\_LOOP\_READ** zu übergeben.

7145 Die zweite Methode erlaubt es, einen neuen Wert für die Anzahl der Wiederholungen zu setzen, die zu ihr gehörende Capability-Konstante ist **ALLOW\_LOOP\_WRITE**.

```
boolean getMute()
```

```
void setMute(boolean state)
```

7150 Ähnlich wie von herkömmlicher Audiohardware wie Verstärkern oder Receivern her bekannt, kann auch ein Java 3D Sound stumm geschaltet werden. Geschieht das, so wird er unabhängig vom Status des `continuous`-Flags unhörbar weiter abgespielt (da es sich ja nicht um eine Deaktivierung sondern um eine gezielte Stummschaltung handelt). Mit diesen Methoden lässt sich ermitteln, ob der Sound-Node gerade stumm geschaltet wurde  
7155 bzw. es ist möglich, ihn umzuschalten (`true` steht für eine Stummschaltung während eine Übergabe von `false` als Parameter den Sound wieder hörbar macht). Die zu diesen Methoden gehörenden Capability-Konstanten sind dementsprechend **ALLOW\_MUTE\_READ** und **ALLOW\_MUTE\_WRITE**.

7160 `int getNumberOfChannelsUsed()`

Diese Methode liefert die Information zurück, wie viele Kanäle auf der Audiohardware benötigt werden, um den zugehörigen Sound abzuspielen. Der damit ermittelte Zahlenwert hängt natürlich von den Fähigkeiten der Hardware und der Positionierung des Sound im virtuellen 3D-Raum ab und kann somit vom User kaum  
7165 beeinflusst werden. Um diese Methode auch in compilierten SceneGraphen nutzen zu können, ist die Capability **ALLOW\_CHANNELS\_USED\_READ** erforderlich.

```
boolean getPause()
void setPause(boolean state)
```

7170 Diese Methoden stellen gewissermaßen das Gegenstück zu `getMute()` und `setMute()` dar. „Pause“ bedeutet, das ein Sound unterbrochen wurde und nicht weiter abgespielt wird. Erst wenn diese Pause durch den Aufruf von `setPause(false)` beendet wird, so wird dieser ab der Stelle weiter abgespielt, an der er sich zum Zeitpunkt des Aufrufes von `setPause(true)` befand. Der aktuelle Zustand des Sound-Nodes in  
7175 Bezug auf diese Pause kann mit der ersten der beiden Funktionen ermittelt werden. Ähnlich wie dem Muting ist die Pause unabhängig vom Status des `continuous`-Flags, da dieses nur das Verhalten nach dem Aufruf von `setEnabled()` beeinflusst. Die zu diesen beiden Methoden gehörenden Capability-Konstanten heißen erwartungsgemäß **ALLOW\_PAUSE\_READ** und **ALLOW\_PAUSE\_WRITE**.

7180  

```
float getPriority()
void setPriority(float priority)
```

Wie bereits beim Konstruktor beschrieben, können Sound-Nodes priorisiert werden, um sicherzustellen, das bei zu wenigen verfügbaren Hardwareressourcen die  
7185 wichtigsten Sounds nach Möglichkeit vorrangig abgespielt werden. Mit diesen beiden Methoden läßt sich der aktuelle Prioritätswert ermitteln bzw. ist es möglich, einen neuen Wert zu setzen. Die zur get-Methode gehörende Capability-Konstante heißt hier **ALLOW\_PRIORITY\_READ**, während das zur set-Methode gehörende Pendant **ALLOW\_PRIORITY\_WRITE** ist.

7190  

```
float getRateScaleFactor()
void setRateScaleFactor(float scaleFactor)
```

Mit dem `RateScaleFactor` bietet sich eine weitere interessante Möglichkeit, den abgespielten Sound zu beeinflussen. Dieser Faktor gibt an, um wie viel die eigentliche  
7195 Abspielfrequenz des Sounds geändert werden soll. Werte größer eins erhöhen dabei die Abspielgeschwindigkeit, was bedeutet, dass die Sounds deutlich höher klingen und auch schneller abgespielt werden. Das Ergebnis ist – abhängig vom Faktor und vom verwendeten Sound – eine Art Micky-Maus-Stimme. Faktoren kleiner eins senken die zum Abspielen verwendete Samplefrequenz, was dazu führt, das die Sounds tiefer klingen und  
7200 langsamer abgespielt werden. Hier wäre das Resultat bei kleinen Faktoren ein sehr tiefes Geräusch. Mit diesen beiden Methoden ist es nun möglich, den aktuellen Scale-Faktor zu ermitteln bzw. ihn auf einen neuen Wert zu setzen. Die dazu gehörenden Konstanten für die Beeinflussung der nach einem `compile()` oder einer „Liveschaltung“ noch  
7205 verbleibenden Capabilities sind **ALLOW\_RATE\_SCALE\_FACTOR\_READ** und **ALLOW\_RATE\_SCALE\_FACTOR\_WRITE**.

```
boolean getReleaseEnable()
void setReleaseEnable(boolean state)
```

7210 Wie bereits beschrieben legt das `release`-Flag fest, ob ein Sound dennoch bis zu Ende gespielt werden soll, wenn er deaktiviert wird oder ob er sofort zu unterbrechen ist.



Mit diesen Methoden ist es nun möglich, den aktuellen Zustand dieses Flags zu ermitteln bzw. es auf einen neuen Wert zu setzen. Die korrespondierenden Capability-Konstanten hierfür heißen **ALLOW\_RELEASE\_READ** und **ALLOW\_RELEASE\_WRITE**.

7215

```
Bounds getSchedulingBounds()
```

```
void setSchedulingBounds(Bounds region)
```

7220

Diese beiden Methoden beziehen sich auf den Einflußbereich, innerhalb dessen ein Sound-Node wirksam sein soll. Mit ihnen ist es möglich, das aktuelle Bounds-Objekt zu ermitteln, das diese Scheduling Bounds spezifiziert oder aber neue Bounds festzulegen. Soll der SceneGraph, in dem sich dieser Sound-Node befindet, kompiliert werden, so sind die Konstanten **ALLOW\_SCHEDULING\_BOUNDS\_READ** bzw. **ALLOW\_SCHEDULING\_BOUNDS\_WRITE** zu verwenden, wenn diese Methoden dann noch nutzbar bleiben sollen.

7225

```
MediaContainer getSoundData()
```

```
void setSoundData(MediaContainer soundData)
```

7230

Wie ebenfalls bei der Beschreibung eines der Konstruktoren bereits zu sehen war, werden die eigentlichen Sounddaten (also das Soundfile in z.B. dem RIFF-WAVE- oder dem Sun-Audio-Format) mit Hilfe eines MediaContainers geladen und an den Sound-Node übergeben. Diese Methoden erlauben es nun, das aktuelle MediaContainer-Objekt zu ermitteln (die zugehörige Capability-Konstante ist **ALLOW\_SOUND\_DATA\_READ**) bzw. ein neues MediaContainer-Objekt und damit einen neuen Sound an den Node zu übergeben (dementsprechend ist **ALLOW\_SOUND\_DATA\_WRITE** die dazu gehörende Konstante für die Capabilities).

7235

```
boolean isPlaying()
```

```
boolean isPlaying(View view)
```

7240

Diese Methoden sind eher von informativer Natur. Sie liefern die Information zurück, ob ein Sound-Node für irgend einen der eventuell mehreren Views gerade abgespielt wird oder aber, ob er gerade speziell für den Beobachter abgespielt wird, der `view` zugeordnet ist. Diese beiden Methoden liefern dabei jeweils immer dann `true` zurück, wenn der Sound hörbar ist oder aber, wenn er in unhörbarem Zustand abgespielt wird (also wenn er z.B. mangels genügend Hardwareressourcen nicht hörbar ist oder aber stumm geschaltet wurde).

7245

Auch für diese Methoden existiert mit **ALLOW\_IS\_PLAYING\_READ** eine Konstante für die Capabilities.

```
boolean isPlayingSilently()
```

7250

```
boolean isPlayingSilently(View view)
```

7255

In Erweiterung der Information, die die vorhergehenden beiden Methoden geben, gibt `isPlayingSilently()` an, ob ein Sound gerade abgespielt wurde und dabei nicht hörbar ist, weil er beispielsweise stumm geschaltet wurde. Auch hier kann diese Information wieder global ermittelt werden oder aber einen spezifischen Beobachter, der `view` zugeordnet ist.

Für diese Methoden ist wie für die vorhergehenden beiden die Konstante

**ALLOW\_IS\_PLAYING\_READ** zu verwenden, wenn sie nach einem `compile()` noch verwendet werden sollen.

```
7260 boolean isReady()
boolean isReady(View view)
```

Sound-Nodes kennen einen speziellen Zustand „Ready“. Dieser ist dann erreicht, wenn der zugehörige MediaContainer in der Lage war, einen Sound so weit zu laden, dass alle für die Wiedergabe notwendigen Informationen vorhanden sind und wenn alle für MediaContainer und Sound-Node intern erforderlichen Initialisierungen erfolgreich ausgeführt werden konnten. Anschließend ist das Sound-Objekt in der Lage, den Sound abzuspielen (was abhängig von der Menge vorhandener und aktiver Soundnodes und den verfügbaren Ressourcen der Audiohardware nicht zwangsläufig in ein hörbares Resultat münden muß). Ob ein Sound nun diesen Zustand „Ready“, also fertig zum Abspielen, erreicht hat, läßt sich mit Hilfe dieser Methoden (global oder für einen speziellen View) ermitteln.

Die letzte verbleibende Capability-Konstante **ALLOW\_IS\_READY\_READ** ermöglicht es wiederum, diese Methoden auch dann zu verwenden, wenn der SceneGraph, zu dem dieses Sound-Objekt gehört, kompiliert wurde oder wenn er live ist.

7275  
Die Klasse Sound leitet sich von einigen jetzt nicht mehr unbekannten Klassen ab:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
7280 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Sound
```

### 11.1.3 BackgroundSound

7285  
Nach so viel grauer Theorie über die zu Grunde liegende Klasse Sound im allgemeinen wird es doch langsam Zeit, das im vorherigen Abschnitt erfahrene endlich praktisch umzusetzen. Wie bereits angekündigt soll das in den folgenden Beispielprogrammen zu erzeugenden Szenen mit ein wenig mehr Aufwand betrieben werden, so dass neben dem Hörgenuß auch etwas fürs Auge geboten wird. Da für den optischen Teil bekanntes aus den vorangegangenen Kapiteln zum Einsatz kommt, muß der entsprechende Code allerdings nicht detaillierter erläutert werden.

7295  
Das folgende Beispielprogramm benötigt wieder eine Navigationsmöglichkeit, die in alt bekannter Weise mit zwei Behaviors gelöst werden soll: mit MouseRotate und MouseZoom. Diese ermöglichen das Rotieren und Verschieben der ViewPlatform mittels gedrückter linker bzw. mittlerer Maustaste und werden wie bisher auch gehabt zusammen mit dem Umgebungslicht in der Methode `createSceneGraph()` angelegt. In dieser Methode finden sich jetzt allerdings auch Aufrufe anderer Methoden wie

7300 `createEnvironment()` (hier wird ein einfarbiger, weißer Background erzeugt),  
`createObjects()` (hier wird mittels eines `QuadArrays` sowie dazugehöriger  
Texturkoordinaten ein rechteckiges Objekt erzeugt, das mit einer Textur versehen wird, so  
dass es wie ein roter Teppich aussieht) sowie `createSoundNodes()` (dies ist die  
eigentlich interessante Methode, die im folgenden detailliert beschrieben werden wird).

7305

Doch bevor es daran gehen kann, mit dem ersten Sound-Node ein Hintergrundgeräusch zu erzeugen, ist ein wichtiger, vorbereitender Schritt nötig: Die Initialisierung und Konfiguration des `AudioDevices`. Die dazu notwendigen Aufrufe finden sich in der `init()`-Methode des Beispielprogrammes:

7310

```
AudioDevice auDev=u.getViewer().createAudioDevice();
auDev.setAudioPlaybackType(AudioDevice.STEREO_SPEAKERS);
```

7315

Mit dem ersten Aufruf wird das `AudioDevice` für den aktuellen Viewer initialisiert. Dieser wird mit Hilfe der Methode `getViewer()` des verwendeten `SimpleUniverse` ermittelt. Ein anschließender Aufruf von `createAudioDevice()` initialisiert dann das `AudioDevice` und versetzt das Programm überhaupt erst in die Lage, Sounds hörbar abspielen zu können. In der zweiten Zeile wird das `AudioDevice` für die Art der angeschlossenen Audiokomponenten konfiguriert. Da in der Regel zwei Lautsprecher angeschlossen sein dürften, um die vom Computer wiedergegebenen Klänge auch in Stereo abspielen zu können, wird hier dementsprechend der Modus **STEREO\_SPEAKERS** gesetzt. Alternative Möglichkeiten dazu wären **MONO\_SPEAKER** (für die Mono-Wiedergabe mittels nur eines Lautsprechers) oder **HEADPHONES** (für die Stereo-Wiedergabe über Kopfhörer). Entsprechend den hier gemachten Angaben über die Audio-Peripherie werden die Sounds innerhalb der Szene anders berechnet um ein möglichst wirklichkeitsgetreues Klangabbild zu erhalten.

7325

Doch nun zur eigentlichen Erzeugung eines Soundnodes für einen Hintergrundsound, was sich kurz und knapp so erledigen lässt:

7330

```
(1)void createSoundNodes(BranchGroup bg)
(2) {
(3) MediaContainer BGMC;
(4) BackgroundSound BGSnd;
7335 (5)
(6) BGMC=new MediaContainer("file:./voegel.wav");
(7) BGSnd=new BackgroundSound(BGMC,1.0f,-1,false,true,true,BigBounds,0);
(8) bg.addChild(BGSnd);
(9) }
```

7340

Der Methode wird eine `BranchGroup` als Parameter mitgeliefert, in welche die `SoundNodes` einzufügen sind, damit sie anschließend live werden können. In den ersten Zeilen finden sich lediglich ein paar Variablendeklarationen. In Zeile 6 wird dann das

benötigte MediaContainer-Objekt erzeugt, das sich um die Beschaffung der Sounddaten kümmern soll. Als Parameter wird hier ein String mit dem URL auf die Sounddatei übergeben. Wie zu sehen ist, handelt es sich hier um ein lokal gespeichertes File, dessen Speicherort in Form eines relativen Pfades angegeben wird. Kann dieses File nicht gefunden werden, so äußert sich das in einer SoundException, die beispielsweise so aussehen kann:

```
7350 javax.media.j3d.SoundException: file:./voegel.wav: Sound source data could not
be loaded
 at
 javax.media.j3d.SoundScheduler.attachSoundData(SoundScheduler.java:2675)
7355 at
 javax.media.j3d.SoundScheduler.processSoundAtom(SoundScheduler.java:1285)
 at
 javax.media.j3d.SoundScheduler.calcSchedulingAction(SoundScheduler.java:1577)
 at javax.media.j3d.SoundScheduler.renderChanges(SoundScheduler.java:787)
7360 at javax.media.j3d.SoundScheduler.processMessages(SoundScheduler.java:229)
```

In diesem Fall ist zu überprüfen, ob die Datei „voegel.wav“ sich tatsächlich an der Stelle befindet, an der das Beispielprogramm sie erwartet. In Zeile 7 wird letztendlich das BackgroundSound-Objekt erzeugt, das für das gewünschte Hintergrundgeräusch sorgen soll. Dabei kommt ein komplexerer der möglichen Konstruktoren zum Einsatz, der aber von der Basisklasse Sound her bereits bekannt ist:

```
7370 BackgroundSound(MediaContainer soundData,float initialGain,int
loopCount,boolean release,boolean continuous,boolean enable, Bounds
region,float priority)
```

Als erster Parameter wird zwangsläufig der MediaContainer übergeben, der sich um die Daten der hier verwendeten Sounddatei kümmert. Der Lautstärkewert initialGain kann mit 1.0 auch auf 100%, also auf der Originallautstärke des Samples belassen werden, da keine anderen Sounds vorhanden sind, auf die man die Lautstärke abstimmen müsste. Mit einem loopCount von -1 wird für dieses BackgroundSound-Objekt festgelegt, dass der Sound in einer Endlosschleife abgespielt werden soll. Die beiden folgenden Parameter sind für dieses Beispiel eher uninteressant, da keine Möglichkeit vorgesehen wurde, das Abspielen des Sounds zu unterbrechen. Würde das dennoch realisiert werden, so würde der Sound 1. sofort unterbrochen werden, wenn er deaktiviert wird (release=false) und er 2. in stumm geschaltetem Zustand weiter abgespielt werden (continuous=true). enable wird hier ebenfalls auf true gesetzt, damit der Sound baldmöglichst hörbar wird, d.h. dass er automatisch abgespielt wird, so bald der Soundnode im Zustand „ready“ ist. Für region wird wieder das bereits früher verwendete, hier global definierte, sehr große Bounds-Objekt übergeben, um den Einflußbereich zu definieren, innerhalb dessen der Sound hörbar sein soll. Auch der Wert für die Priorität ist hier eher nicht von Interesse, da nur ein einziger Sound zum Einsatz kommt.

7390 Wird nun dieses Beispielprogramm compiliert und ausgeführt, so ist zum einen die

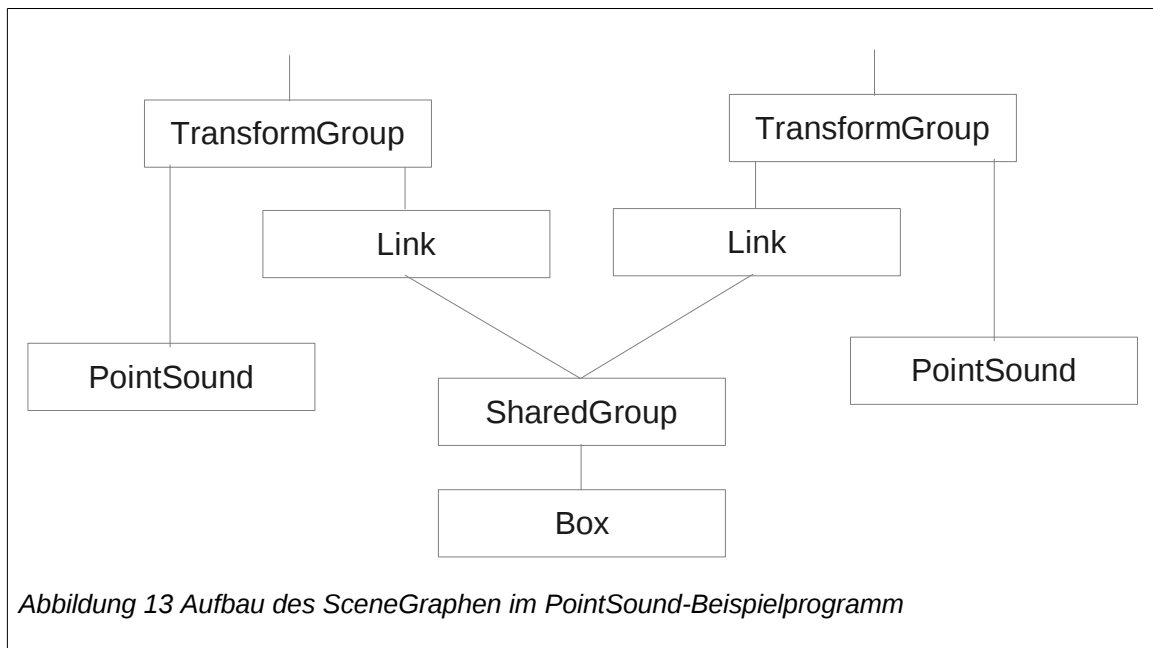
gewünschte Szene mit dem komplett weißen Hintergrund sowie dem roten Teppich zu sehen und zum anderen das Vogelgezwitscher des Soundfiles „voegel.wav“ zu hören. Dass es wirklich ein Hintergrundgeräusch ist, können Sie leicht überprüfen, in dem Sie die Position des Beobachters mittels der Mouse-Behaviors verändern wird: Egal, wie Sie sich innerhalb der Szene drehen oder bewegen, die Lautstärke oder die Richtung, aus der das Hintergrundgeräusch kommt, ändern sich dabei nicht.

Da die Klasse BackgroundSound den von ihrer übergeordneten Klasse geerbten Methoden keine neuen hinzufügt, bleibt nur noch ein Blick auf die Ableitungsverhältnisse dieser Klasse:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Sound
 javax.media.j3d.BackgroundSound
```

#### 11.1.4 PointSound

Im nächsten Schritt soll es nun darum gehen, einen so genannten PointSound, also eine punktförmige Schallquelle, 3D-Objekten innerhalb der Szene zuzuordnen. Dazu wird die Szene optisch wiederum etwas aufgepeppt, in dem jetzt zwei Lautsprecherboxen hinzukommen. Das sind nichts weiter als Primitives vom Typ Box, die mit einer Textur versehen wurden, so dass sie eben wie ganz normale HiFi-Boxen aussehen. Da diesen punktförmige Schallquellen zugeordnet werden, die den Schall gleichmäßig in alle Richtungen abstrahlen, kommt es dem Sachverhalt sehr entgegen, dass bei der Texturierung des Box-Primitives alle Seiten je einmal mit der Textur versehen werden. Auch wenn das für reale Lautsprecherboxen eher ungewöhnlich ist, die hier verwendeten haben auf jeder Seite Lautsprecher eingebaut, die den Schall abstrahlen (und dass die Textur sich in gleicher Weise auch an der Ober- und Unterseite der Boxen wiederfindet, ist dank der gewählten Position und der eingeschränkten Bewegungsfreiheit des Beobachters gar nicht zu sehen). Da zwei Boxen für links und rechts aufgestellt werden, bietet es sich hier wieder an, diese mit einem SharedGroup-Objekt zusammenzufassen. Hingegen sollen zwei unterschiedliche PointSound-Objekte zum Einsatz kommen, die sich zwar jeweils an der Position der zugehörigen Box befinden, aber unterschiedliche Sounds abspielen. Diese müssen also vor der SharedGroup von den TransformGroups, die die Position der Boxen festlegen, als Child eingefügt werden. Das ergibt dann also folgende SceneGraph-Struktur:



Da die Lautsprecher selber nicht in der Mitte der Boxen eingebaut sind, muß die Position der PointSounds also relativ zur Position der Boxen selber noch etwas angehoben, sprich in Y-Richtung verschoben werden. Das ist jedoch kein Problem, da die Klasse PointSound  
 7435 eine entsprechende Funktionalität mitbringt.

Der Aufbau der gesamten Szene inklusive der auch hier wieder benötigten Behaviors für die Navigation passiert wieder komplett in der Methode createSceneGraph() bzw. in anderen, von dort aus aufgerufenen Methoden. Dabei handelt es sich unter anderem auch  
 7440 um den Aufruf von createSoundNodes(), wobei hier jetzt die beiden TransformGroups – deren Child die PointSounds ja werden sollen – als Parameter mitgegeben werden müssen. Diese Methode selber sieht dann folgendermaßen aus:

```

(1) void createSoundNodes(TransformGroup TGL, TransformGroup TGR)
7445 (2) {
(3) MediaContainer MCL, MCR;
(4) PointSound PSndL, PSndR;
(5) Point2f[] DistGain = new Point2f[3];
(6)
7450 (7) DistGain[0] = new Point2f(0.0f, 1.0f);
(8) DistGain[1] = new Point2f(5f, 1.0f);
(9) DistGain[2] = new Point2f(20f, 0.0f);
(10)
(11) MCL = new MediaContainer("file:./loon.wav");
7455 (12) PSndL = new PointSound(MCL, 1.0f, -1, false, true, true, BigBounds, 0, new
 Point3f(0.0f, 0.6f, 0.0f), DistGain);
(13) TGL.addChild(PSndL);
(14)

```

```

(15) MCR=new MediaContainer("file:./kuh.wav");
7460 (16) PSndR=new PointSound(MCR,0.15f,-1,false,true,true,BigBounds,0,new
 Point3f(0.0f,0.6f,0.0f),DistGain);
(17) TGR.addChild(PSndR);
(18) }

```

7465 Es fällt auf, dass jetzt fast alles zweimal erledigt wird. Das ist nur logisch, da ja ein Stereo-Effekt simuliert werden soll und somit eine Lautsprecherbox links und eine rechts steht und mit Sound „versorgt“ werden muß. In den Zeilen 3 und 4 finden sich demzufolge Variablendeklarationen für die beiden MediaContainer sowie die PointSounds. In Zeile 5 wird dann ein Array angelegt, das in den Zeilen 7 bis 9 mit Daten gefüllt wird und bei der

7470 Erzeugung der PointSound-Objekte benötigt wird. Doch bevor dessen Konstruktor genauer beschrieben wird, ein näherer Blick auf den Rest des Codes. In den Zeilen 11 und 15 werden die MediaContainer erzeugt, die jeweils ein anderes Soundfile laden, um die rechten und die linke Lautsprecherbox besser unterscheidbar zu machen. In den Zeile 12 und 16 werden die gleich noch detaillierter beschriebenen PointSound-Objekte erzeugt und diese dann jeweils in den Zeilen 13 und 18 als Child derjenigen TransformGroups dem SceneGraphen hinzugefügt, die auch die Position der zugehörigen Lautsprecherbox festlegen.

7475

Bei der Erzeugung der PointSound-Objekte in den Zeilen 12 und 16 wurde dieser

7480 Konstruktor verwendet:

```

PointSound(MediaContainer soundData,float initialGain,int
loopCount,boolean release,boolean continuous,boolean enable,Bounds
region,float priority,Point3f position,Point2f[] distanceGain)

```

7485

Da es weitere Konstruktoren gibt, die von den Eigenschaften her, die mittels der übergebenen Parameter festgelegt werden, identisch sind, für die Übergabe der Werte lediglich andere Datentypen verwenden, seien diese in dem Zusammenhang ebenfalls erwähnt:

7490

```

PointSound(MediaContainer soundData,float initialGain,int
loopCount,boolean release,boolean continuous,boolean enable,Bounds
region,float priority,Point3f position,float[]
attenuationDistance,float[] attenuationGain)

```

7495

```

PointSound(MediaContainer soundData,float initialGain,int
loopCount,boolean release,boolean continuous,boolean enable,Bounds
region,float priority,float posX,float posY,float posZ,Point2f[]
distanceGain)

```

7500

```

PointSound(MediaContainer soundData,float initialGain,int
loopCount,boolean release,boolean continuous,boolean enable,Bounds
region,float priority,float posX,float posY,float posZ,float[]

```

```
attenuationDistance,float[] attenuationGain)
```

7505

Die ersten Parameter sind bereits von der Basisklasse Sound her bekannt und damit auch vom BackgroundSound. Dabei handelt es sich zum einen wieder um den MediaContainer, der die Sounddaten liefert sowie zum anderen um den Wert für die Gesamtlautstärke des abzuspielenden Sounds. Hier sind im Code bereits Unterschiede zu sehen. Während „loon.wav“ mit 100%iger Lautstärke abgespielt wird, ist der Faktor bei „kuh.wav“ 0.15, was 15% entspricht. Das war an dieser Stelle notwendig, um beide Sounds etwa gleich laut erscheinen zu lassen, das Kuh-Sample hätte den anderen Sound sonst völlig übertönt. Auch beim loopCount gibt es nichts neues, für dieses Beispiel wird wieder eine Endlosschleife gewählt. release und continuous, die das Verhalten des Sound-Nodes beeinflussen, wenn der Sound mittels setEnable() (de)aktiviert wird, sind auch hier eher unwichtig, da im Beispiel der Sound nicht abgeschaltet wird. Wichtig ist wieder, dass der PointSound von Anfang an mittels des Parameters enable aktiviert wird, und dass sein Einflußbereich festgelegt wird, innerhalb dessen er hörbar sein soll.

7510

7515

7520

Die folgende Eigenschaft kann dann je nach verwendetem Konstruktor mittels unterschiedlicher Datentypen festgelegt werden. Es handelt sich um die Position des Sound-Nodes relativ zu der Position, die durch übergeordnete Transformationen festgelegt wird. Das heißt für dieses Beispiel, dass zusätzlich zu der Position, die durch die beiden TransformGroups festgelegt wird, die Position des PointSounds um 0.6 m nach oben verschoben wird, also etwa in die Höhe, in der die Lautsprecher eingebaut sind. Wie zu sehen ist, kann diese relative Position einmal mit Hilfe eines Point3f-Objektes festgelegt werden aber auch über drei einzelne floats für jeweils X-, Y- und Z-Position.

7525

7530

Die letzte Eigenschaft, die festzulegen ist, hat mit dem Array aus Point2f-Objekten zu tun, das am Anfang der beschriebenen Methode createSoundNodes() erzeugt und mit Daten gefüllt wurde. Ein Blick auf die anderen Konstrukteure, in denen diese Daten mit Hilfe zweier float-Arrays attenuationDistance und attenuationGain übergeben werden, verraten bereits ein wenig, was hier definiert wird. Diese Arrays legen mit Hilfe der Datenpaare für Entfernung und Lautstärkefaktor fest, in welchem Abstand zum PointSound-Objekt der zugehörige Sound noch wie laut zu hören sein soll. Die Zwischenwerte zwischen den einzelnen Entfernungsangaben werden dabei jeweils linear interpoliert. Auch ist es naheliegend, dass die Entfernungswerte innerhalb des Arrays immer größer werden müssen. Ist eine Entfernungsangabe kleiner als Ihr Vorgänger, so wird eine Exception geworfen oder der Sound ist schlichtweg nicht hörbar.

7535

7540

Für das Beispielprogramm bedeutet dies nun: da für die ersten beiden Gain-Werte des Array jeweils der Wert 1.0 angegeben wurde, ist in dem Bereich von 0.0 m Entfernung bis 5.0 m Entfernung zur Schallquelle keine Abschwächung der Lautstärke zu vernehmen. Damit ist sichergestellt, dass bei einer Position in unmittelbarer Nähe oder gar direkt zwischen den beiden Boxen diese gleich laut zu hören sind. Das letzte Wertepaar legt hingegen eine Entfernung von 20 m sowie einen Gain-Wert von 0.0 fest. Das bedeutet nun logischerweise nichts anderes, dass die hörbare Lautstärke der PointSound-Objekte im Bereich von 5m bis 20 m linear zur Entfernung abnimmt und bei Entfernungen von 20 m oder größer dann gar nichts mehr zu hören ist.

7545

7550

Diese Arrays bieten also eine wirklich elegante Möglichkeit, das Abstrahlverhalten der Schallquellen zielgerichtet und für die Erfordernisse passend zu definieren. Auch zeigt



sich hier eine Optimierungsmöglichkeit: Da die PointSound-Objekte ab einer Entfernung von 20 m nicht mehr hörbar sind, kann der Einflußbereich eigentlich auch dementsprechend eingeschränkt werden. Das heißt, dass für dieses Beispiel statt des alt  
7555 bekannten BigBounds-Objektes besser eine BoundingSphere mit einem Radius von exakt 20 m hätte verwendet werden können.

Wird dieses Programm nun compiliert und ausgeführt, zeigen sich interessante Effekte bei diesem Szenenaufbau. Wie zu erwarten wird der Sound mit wachsender Entfernung  
7560 immer leiser. Andererseits ändert sich aber auch die hörbare Position der Schallquellen entsprechend der Position der Lautsprecherboxen relativ zum Beobachter. Anders gesagt: Navigiert man mit der Maus erst genau zwischen die beiden Boxen und dreht sich dann um 180°, so werden – zumindest bei einer Stereowiedergabe – auch die Seiten  
7565 vertauscht, von denen die Samples zu hören sind. Die Kuh ist also passend zur neuen relativen Position der rechten Lautsprecherbox links zu hören und nicht mehr rechts.

Da diese Klasse mit Position und Distance/Gain einige neue Eigenschaften hat, sind dementsprechend auch neue Methoden sowie Capability-Konstanten hinzugekommen:

```
7570 void getDistanceGain(float[] distance,float[] gain)
void getDistanceGain(Point2f[] attenuation)
void setDistanceGain(float[] distance, float[] gain)
void setDistanceGain(Point2f[] attenuation)
```

Diese Methoden erlauben es, die aktuell verwendeten Distance-Gain-Datenpaare  
7575 zu ermitteln, bzw. neue Werte zu setzen. Die dazu gehörenden Capability-Konstanten, die eine Nutzung dieser Methoden auch nach dem Compilieren erlauben, sind **ALLOW\_DISTANCE\_GAIN\_READ** und **ALLOW\_DISTANCE\_GAIN\_WRITE**

```
int getDistanceGainLength()
```

7580 Diese Methode liefert die Anzahl der Distance-Gain-Wertepaare zurück. Für das obige Beispielprogramm bedeutet dies, dass der zurückgegebene Wert gleich der Länge des Point2f-Arrays ist.

```
void getPosition(Point3f position)
7585 void setPosition(float x,float y,float z)
void setPosition(Point3f position)
```

Werden diese Methoden verwendet, so ist es möglich, die aktuelle relative Position des PointSound-Objektes zu ermitteln (die Koordinaten werden dann in das übergebene Point3f-Objekt kopiert) oder diese auf einen neuen Wert zu setzen. Die für diese  
7590 Methoden geltenden Capability-Konstanten sind **ALLOW\_POSITION\_READ** und **ALLOW\_POSITION\_WRITE**.

Bevor mit der Klasse ConeSound ein weiteres, sicher noch interessanteres Sound-Objekt wieder an Hand eines praktischen Beispiels beschrieben wird, soll natürlich der

7595 obligatorische Blick auf die Klassenverwandtschaft des PointSounds geworfen werden:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
7600 javax.media.j3d.Leaf
 javax.media.j3d.Sound
 javax.media.j3d.PointSound
```

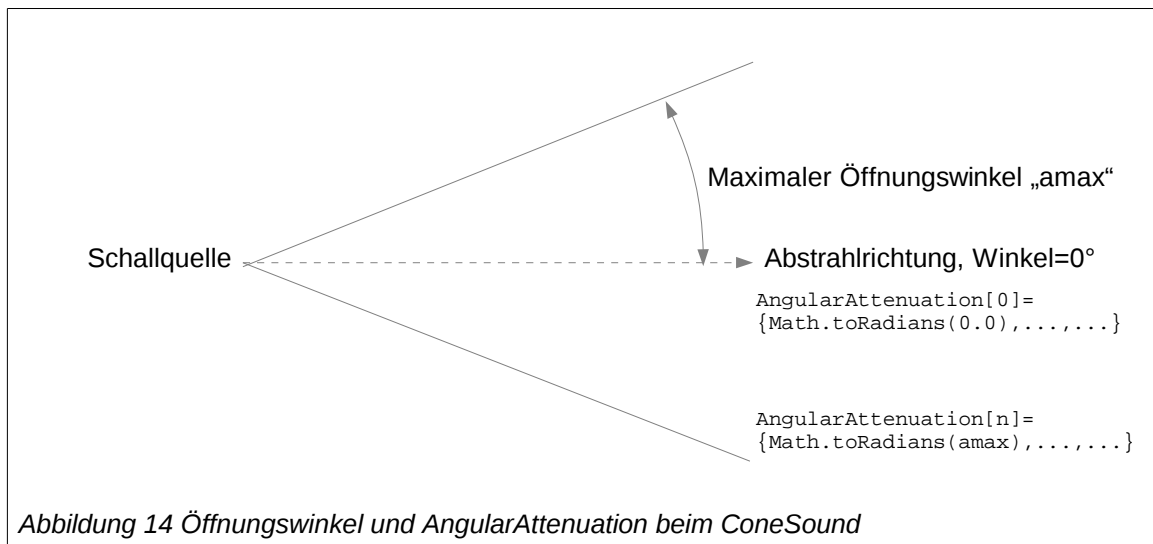
### 11.1.5 ConeSound

7605

Wie der Name bereits verrät, erzeugt dieser Sound-Node eine gerichtete Schallabstrahlung. Wie bereits von Java 3D gewohnt, läßt sich diese auch wieder detailliert konfigurieren. Da ConeSound aus der Klasse PointSound hervorgeht, gibt es zum einen wieder die Möglichkeit, die Distance Gain, also das Lautstärkeverhalten bei wachsender Entfernung anzugeben. Zum anderen ist es hier zusätzlich möglich, mit einem Array aus je drei Werten die Lautstärkeänderung vom Zentrum des Konus zu dessen Rand hin festzulegen (wozu Sie beispielsweise Point3f-Objekte verwenden können). Diese drei Werte sind dabei zum einen ein Winkel in der Einheit Radians, der den Abstand zum Zentrum festlegt, ein Lautstärkewert für den Sound der bei diesem Winkel noch vorhanden sein soll, sowie eine Filterfrequenz für einen Tiefpaß, der optional den Klang des Sounds beeinflussen kann. Der Winkel für den ersten Satz an Daten für die Angular Attenuation muß dabei zwangsläufig immer 0.0 sein, da die Zwischenwerte wieder interpoliert werden und somit ein Start-Gain-Wert angegeben werden muß. Das letzte Werte-Triplett hingegen legt mit der Angabe des darin enthaltenen Winkels fest, wie groß der Öffnungswinkel des Konus insgesamt sein soll. Und natürlich gilt auch hier wieder, dass die Winkel innerhalb dieses Datenarrays mit zunehmender Indexnummer immer größer werden müssen.

Das klingt kompliziert, ist jedoch eigentlich ganz einfach. Die Angular Attenuation funktioniert nach dem gleichen Prinzip wie der Distance Gain für die Entfernung, nur, dass hier statt Entfernungsangaben der Winkel innerhalb des konischen Bereiches angegeben wird, in dem der Sound abgestrahlt wird. Dazu kommt dann lediglich der Filter-Parameter, der für den optionalen Tiefpaß festlegt, wie hoch die Grenzfrequenz für eben diesen Winkel sein soll. Die folgende Abbildung wurde um ein wenig Pseudocode ergänzt, um Funktion und Wirkungsweise dieser Angular Attenuation zu verdeutlichen.

7630



Auch dieser Sound-Node soll natürlich mit einem Beispielprogramm ausprobiert werden. Dazu wird der Code aus dem vorhergehenden Abschnitt wieder ein wenig umgebaut. So wird nur noch eine Lautsprecherbox mit einem Lautsprecher dargestellt, da der Schall ja nicht mehr rundum abgestrahlt wird. Diese Box erhält eine Position etwas rechts abseits der Position der ViewPlatform. Mit dem MouseZoom Behavior ist es nun möglich, sich langsam an der Box vorbeizubewegen um den Effekt der konusförmigen, gerichteten Schallabstrahlung feststellen zu können. Damit sicher gestellt ist, dass das hörbare Ergebnis nicht von der reinen entfernungsabhängigen Lautstärkedämpfung herrührt, wird kein Distance-Gain-Array festgelegt. Das bedeutet, es werden die Default-Werte der zu Grunde liegenden Klasse PointSound verwendet, welche für diesen Fall keine entfernungsabhängige Lautstärkeänderung vorsehen.

Werden diese Anforderungen in Code gegossen, sieht das Ergebnis so aus:

```
(1) void createSoundNodes(TransformGroup TG)
(2) {
(3) MediaContainer MC;
7650 (4) Point3f[] AngGain=new Point3f[3];
(5) ConeSound CSnd;
(6)
(7) AngGain[0]=new Point3f(0.0f,1.0f,ConeSound.NO_FILTER);
(8) AngGain[1]=new
7655 Point3f((float)Math.toRadians(50),1.0f,ConeSound.NO_FILTER);
(9) AngGain[2]=new
Point3f((float)Math.toRadians(70),0.0f,ConeSound.NO_FILTER);
(10)
(11) MC=new MediaContainer("file:./beep.au");
7660 (12) CSnd=new ConeSound(MC,1.0f,0.0f,0.0f,0.0f,0.0f,0.0f,1.0f);
(13) CSnd.setEnabled(true);
(14) CSnd.setLoop(ConeSound.INFINITE_LOOPS);
(15) CSnd.setSchedulingBounds(BigBounds);
```

```

(16) CSnd.setAngularAttenuation(AngGain);
7665 (17) TG.addChild(CSnd);
(18) }

```

7670 Als erstes werden wieder diverse Variablen sowie das Array für die Angular Attenuation definiert. Dieses wird dann in den Zeilen 7 bis 10 mit Werten gefüllt, die auch recht leicht zu interpretieren sein sollten. Im Zentrum des Konus, also bei einem Winkel von 0° hat der Sound seine volle Lautstärke und mittels der Konstanten **NO\_FILTER** wird festgelegt, dass keine Tiefpaßfilterung erfolgen soll. In Zeile 8 werden die gleichen Werte für einen Winkel von 50° übergeben. Da auch hier die Zwischenwerte interpoliert werden, bedeutet dies nichts anderes, als dass in einem Bereich von 0° bis 50° die Lautstärke innerhalb des

7675 Trichters, in dem der Sound zu hören ist, nicht geringer wird. Wie mit einem Blick auf Zeile 9 festgestellt werden kann, ändert sich das für den Bereich von 50° bis 70° - hier nimmt die Lautstärke mit wachsendem Winkel ab, bis schließlich bei Werten ab 70° nichts mehr zu hören ist.

7680 In Zeile 11 wird dann schließlich das Soundfile geladen und nutzbar gemacht, in dem wieder ein alt bekannter MediaContainer erzeugt wird.

Da keine Werte für den Distance Gain übergeben werden sollen, ist es hier erforderlich, einen anderen Konstruktor zu verwenden:

```

7685 ConeSound(MediaContainer soundData, float initialGain, float posX,
float posY, float posZ, float dirX, float dirY, float dirZ)

```

7690 Der erste Parameter ist sicher klar, hier werden die Sounddaten über den Umweg des sie verwaltenden MediaContainer übergeben. Der zweite ist ebenfalls bereits bekannt, er legt die Gesamtlautstärke fest, die zusammen mit den Lautstärkewerten aus den Arrays für Distance Gain und Angular Attenuation die hörbare Lautstärke an jeder Position im Raum um die Schallquelle des ConeSound herum festlegt. Mit `posX`, `posY` und `posZ` wird anschließend nur noch die relative Position des Sound-Nodes festgelegt. Die letzten drei

7695 Parameter hingegen sind dann für diesen Soundtyp spezifisch. Da dieser eine gerichtete Schallabstrahlung simuliert, muß natürlich auch die Richtung angegeben werden, in die der Schall abgestrahlt werden soll. Das geschieht hier mit einem Vektor, der sich aus den Parametern `dirX`, `dirY` und `dirZ` zusammen setzt.

7700 In den folgenden Zeilen wird der Sound dann noch aktiviert (was bei der Verwendung dieses Konstruktors nicht geschehen ist und deswegen unbedingt separat erfolgen muß), die Anzahl der Wiederholungen werden spezifiziert (auch hier wieder eine Endlosschleife), es wird wieder der Einflußbereich, innerhalb dessen der Sound hörbar sein soll festgelegt und die Daten für die Angular Attenuation werden übergeben. Da

7705 diese Methoden gleich noch detailliert beschrieben werden, soll an dieser Stelle noch ein Blick auf einen anderen Konstruktor geworfen werden, der die logische Erweiterung zu den im Abschnitt über den PointSound beschriebenen darstellt und interessantes bietet:

```

7710 ConeSound(MediaContainer soundData,
 float initialGain,
 int loopCount,
 boolean release, boolean continuous, boolean enable,
 Bounds region,
 float priority,
7715 Point3f position,
 Point2f[] frontDistanceAttenuation,
 Point2f[] backDistanceAttenuation,
 Vector3f direction,
 Point3f[] angularAttenuation)
7720

```

Die ersten Parameter wurden 1:1 übernommen und sollten bekannt sein: Es werden damit die Sounddaten über den Umweg des MediaContainer übergeben, die Gesamtlautstärke und die Anzahl der Wiederholungen werden festgelegt, das Verhalten bei einer Unterbrechung wird spezifiziert und es kann angegeben werden, ob der Sound bereits aktiviert werden soll. Es folgen dann noch die Festlegung des Einflußbereiches, die Priorität sowie die relative Position des Sounds.

Anschließend wird es interessant, da sich hier gleich zwei verschiedene Arrays für die Distance Attenuation (was gleichbedeutend mit dem Distance Gain ist) übergeben werden können. Wie die Namen andeuten, handelt es sich hier um einen Wert für vorne (also die spezifizierte Abstrahlrichtung) und einen für die Rückseite (also die der Abstrahlrichtung entgegengesetzte Seite). Damit läßt sich ein Verhalten simulieren, was auch von realen Lautsprechern bekannt ist. Diese sollen hauptsächlich auch nach vorne abstrahlen, auf Grund ihres Aufbaus wird ein Teil des Schalls aber auch immer in die entgegengesetzte Richtung geschickt. Die Werte innerhalb der beiden für `frontDistanceAttenuation` und `backDistanceAttenuation` zu übergebenden Arrays sind dabei absolut gleichbedeutend mit denen des Distance-Gain-Arrays beim `PointSound`: Es wird in Wertepaaren die Entfernung zur Schallquelle und der dort gültige Lautstärkefaktor angegeben. Zwischenwerte werden dabei wieder interpoliert, was voraussetzt, dass der erste Entfernungswert 0 ist und die Entfernungen mit wachsenden Indexwerten immer größer werden. Wird für `backDistanceAttenuation` `null` übergeben, so bedeutet das, dass dieser `ConeSound` keinen Schall für die die dem spezifizierten Abstrahlvektor entgegengesetzte Richtung berechnet.

Dieser Abstrahl-Richtungsvektor, der die relative Richtung angibt, in die der Schall geschickt werden soll, wird mit dem Parameter `direction` festgelegt. Relativ heißt auch hier wieder, dass dieser Vektor abhängig von Rotationen im SceneGraph übergeordneter Transformationen eine Richtung festlegt, die sich nur auf die Achsen des `ConeSound`-Objektes bezieht.

Als letzter Parameter wird hier die Angular Attenuation festgelegt, wie sie oben bereits beschrieben wurde.

Auf Grund der neu hinzugekommenen Features und Funktionalitäten bietet die Klasse `ConeSound` auch wieder einige neue Methoden, denen sich auch Capability-Konstanten

zuordnen lassen, die dann wichtig werden, wenn der SceneGraph in compiliertem Zustand oder wenn er live ist die Möglichkeit bieten soll, diese Methoden dennoch zu verwenden. Des weiteren können natürlich auch diese Methoden mittels `setCapabilityIsFrequent()` und ihren zugehörigen Capability-Konstanten als besonders häufig benutzt markiert werden.

```
7760 void getAngularAttenuation(float[] distance,float[] gain,float[] filter)
void getAngularAttenuation(Point3f[] attenuation)
7765 void setAngularAttenuation(float[] distance,float[] gain,float[] filter)
void setAngularAttenuation(Point2f[] attenuation)
void setAngularAttenuation(Point3f[] attenuation)
```

Diese Methoden ermitteln die aktuellen Werte für die Angular Attenuation und kopieren sie in die als Parameter übergebenen Arrays bzw. sie ermöglichen es, neue Werte zu setzen. Bei den Methodenaufrufen, bei denen mehrere float-Arrays als Parameter erwartet werden, müssen diese separaten Arrays zum einen die gleiche Länge haben und beim Aufruf der get-Methode zusätzlich auch groß genug sein. Die Gesamtlänge des aktuell verwendeten Angular Attenuation Arrays kann dabei mit der

7775 nächstfolgenden Methode ermittelt werden.

Damit diese Methoden auch dann nutzbar bleiben, wenn der zugehörige SceneGraph live oder compiliert ist, müssen die Capabilities **ALLOW\_ANGULAR\_ATTENUATION\_READ** bzw. **ALLOW\_ANGULAR\_ATTENUATION\_WRITE** gesetzt sein.

```
7780 int getAngularAttenuationLength()
```

Diese Methode liefert die Gesamtlänge des aktuell verwendeten Angular Attenuation Arrays zurück. Für das obige Beispielprogramm wäre der Rückgabewert also 3.

```
7785 void getDirection(Vector3f direction)
void setDirection(float x,float y,float z)
void setDirection(Vector3f direction)
```

Es wird der aktuelle Vektor, der die Abstrahlrichtung des ConeSounds festlegt, ermittelt bzw. auf einen neuen Wert gesetzt, wenn diese Methoden verwendet werden.

7790 Die zu diesen Methoden gehörenden Capability-Konstanten sind **ALLOW\_DIRECTION\_READ** und **ALLOW\_DIRECTION\_WRITE**.

```
void getDistanceGain(float[] frontDistance,float[] frontGain,float[] backDistance,float[] backGain)
7795 void getDistanceGain(Point2f[] frontAttenuation, Point2f[] backAttenuation)
```

Diese Methoden liefern die derzeit verwendeten Distance-Gain-Daten für die Abstrahlrichtung (`frontDistanceGain`) und die ihr entgegengesetzte Richtung

(backDistanceGain) zurück, in dem die Werte in die als Parameter übergebenen Arrays kopiert werden.  
7800 Die zu diesen Methoden gehörende Capability-Konstante **ALLOW\_DISTANCE\_GAIN\_READ** wurde aus der Klasse PointSound geerbt.

```
void setBackDistanceGain(float[] distance,float[] gain)
7805 void setBackDistanceGain(Point2f[] attenuation)
```

Es werden die Distance-Gain-Werte für die der eigentlichen Abstrahlrichtung entgegengesetzte Seite neu festgelegt. Auch die hierzu gehörende Capability-Konstante **ALLOW\_DISTANCE\_GAIN\_WRITE** stammt aus der übergeordneten Klasse.

```
7810 void setDistanceGain(float[] frontDistance,float[]
frontGain,float[] backDistance,float[] backGain)
void setDistanceGain(Point2f[] frontAttenuation, Point2f[]
backAttenuation)
```

Statt nur für eine Abstrahlrichtung erlauben es diese beiden Methoden, die  
7815 Distance-Gain-Werte für beide Richtungen festzulegen bzw. neu zu setzen. Wie bei der vorhergehend beschriebenen Methode ist **ALLOW\_DISTANCE\_GAIN\_WRITE** die korrespondierende Capability-Konstante, die mit setCapability() verwendet werden muß, wenn die Distance Gain Werte auch dann noch veränderbar sein sollen, wenn der SceneGraph, zu dem dieses ConeSound-Objekt gehört, live oder kompiliert ist.

7820 Wie bereits erwähnt, leitet sich ConeSound direkt von der bereits bekannten Klasse PointSound ab:

```
java.lang.Object
7825 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Sound
 javax.media.j3d.PointSound
7830 javax.media.j3d.ConeSound
```

## 11.2 Klänge und Schall – mehr als nur Soundsamples

Eine weitere herausragende Fähigkeit von Java 3D findet sich in der Klasse Soundscape.  
7835 Sie bietet etwas, was viele andere 3D-Engines nicht mal ansatzweise erlauben: die Möglichkeit, die Sounds, die sich bereits in einer Szene befinden dadurch noch realistischer wirken zu lassen, in dem sie so beeinflußt werden, dass sie auch akustisch zur sichtbaren 3D-Umgebung passen.

Was das bedeutet, wird vielleicht an Hand eines Beispiels schneller deutlich: In der Realität würden wir von einer großen leeren Halle aus Blech- oder Betonwänden ein bestimmtes akustisches Verhalten erwarten: ein ziemlich starkes Echo. Hingegen würde ein kleiner Raum, der mit Teppich, Möbeln und vielleicht sogar Wandteppichen



```

(3) MediaContainer MC;
(4) Soundscape SoundS;
(5) AuralAttributes Aural;
7890 (6)
(7) MC=new MediaContainer("file:./loon.wav");
(8) PSnd=new PointSound(MC,0.6f,1,true,true,true,BigBounds,0,new
 Point3f(0.0f,0.6f,0.0f),null);
(9) PSnd.setCapability(PointSound.ALLOW_ENABLE_WRITE);
7895 (10) SG.addChild(PSnd);
(11)
(12) Aural=new AuralAttributes(3.0f,1.0f,0.99f,800.0f,20,null,1.0f,1.0f);
(13) SoundS=new Soundscape(new BoundingBox(new Point3d(-17.5,-5,-17.5),new
 Point3d(17.5,5,17.5)),Aural);
7900 (14) bg.addChild(SoundS);
(15) }

```

Die ersten Zeilen sollten klar sein: Neben einigen Variablendeklarationen wird das PointSound-Objekt erzeugt, das als Schallquelle innerhalb des geschlossenen virtuellen Raumes dienen soll. Beachtenswert ist bei diesem PointSound, dass die Gesamtlautstärke initialGain vorbeugend auf einen Wert von 60% gesetzt wurde. Das kann beim Einsatz von Soundscape und AuralAttributes erforderlich sein, da es sonst durch die noch hinzukommenden Audiodaten unter Umständen zu Übersteuerungen und damit unschönen Verzerrungen kommen kann.

7910

Anschließend werden die Objekte erzeugt, die dazu dienen, den Klang dieses Raumes zu berechnen. Das ist zum einen ein AuralAttributes-Objekt (in Zeile 12), welches festlegt, wie der Klang verändert werden soll. Und zum anderen ist das in Zeile 13 ein Soundscape-Objekt, das Größe und Position des Raumes festlegt, innerhalb der sich die Schallquelle sowie der Beobachter befinden muß, damit dieser den durch die AuralAttributes veränderten Schall hören kann. Befindet er sich außerhalb des Bereiches eines Soundscapes, wird der abgestrahlte Schall in völlig unveränderter Form hörbar.

Wird das Beispielprogramm nun compiliert und ausgeführt, ist ein deutlicher Unterschied zu hören: Neben dem eigentlichen Sound wird jetzt ein Echo geworfen, das im reinen Sample selbst nicht zu hören ist und damit offenbar dynamisch erzeugt wird.

Die Klasse AuralAttributes bietet viele Manipulationsmöglichkeiten, die den Klang beeinflussen und verändern. In der HiFi-Branche ist es üblich, Klänge mit sehr blumigen, meist jedoch immer noch nichtssagenden Begriffen zu beschreiben (von der High-End-Branche, in der mit noch viel abstruserem Voodoo-Zauber Geschäfte gemacht werden, einmal ganz zu schweigen). Aus diesem Grund soll hier nur der prinzipielle Einfluß der einzelnen Parameter auf die Berechnung des Raumklanges beschrieben werden. Wie sich das dann im einzelnen auf den hörbaren Klang auswirkt, läßt sich ja durch einfaches Verändern der Parameter leicht selbst herausfinden. Letztendlich gehört für die Justage eines Raumklanges immer ein wenig Erfahrung und Fingerspitzengefühl dazu, so dass sie

hier um ein paar Experimente mit den Parametern und Möglichkeiten sicher sowieso nicht herumkommen.

7935 `AuralAttributes(float gain,float rolloff,float  
reflectionCoefficient,float reverbDelay,int reverbOrder,Point2f[]  
distanceFilter,float frequencyScaleFactor,float  
velocityScaleFactor)`

7940 Mit dem ersten Parameter läßt sich wieder ein Lautstärkewert festlegen, der sich dieses mal auf die Effektberechnung zieht. Hier wird auch deutlich, warum die Lautstärke des eigentlichen Sound im Beispielprogramm verringert wurde. Da der Effekt mit einem `gain`-Faktor von 3.0 relativ laut ist, hätte es sonst hörbare Verzerrungen gegeben. Ob und in wie weit die Gesamtlautstärke vermindert werden muß, hängt dabei immer von den

7945 `AuralAttributes`-Parametern sowie der gewünschten Effektlautstärke ab.

`rolloff` ist ein Faktor, der sich auf die Schallgeschwindigkeit bezieht. Diese liegt bei Raumtemperatur innerhalb der Erdatmosphäre bei rund 333 m/s und könnte mit Hilfe dieses Faktors verändert werden. Das hätte dann Auswirkungen, wenn das

7950 `AuralAttributes`-Objekt so genannte Reverbation Bounds zur Berechnung des Klanges verwenden würde. Das ist ein Bounds-Objekt, dessen „Wände“ herangezogen werden würden, um die Schalllaufzeiten und damit die Verzögerungen, die ein Echo haben würde, zu ermitteln.

7955 Der `reflectionCoefficient` gibt an, wie stark der Schall zurückgeworfen werden soll, er sagt somit etwas über die akustischen Materialeigenschaften des Raumes aus, der simuliert wird. Für einen schallgedämpften Raum, der beispielsweise viel Stoff (wie Teppiche) enthält, müßte dieser Koeffizient sehr klein sein und in der Nähe von 0 liegen. Für einen Raum, der stark schallreflektierend ist (weil er beispielsweise recht glatte, harte

7960 und damit schallreflektierende Wände und Decken hat), müßte der Wert sehr groß sein und in der Nähe von 1.0 liegen. Wie zu sehen ist, liegt der zulässige Wertebereich des `reflectionCoefficient` bei 0.0..1.0 und er ist um so größer um so stärker der Schall reflektiert werden soll.

7965 Mit `reverbDelay` wird die Schalllaufzeit festgelegt, die angibt, nach welcher Zeitspanne das erste Echo zu hören sein soll. Um so kleiner dieser Wert ist um so halliger ist das Resultat, um so größer er wird, um so mehr wird aus dem Hall ein Echo, bei dem der reflektierte Schall erst deutlich später erneut zu hören ist. Werden anschließend mittels `setReverbBounds(Bounds)` die so genannten Reverbation Bounds festgelegt, so wird

7970 dieser feste Zahlenwert ignoriert und ein Verzögerungswert an Hand der Größe der Bounds und der Entfernung seiner Seiten berechnet.

Der folgende Parameter bezieht sich ebenfalls auf die Reflektionen. Mit `reverbOrder` ist es möglich, deren Anzahl zu begrenzen. Wird statt einer positiven Zahl, die dann nicht

7975 mehr als die so angegebenen Echos zuläßt, eine negative angegeben, so bedeutet das, dass die Anzahl der Reflektionen unbegrenzt ist. In diesem Fall sollte der `reflectionCoefficient` unbedingt kleiner als 1.0 sein, da sich die Reflektionen sonst

ungedämpft immer wieder wiederholen und das Ganze innerhalb kürzester Zeit zu einer rechten Kakophonie ausarten würde.

7980

Das Konzept des `distanceFilter` ist in ähnlicher Weise bereits von den Sounds her bekannt. Wird hier – wie im Beispielprogramm – `null` angegeben, so erfolgt keine Tiefpaßfilterung in Abhängigkeit zur Entfernung von der Schallquelle. Wird statt dessen ein `Point2f`-Array mit Wertepaaren gefüllt und übergeben, bei denen der erste Wert wieder für den Abstand zur jeweiligen Schallquelle steht und der zweite für die Tiefpaß-Grenzfrequenz, so wird der Sound entsprechend diesen Werten mit wachsendem Abstand gefiltert. Da die Zwischenwerte innerhalb des Arrays wieder interpoliert werden, ist es auch hier erforderlich, dass die Daten am Index 0 für eine Entfernung von 0 m festgelegt werden (also für die direkte Position der Schallquelle) und die Entfernungswerte mit wachsenden Indexwerten ebenfalls immer weiter zunehmen.

7985

7990

Mittels `frequencyScaleFactor` ist es möglich, die Frequenz des Sounds generell zu verändern. Das heißt, um so höher dieser Wert, um so höher klingt der Sound, um so kleiner der Faktor ist, um so tiefer und grollender ist das Ergebnis. Lediglich bei einem Faktor von 1.0 bleibt die Frequenz des Sounds unverändert.

7995

Der letzte Parameter, `velocityScaleFactor`, bezieht sich auf den – in diesem Beispielprogramm nicht mit abgedeckten – Dopplereffekt. Dieser Faktor legt fest, wie stark der Effekt bei bewegten Objekten sein soll, d.h. wie stark die Verzerrung der Frequenz sein soll, wenn sich das 3D-Objekt – oder besser dessen Schallquelle – auf den Beobachter zubewegt oder von ihm weg. Der akustische Eindruck ist dabei so, dass die Bewegung dieses 3D-Objektes um so schneller erscheint, um so höher dieser Faktor und damit um so höher die Frequenzverschiebung ist.

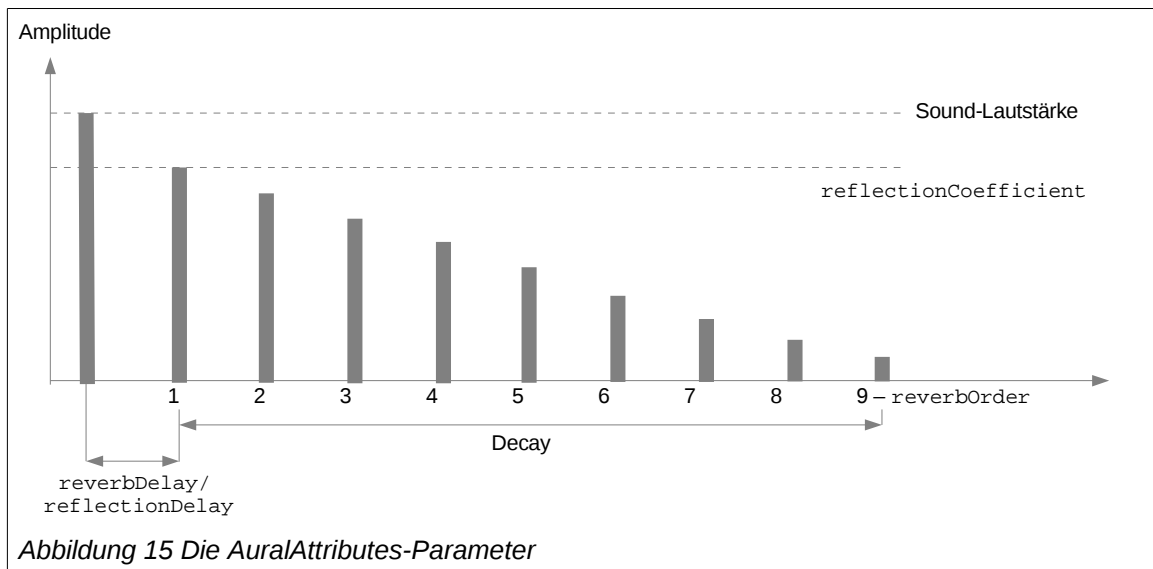
8000

8005

Auch wenn die kleine Anzahl an Parametern es nicht so erscheinen lässt, aber allein mit `reflectionCoefficient`, `reverbDelay` und `reverbOrder` lassen sich die unterschiedlichsten Räume akustisch simulieren. Hier gilt im allgemeinen: Um so mehr und um so stärkere Echos geworfen werden (`reflectionCoefficient` und `reverbOrder`), um so mehr klingt das nach einem leeren, nackten Raum wie einer Halle, einem Keller oder ähnlichem, in dem sich keine schalldämmenden Materialien wie Möbel oder Teppiche befinden. Und um so größer zusätzlich die Schallaufzeit bis zum ersten Echo ist (`reverbDelay`), um so größer scheint dieser Raum zu sein. Hier lohnt es sich zu experimentieren, da die Ergebnisse – wenn sie denn gut auf den visuellen Eindruck der restlichen 3D-Szene abgestimmt sind – eine virtuelle 3D-Welt deutlich realistischer wirken lassen können.

8010

8015



Auch die Klasse AuralAttributes bringt wieder einige interessante Methoden mit:

```
8020 float getAttributeGain()
void setAttributeGain(float gain)
```

Diese Methoden sind äquivalent zum ersten Parameter des oben beschriebenen Konstruktors. Damit ist es möglich, die Gesamtlautstärke des berechneten Effekts zu ermitteln bzw. sie auf einen neuen Wert zu setzen. Die zugehörigen Capability-

```
8025 Konstanten sind ALLOW_ATTRIBUTE_GAIN_READ und
ALLOW_ATTRIBUTE_GAIN_WRITE
```

```
float getDecayFilter()
void setDecayFilter(float frequencyCutoff)
```

```
8030 Der Term „Decay“, der bei diesen und den folgend beschriebenen Methoden
verwendet wird, beschreibt den Bereich des berechneten Sounds von der ersten
Reflektion bis zum vollständigen Abklingen jeglichen Nachhalls bzw. Echos. Wo Decay
genau einzuordnen ist, verdeutlicht auch noch einmal das Bild oben. Diese Methoden
beziehen sich auf eine Filterung all dieser Soundanteile. Mit ihnen ist es möglich, die
8035 aktuelle Grenzfrequenz des Tiefpaßfilters zu ermitteln bzw. sie auf einen neuen Wert zu
setzen. Soll eine dieser Methoden genutzt werden, wenn der zugehöriger SceneGraph
live oder kompiliert ist, so sind die Capabilities ALLOW_DECAY_FILTER_READ bzw.
ALLOW_DECAY_FILTER_WRITE zu setzen.
```

```
8040 float getDecayTime()
void setDecayTime(float decayTime)
```

Diese Methoden beziehen sich auf die Decay-Gesamtlänge in der Einheit Millisekunden. Sie erlauben es, die aktuelle Decay-Dauer zu ermitteln bzw. sie auf einen neuen Wert zu setzen. Die zugehörigen Capability-Konstanten sind

```
8045 ALLOW_DECAY_TIME_READ und ALLOW_DECAY_TIME_WRITE.
```

```
float getDensity()
```

```
void setDensity(float ratio)
```

8050 Die Density ist ein Parameter, der beim oben beschriebenen Konstruktor noch nicht  
vorkam. Dieser Faktor, der sich im Bereich von 0.0 .. 1.0 bewegt, beeinflusst die spektrale  
Klangfärbung der Reflektionen (das so genannte Timbre). Mit diesen Methoden ist es nun  
möglich, den aktuellen Density-Faktor zu ermitteln bzw. einen neuen Wert zu setzen.  
Capability-Konstante für das Setzen ist **ALLOW\_DENSITY\_READ**, während  
8055 **ALLOW\_DENSITY\_WRITE** zur get-Methode gehört.

```
float getDiffusion()
```

```
void setDiffusion(float ratio)
```

8060 Diffusion ist ein weiterer interessanter Parameter. Dieser Faktor, dessen Wert  
ebenfalls im Bereich 0.0 .. 1.0 liegen muß, beeinflusst die Streuung der reflektierten  
Schallanteile und bewirkt damit einen mehr oder weniger „weichen“ Klang des Halls bzw.  
Echos. Diese beiden Methoden erlauben es nun, den aktuellen Wert zu ermitteln bzw.  
Diffusion auf einen neuen Wert zu setzen. Sollen diese Methoden auch nach einem  
compile() angewendet werden bzw. wenn das zugehörige Soundscape-Objekt live ist,  
8065 so sind die Capabilities für **ALLOW\_DIFFUSION\_READ** bzw.  
**ALLOW\_DIFFUSION\_WRITE** zu setzen.

```
void getDistanceFilter(float[] distance, float[] frequencyCutoff)
```

```
void getDistanceFilter(Point2f[] attenuation)
```

```
void setDistanceFilter(float[] distance, float[] frequencyCutoff)
```

8070 

```
void setDistanceFilter(Point2f[] attenuation)
```

Diese Methoden erlauben es, den aktuellen Distance Filter, der Filterfrequenzen in  
Abhängigkeit von der Entfernung zur Schallquelle festlegt, zu ermitteln, bzw. einen neuen  
Filter mit Hilfe eines Point2f-Arrays oder aber zweier float-Arrays festzulegen. Die  
zugehörigen Capability-Konstanten lauten für diese Methoden  
8075 **ALLOW\_DISTANCE\_FILTER\_READ** und **ALLOW\_DISTANCE\_FILTER\_WRITE**.

```
int getDistanceFilterLength()
```

Mit dieser Methode läßt sich die Länge des aktuell verwendeten Distance-Filter-  
Arrays ermitteln.

8080

```
float getFrequencyScaleFactor()
```

```
void setFrequencyScaleFactor(float frequencyScaleFactor)
```

8085 Der Faktor für die Verschiebung, der die Wiedergabefrequenz beeinflusst, läßt sich  
mittels dieser Methoden ermitteln oder auf einen neuen Wert setzen. Die Konstanten für  
die Capabilities, die zusammen mit setCapability() oder  
setCapabilityIsFrequent() verwendet werden können, lauten hier  
**ALLOW\_FREQUENCY\_SCALE\_FACTOR\_READ** und  
**ALLOW\_FREQUENCY\_SCALE\_FACTOR\_WRITE**.

8090 `float getReflectionCoefficient()`

`void setReflectionCoefficient(float coefficient)`

Diese Methoden holen bzw. setzen den Wert für den Reflektionskoeffizienten, der festlegt, wie stark der Schall reflektiert werden soll. Gültige Werte liegen hierfür im Bereich von 0.0 (keine Reflektion) bis 1.0 (ungedämpfte Schallreflexion). Die dafür

8095 eventuell erforderlichen Capability-Konstanten sind  
**ALLOW\_REFLECTION\_COEFFICIENT\_READ** bzw.  
**ALLOW\_REFLECTION\_COEFFICIENT\_WRITE**.

`float getReflectionDelay()`

8100 `void setReflectionDelay(float reflectionDelay)`

Während die Reverberation Delay die Verzögerung zur Berechnung der Reflektionen generell beeinflusst, bezieht sich die Reflection Delay auf die einmalige Verzögerungszeit bis zur ersten Reflektion des Schalls. Wird hierfür kein spezifischer Wert übergeben, so ist diese Verzögerung bis zur ersten Reflektion gleich der Reverberation Delay.

8105 Diese beiden hier beschriebenen Methoden erlauben es nun, den Wert für die Reflection Delay zu ermitteln bzw. einen neuen Wert festzulegen. Die hierzu gehörenden Capability-Konstanten heißen dementsprechend **ALLOW\_REFLECTION\_DELAY\_READ** und **ALLOW\_REFLECTION\_DELAY\_WRITE**.

8110 `Bounds getReverbBounds()`

`void setReverbBounds(Bounds reverbVolume)`

Statt einen festen Reverberation-Verzögerungswert anzugeben, ist es ebenfalls möglich, Bounds zu verwenden, an Hand derer die Signallaufzeiten für die Größe des so spezifizierten Raumes berechnet werden. Die Größe des zur Berechnung zu

8115 verwendenden Raumes wird mit Hilfe eines Bounds-Objektes festgelegt, das oftmals identisch mit dem beim zugehörigen Soundscape-Objekt verwendeten Bounds-Objekt sein dürfte.

Mittels dieser beiden Methoden wird nun das aktuelle Bounds-Objekt geholt bzw. ein neues gesetzt. Wird ein Reverberation-Bounds-Objekt verwendet, so überschreibt dieses  
8120 alle eventuell für die Reverberation Delay spezifizierten Werte.

`float getReverbDelay()`

`void setReverbDelay(float reverbDelay)`

8125 Werden keine Reverberation-Bounds verwendet, so kommt ein fester Reverberation-Verzögerungswert zum Einsatz, dessen aktueller Wert mittels dieser Methoden ermittelt bzw. der damit auf einen neuen Wert gesetzt werden kann. Die hierfür eventuell nötigen Capability-Konstanten lauten **ALLOW\_REVERB\_DELAY\_READ** und **ALLOW\_REVERB\_DELAY\_WRITE**.

8130 `float getReverbCoefficient()`

`void setReverbCoefficient(float coefficient)`

Der Reverberation Koeffizient beeinflusst die Lautstärke der Reflektionen inklusive des Decay-Anteils. Mit diesen Methoden ist es möglich, ihn zu ermitteln bzw. einen neuen

8135 Wert für den Reverberation Koeffizienten zu setzen. Für eine Verwendung dieser Methoden innerhalb eines SceneGraphen, der live oder compiliert ist, werden die Konstanten **ALLOW\_REVERB\_COEFFICIENT\_READ** und **ALLOW\_REVERB\_COEFFICIENT\_WRITE** benötigt.

```
int getReverbOrder()
8140 void setReverbOrder(int reverbOrder)
```

Diese Methoden ermitteln bzw. setzen einen Wert für die maximale Anzahl der Reflektionen und korrespondieren mit den Capability-Konstanten **ALLOW\_REVERB\_ORDER\_READ** und **ALLOW\_REVERB\_ORDER\_WRITE**.

```
8145 float getRolloff()
void setRolloff(float rolloff)
```

Rolloff verändert die zur Berechnung der Schalllaufzeiten nötige Schallgeschwindigkeit um den mit diesen Methoden zu ermittelnden bzw. neu anzugebenden Faktor. Die hierzu gehörenden Konstanten für die Capabilities sind

8150 **ALLOW\_ROLLOFF\_READ** und **ALLOW\_ROLLOFF\_WRITE**.

```
float getVelocityScaleFactor()
void setVelocityScaleFactor(float velocityScaleFactor)
```

Die letzten beiden hier zu beschreibenden Methoden beziehen sich auf einen

8155 Faktor, der die Stärke des Doppler-Effektes bei bewegten Objekten festlegt. Dieser Faktor, der den Doppler-Effekt um so stärker hörbar macht, um so größer er ist, wird mit diesen Methoden geholt bzw. neu gesetzt. Die Capability-Konstanten, die bei einem Aufruf von `setCapability()` übergeben werden müssen, wenn eine oder beide dieser Methoden auch in einem SceneGraph verwendet werden sollen, der live oder compiliert

8160 ist, heißen hierfür **ALLOW\_VELOCITY\_SCALE\_FACTOR\_READ** und **ALLOW\_VELOCITY\_SCALE\_FACTOR\_WRITE**.

Bevor gleich anschließend die für diese Klang-Effekte eben so wichtige Klasse Soundscape beschrieben wird, abschließend wieder ein kurzer Blick auf die Ableitung der

8165 Klasse AuralAttributes:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.NodeComponent
8170 javax.media.j3d.AuralAttributes
```

### 11.2.2 Soundscape

Wie am Ende des vorherigen Abschnittes zu sehen war, erbt AuralAttributes nicht von der

8175 Klasse Node, was demzufolge ausschließt, dass ein AuralAttributes-Objekt direkt in einen  
SceneGraph eingefügt werden kann. Das wäre aus noch einem weiteren Grund nicht  
sinnvoll, da schließlich bisher nicht spezifiziert wurde, innerhalb welchen Bereiches in der  
virtuellen 3D-Welt Sounds in der durch AuralAttributes spezifizierten Weise verändert  
werden sollen. Wie das Beispielprogramm sowie die Überschrift dieses Abschnittes es  
8180 jedoch bereits verraten, ist das nötige Bindeglied zwischen dem SceneGraphen und dem  
AuralAttributes-Objekt die Klasse Soundscape.

In Zeile 13 des Beispielprogrammes ist zu sehen, wie ein solches Objekt erzeugt werden  
kann, während die Zeile 14 zeigt, dass tatsächlich dieses Soundscape-Objekt der Node  
8185 ist, der zum SceneGraphen hinzugefügt wird, um die raumakustischen Veränderungen  
wirksam zu machen. Neben einem parameterlosen Default-Konstruktor existiert für diese  
Klasse noch die folgende, auch im Beispiel verwendete Möglichkeit:

```
8190 Soundscape(Bounds region, AuralAttributes attributes)
```

Der erste Parameter legt mittels eines Bounds-Objektes den Raumbereich fest, innerhalb  
dessen der Klang sämtlicher darin befindlicher Schallquellen gemäß der Definitionen des  
als zweiten Parameter übergebenen AuralAttributes-Objektes verändert werden soll. Für  
diese Parameter existieren auch verschiedene Methoden, die besonders dann interessant  
8195 sind, wenn der Default-Konstruktor dieser Klasse verwendet wurde:

```
Bounds getApplicationBounds()
void setApplicationBounds(Bounds region)
```

Die Application Bounds legen wiederum den Einflußbereich fest, innerhalb dessen  
8200 die Klangveränderung wirksam sein soll, sie entsprechen also dem Parameter `region`  
des oben gezeigten Konstruktors. Diese beiden Methoden erlauben es nun, das für  
diesen Zweck aktuell verwendete Bounds-Objekt zu ermitteln oder aber ein neues  
festzulegen. Die dazu gehörenden Capability-Konstanten sind leicht an ihrem Namen zu  
erkennen, sie heißen **ALLOW\_APPLICATION\_BOUNDS\_READ** und  
8205 **ALLOW\_APPLICATION\_BOUNDS\_WRITE**.

```
AuralAttributes getAuralAttributes()
void setAuralAttributes(AuralAttributes attributes)
```

Auch auf die AuralAttributes – welches festlegt, wie der Klang innerhalb des durch  
8210 das Soundscape festgelegten Raumbereiches verändert werden soll – ist ein späterer  
Zugriff möglich. Die Methoden, die dafür zur Verfügung stehen, liefern ebenfalls wieder  
das aktuell verwendete AuralAttributes-Objekt zurück bzw. ermöglichen es, ein neues zu  
setzen. Die hierfür eventuell zu setzenden Capabilities lassen sich mit den Konstanten  
**ALLOW\_ATTRIBUTES\_READ** und **ALLOW\_ATTRIBUTES\_WRITE** festlegen.

```
8215 java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
```



`javax.media.j3d.Leaf`

8220

`javax.media.j3d.Soundscape`

Auch wenn das Verblüffen mag, aber damit ist jetzt alles bekannt, was atemberaubende Soundeffekte in einer virtuellen 3D-Welt ermöglicht. Tatsächlich erlaubt es das SceneGraph-Konzept recht simpel, die Soundnodes für vielfältigste Zwecke einzusetzen. So genügt es beispielsweise, ein Sound-Objekt in einen Teil-SceneGraphen einzuhängen, der ein bewegtes Objekt darstellt, um diesem das passende Geräusch zu verleihen. Wohin auch immer sich das Objekt bewegt, der Soundnode wird ihm folgen und der User wird eine sich verändernde Position nicht nur sehen, sondern – zumindest bei der Verwendung von Stereo-fähiger Audiohardware – auch hören, dass dieses sich bewegt und seine Position verändert. Zusammen mit den passenden Soundscape/AuralAttribute-Kombinationen verändert sich dieses Geräusch passend zur Umgebung bzw. fügt dem Sound des bewegten Objektes einen Doppler-Effekt hinzu. Das interessante dabei ist, dass all das ohne eine einzige Zeile aktiven Java-Codes möglich ist, es muß also während der Laufzeit keinerlei Aufwand an Code und Rechenzeit innerhalb der eigenen Applikation spendiert werden.

### 11.3 Sound unter Java 3D 1.3.2

Mit der neuen Version 1.3.2 von Java 3D, gibt es nennenswerte Änderungen beim Sound. Da geplant ist, die Soundarchitektur fast vollständig zu ersetzen bzw. umzuschreiben, werden bereits ab Version 1.3.2 einige möglicherweise inkompatible Änderungen vorgenommen.

So ist innerhalb eines SimpleUniverse ab dieser J3D-Version per Voreinstellung deaktiviert. Damit die bisher verwendeten Soundnodes in alt bekannter Weise tönen, ist es nötig, wie unter <http://www.javadesktop.org/forums/thread.jspa?threadID=6102&tstart=90> beschrieben, entsprechende Applikationen mit der Option

-Dj3d.audiodevice=com.sun.j3d.audioengines.javasound.JavaSoundMixer zu starten oder aber explizit ein AudioDevice3D zu erzeugen und der PhysicalEnvironment des Universe hinzuzufügen.

Wie von Seiten Suns zu hören ist, liegt die Zukunft der Audio-Architektur von Java 3D wohl in der Implementation von OAL, dem Open Audio Layer. Wenn dieser in einer der kommenden J3D-Releases verfügbar ist, dürfte durchaus mit weiteren Änderungen bei der Verwendung von Sounds zu rechnen sein.

## 12 Fortgeschrittene Techniken

8260

Dieses Kapitel soll Ihnen einen Einblick in verschiedene Klassen und Funktionalitäten bieten, die bisher nur am Rande gestreift wurden oder -da sie sich nicht in einen logischen Zusammenhang mit den gezeigten Beispielprogrammen bringen ließen – noch gar nicht erwähnt wurden und einige fortgeschrittene Techniken der 3D-Programmierung und -Darstellung beschreiben. Das soll jetzt jedoch nicht heißen, dass die bisher verschwiegenen Klassen so exotisch sind, dass sie nur seltenst zu gebrauchen wären. Vielmehr ist es so, dass vieles erst bei größeren 3D-Applikationen wirklich wichtig und von Interesse wird. Und selbst wenn sie exotisch wären – wer sagt denn, dass Sie nicht eines Tages in die Verlegenheit kommen könnten, eine Spezialapplikation zu entwickeln, die exakt diese Funktionalitäten benötigen würde? Sie wissen doch: um so spezieller eine Aufgabe ist, um so besser wird die Arbeit dafür bezahlt. Die konsequente Steigerung wäre allerdings, dass die Applikationen, die so speziell sind, das kein Mensch sie benötigt, am allerbesten bezahlt werden würden.

8265

8270

### 8275 12.1 Die Basisklasse SceneGraphObject

8280

Diese Klasse wurde bereits mehrfach erwähnt aber nie wirklich separat beschrieben. Dennoch ist sie – eben weil sie praktisch DIE Basisklasse für Java 3D ist – durchaus wichtig. Allerdings ist sie zu diesem Zeitpunkt alles andere als unbekannt, da sehr viele der bis jetzt beschriebenen Klassen Eigenschaften und Methoden vom SceneGraphObject geerbt haben.

8285

SceneGraphObject selbst ist eine abstrakte Klasse, was bedeutet, dass es wiederum nicht möglich ist, ein reines SceneGraphObject mittels einem der Konstruktoren dieser Klasse zu erzeugen. Interessantes und wichtiges findet sich jedoch bei den Methoden, von denen viele nicht mehr unbekannt sind:

```
void setCapability(int bit)
```

8290

Diese Methode ist eine der Wichtigsten überhaupt. Wie bereits eingehend beschrieben, ist es mit ihr und mit speziellen Capability-Konstanten möglich, verschiedene Fähigkeiten der jeweiligen Nodes zu markieren, so dass Java 3D weiß, dass diese so markierten Methoden auch nach einer Optimierung (sei es durch ein explizites `compile()` oder aber weil der zugehörige Node live ist) verwendbar sein müssen. Wird eine Methode, die nicht auf diese Art gekennzeichnet wurde, dennoch aufgerufen, wenn der Node live oder kompiliert ist, so würde das zu einer `RestrictedAccessException` führen.

8295

Die Klasse SceneGraphObject selbst liefert keine eigenen Capability-Konstanten mit, die einzeln als Parameter `bit` übergeben werden könnten, jedoch finden sich diese in praktisch allen von dieser Klasse abgeleiteten Nodes.

8300

Wird diese Methode aufgerufen, wenn der zugehörige Node selbst bereits live oder kompiliert ist, so führt das zu einer `RestrictedAccessException`.

```
void setCapabilityIsFrequent(int bit)
```

In Erweiterung der vorhergehend beschriebenen Methode erlaubt es diese,  
8305 Fähigkeiten nicht nur dafür zu markieren, dass sie überhaupt verwendbar bleiben sollen,  
sondern sie Teilt Java 3D darüber hinaus mit, dass die zu den Capability-Konstanten  
gehörenden Methoden und Eigenschaften besonders häufig verwendet werden. Das  
veranlasst Java 3D dazu, diese in einer wiederum anderen Art zu Optimieren um so der  
häufigen Verwendung zu entsprechen. Das Resultat ist eine noch bessere Performance  
8310 besonders bei diesen häufig verwendeten Methoden.  
Auch diese Methode darf nur dann verwendet werden, wenn der eigene Node weder live  
noch compiliert ist.

```
void clearCapability(int bit)
```

8315 Soll eine bereits gesetzte Capability wieder entfernt werden, so ist das mit dieser  
Methode möglich. Als Parameter ist wieder die Capability-Konstante anzugeben, die  
gelöscht werden soll.  
Eine Verwendung dieser Methode ist ebenfalls nicht mehr möglich und führt zu einer  
RestrictedAccessException, wenn der zugehörige Node compiliert wurde oder aber  
8320 wenn er live ist.

```
void clearCapabilityIsFrequent(int bit)
```

Auch zu `setCapabilityIsFrequent()` existiert ein Gegenstück, mit dem sich  
eine Capability löschen läßt – so fern das Objekt weder live noch compiliert ist.

8325

```
boolean getCapability(int bit)
```

Natürlich existiert auch eine Methode, mit der sich ermitteln läßt, ob eine bestimmte  
Capability bereits gesetzt wurde oder nicht. Die Konstante der Capability, die abgefragt  
werden soll, wird wiederum als Parameter übergeben. Der Rückgabewert dieser Methode  
8330 informiert dann darüber, ob diese Capability bereits gesetzt wurde (`true`) oder nicht  
(`false`).  
Auch diese Methode kann nur in einem Node aufgerufen werden, der nicht live ist und  
auch noch nicht compiliert wurde.

8335 

```
boolean getCapabilityIsFrequent(int bit)
```

Und natürlich existiert für die `isFrequent`-Capabilities eine entsprechende  
Möglichkeit, deren Status abzufragen. Die jeweilige Capability-Konstante wird wieder als  
Parameter übergeben und der Rückgabewert dieser Methode sagt aus, ob sie bereits  
gesetzt wurde.  
8340 Auch diese Methode wirft eine `RestrictedAccessException`, wenn sie in einem Node  
aufgerufen wird, der live oder compiliert ist.

```
private void duplicateSceneGraphObject(SceneGraphObject
originalNode)
```

8345 Diese Methode kopiert alle `SceneGraphObject`-Informationen vom `originalNode`  
in den eigenen Node. Diese Methode wird normalerweise von `cloneNode()` aufgerufen  
welche wiederum von `cloneTree()` aus ausgeführt wird. Es ist zu beachten, dass  
`duplicateSceneGraphObject()` selbst nie direkt von einer Applikation aufgerufen

8350 werden sollte sondern immer nur über den Weg der Methoden `cloneNode()` bzw. `cloneTree()`, deren Sinn und Zweck im folgenden in den Abschnitten zu den Klassen `Node` und `Group` noch detailliert beschrieben wird.

```
java.lang.Object getUserData()
void setUserData(java.lang.Object userData)
```

8355 `UserData` bieten die Möglichkeit, beliebige eigene Informationen an `Nodes` anhängen zu können. Mittels dieser beiden Methoden ist es nun möglich, die `UserData` eines Objektes zu holen oder aber diese überhaupt erst einmal einem `Node` hinzuzufügen.

8360 Da mit `java.lang.Object` der Java-Basisdatentyp für die `UserData`-Objekte vorgesehen ist, können diese wirklich jede erdenkliche Form annehmen. Im einfachsten Fall kann man mittels dieser Methoden also einen `String` mit Debugging-Informationen an einen `Node` anhängen bzw. sie später abfragen, es ist aber genau so gut möglich, Objekte aus beliebig komplexen, selbst definierten Klassen zu verwenden.

8365 Die `UserData` und damit diese beiden Methoden sind – eben weil sie beliebige Daten beinhalten können - nicht Teil der möglichen Optimierungsvorgänge wenn ein `Node` compiliert oder live geschaltet wird. Demzufolge können sie jederzeit uneingeschränkt benutzt werden – und es existieren auch keine `Capabilities`, die für sie gesetzt werden müßten.

8370 `boolean isCompiled()`

Der Returnwert dieser Methode sagt aus, ob der zugehörige `Node` bereits compiliert wurde (`true`) oder nicht (`false`). Auch diese Methode kann jederzeit benutzt werden, sie unterliegt keinerlei Einschränkungen durch eine Optimierung.

8375 `boolean isLive()`

Auch diese Methode kann jederzeit uneingeschränkt verwendet werden, was in der Natur ihres Zwecks liegt: sie informiert darüber, ob der aktuelle `Node` Teil eines `SceneGraphen` ist, der live ist (`true`) oder nicht (`false`).

8380 `void updateNodeReferences(NodeReferenceTable referenceTable)`

8385 Hierbei handelt es sich um eine Art Callback-Methode, welche verwendet werden kann, um zu überprüfen, ob alle von diesem `Node` referenzierten `SceneGraph`-Objekte bereits vollständig dupliziert wurden. Diese Methode wird vom bereits kurz angesprochenen `cloneTree()` aufgerufen, so bald alle `Nodes` im zugehörigen Teil-`SceneGraph` dupliziert wurden. Auch diese Methode sollte niemals direkt von einer Applikation aufgerufen werden, vielmehr sollte das der in kürze zu besprechenden Methode `cloneTree()` vorbehalten bleiben.

Wie bereits erwähnt ist `SceneGraphObject` das Basisobjekt der Java 3D API:

8390 `java.lang.Object`

## 12.2 Die Basisklasse Node

8395

Eine der direkt von `SceneGraphObject` abgeleiteten Klassen ist `Node`, das Basiselement für sehr viele weitere Klassen, die Teil eines `SceneGraphen` werden können. In dieser Klasse finden sich schließlich auch die bereits erwähnten Methoden `cloneNode()` und `cloneTree()`, die wichtig werden, wenn ein Teil-`SceneGraph` dupliziert werden soll.

8400

Auch `Node` ist eine abstrakte Klasse, schließlich würde es aber auch keinen Sinn machen, einen `Node` zu konstruieren, der innerhalb eines `SceneGraphen` keinerlei sichtbaren oder funktionalen Sinn hat. Deswegen soll auch sofort zur Betrachtung der Methoden dieser Klasse übergegangen werden:

8405

```
Node cloneNode(boolean forceDuplicate)
```

Diese Methode dupliziert den eigenen `Node` und liefert die Kopie als Returnwert zurück. Normalerweise wird diese Methode von `cloneTree()` aufgerufen, sie sollte jedoch nicht direkt aus einer Applikation heraus verwendet werden.

8410

Es ist zu beachten, dass diese Methode in allen User-definierten Subklassen unbedingt durch eine eigene Methode überschrieben werden muß. Das ist erforderlich, weil die Basisklasse ja nicht wissen kann, welche zusätzlichen Daten die Subklasse beinhaltet. Da diese Daten ebenfalls kopiert werden müssen, ist es also erforderlich, dass alle eigenen Ableitungen eine eigene `cloneNode()`-Methode mitbringen. Diese Methode sollte aus folgendem Code bestehen, wobei `UserSubClass` hier die von `Node` oder einer ihrer Subklassen abgeleitete, benutzerdefinierte Klasse ist:

8415

```
public Node cloneNode(boolean forceDuplicate)
{
 UserSubClass usc = new UserSubClass();
 usc.duplicateNode(this, forceDuplicate);
 return usc;
}
```

8420

8425

Tatsächlich ist es so, dass alle `Nodes` (oder genauer: alle von `Node` abgeleiteten Java-3D-Klassen) diese Methode überschreiben. Auch wenn es in den Methodenbeschreibungen bisher nicht erwähnt wurde, so findet sich diese Methode jedoch überall wieder. Da sie aber auch überall die gleiche Aufgabe hat – nämlich das Kopieren der klassenspezifischen Daten zu veranlassen – wurde auf ein explizites

8430

Wiederholen dieser Tatsache jedoch verzichtet.

Das Kopieren sämtlicher relevanten Daten wird dann in der ebenfalls von jeder Klasse mitzubringenden Methode `duplicateNode()` erledigt:

8435 `void duplicateNode(Node originalNode, boolean forceDuplicate)`

Hier werden nun alle Daten aus dem als Parameter übergebenen `originalNode` in das eigene Objekt kopiert. Für benutzerdefinierte Klassen, die sich von `Node` ableiten, bedeutet das, alle wichtigen (ebenfalls benutzerdefinierten) Daten innerhalb dieser Methode müssen in den eigenen `Node` kopiert werden. Diese Methode entscheidet dabei selbst, ob die Daten wirklich kopiert werden oder aber ob lediglich eine Referenz auf das Originalobjekt mitgegeben wird. Lediglich dann, wenn `forceDuplicate` auf `true` gesetzt wurde, so müssen alle Daten wirklich alle kopiert (im Extremfall also neu erzeugt und deren Inhalt kopiert) werden.

8445 Die Methode `cloneNode()` darf nur in `SceneGraphen` verwendet werden, die weder live noch kompiliert sind, andernfalls würde eine `RestrictedAccessException` geworfen werden.

`Node cloneTree()`

8450 `Node cloneTree(boolean forceDuplicate)`

`Node cloneTree(boolean forceDuplicate, boolean allowDanglingReference)`

Hiermit ist es möglich, einen kompletten Sub-`SceneGraphen` zu kopieren. Der mögliche Parameter `forceDuplicate` gibt dabei an, ob enthaltene Daten in jedem Fall kopiert werden müssen (`true`) oder ob es gegebenenfalls auch genügt, Referenzen in die Kopie des `SceneGraphen` zu übergeben (`false`).

Der ebenfalls mögliche Parameter `allowDanglingReference` bezieht sich auf ein Problem, dass beim Kopieren auftreten kann, der so genannten Dangling Reference, also einer Referenz, die gewissermaßen in der Luft hängt. Das kann passieren, wenn eine `SceneGraph` kopiert wird, der auch einen `Node` enthält, welcher wiederum einen anderen Teil-`SceneGraphen` referenziert. Nach dieser Operation würden dann zwei `Nodes` (das Original und der Kopierte) auf diesen `SceneGraphen` verweisen. Das wird dann ein Problem, wenn es genau das nicht zulässig ist. Die Methode `cloneTree()` handhabt so etwas auf zwei Arten. Ist `allowDanglingReference` `false`, so wird eine

8465 `DanglingReferenceException` geworfen, die gegebenenfalls abgefangen werden kann. Wurde dieser Parameter hingegen auf `true` gesetzt, so liefert die Methode `updateNodeReferences()` eine Referenz auf das Objekt zurück und das Ergebnis der `cloneTree()`-Operation ist ein `SceneGraph` mit eben jenen Dangling References.

8470 Wird `cloneTree()` innerhalb eines `SceneGraphen` aufgerufen, der live ist oder bereits kompiliert wurde, so ist das Resultat eine `RestrictedAccessException`.

`Node cloneTree(NodeReferenceTable referenceTable)`

`Node cloneTree(NodeReferenceTable referenceTable, boolean forceDuplicate)`

8475 `Node cloneTree(NodeReferenceTable referenceTable, boolean forceDuplicate, boolean allowDanglingReferences)`

Auch diese Methoden bewirken, dass eine Kopie des aktuellen Teil-`SceneGraphen` erzeugt und zurückgeliefert wird. In der `referenceTable` werden Informationen über das Cloning gespeichert, diese enthält anschließend das Mapping von Originalnodes zu

8480 den daraus erzeugten Kopien.  
Der Parameter `forceDuplicate` gibt wiederum an, ob Referenzen zu Objekten erlaubt sind oder nicht und `allowDanglingReferences` legt fest, ob die so genannten Dangling References zulässig sind oder ob eine `DanglingReferenceException` geworfen werden soll, wenn so etwas auftritt.

8485 Die Methode `cloneTree()` darf nur in SceneGraphen verwendet werden, die weder live noch kompiliert sind, andernfalls würde eine `RestrictedAccessException` geworfen werden.

```
Bounds getBounds()
```

8490 Diese Methode liefert das Bounds-Objekt für diesen Node zurück. In der Regel handelt es sich bei dem zurückgelieferten Objekt – so fern die Bounds automatisch erzeugt und nicht explizit gesetzt wurden – um eine `BoundingSphere`, lediglich Geometry-Objekte liefern ein wesentlich genaueres `BoundingBox`-Objekt zurück.  
Die hierzu gehörende Capability-Konstante ist **ALLOW\_BOUNDS\_READ**.

8495

```
void setBounds(Bounds bounds)
```

Ein Bounds-Objekt kann auch explizit an einen Node übergeben werden. Das ist mittels dieser Methode möglich, wobei die Capability **ALLOW\_BOUNDS\_WRITE** gesetzt sein muß, wenn das auch in kompiliertem Zustand oder wenn der Node live ist, möglich sein soll.

8500

```
boolean getBoundsAutoCompute()
```

```
void setBoundsAutoCompute(boolean autoCompute)
```

Die Bounds eines Nodes können automatisch berechnet werden, was per Voreinstellung immer geschieht. Mittels dieser beiden Methoden ist es nun möglich zu ermitteln, ob dieser Modus aktiviert ist oder aber er kann aktiviert (`autoCompute=true`) oder deaktiviert (`autoCompute=false`) werden. Die für diese Methoden eventuell benötigten Capability-Konstanten heißen **ALLOW\_AUTO\_COMPUTE\_BOUNDS\_READ** und **ALLOW\_AUTO\_COMPUTE\_BOUNDS\_WRITE**.

8505

8510

```
boolean getCollidable()
```

```
void setCollidable(boolean collidable)
```

Ob ein Node sowie seine eventuell vorhandenen Children (so es denn ein Group-Node ist) für Kollisionen bzw. deren Feststellung verwendet werden kann, wird mittels des `collidable`-Flags festgelegt. Diese Methoden erlauben es, dessen aktuellen Wert zu ermitteln oder aber ihn neu zu setzen. Sollen sie auch in kompiliertem Zustand oder aber wenn der Node live ist verwendet werden können, so sind die Capabilities **ALLOW\_COLLIDABLE\_READ** bzw. **ALLOW\_COLLIDABLE\_WRITE** zu setzen. Soll dieser Node nach einer Kollision innerhalb eines daraus resultierenden SceneGraphPath-Objektes aufgeführt werden, so ist das mit Hilfe der Capability **ENABLE\_COLLISION\_REPORTING** festzulegen.

8515

8520

```
boolean getPickable()
```

```
void setPickable(boolean pickable)
```

8525 Ähnlich dem Flag `collidable` existiert mit `pickable` eines, das festlegt, ob ein Node mittels Picking aufgefunden werden kann oder nicht. Dieses Flag kann mit Hilfe der ersten Methode abgefragt werden, wo hingegen die zweite Methode ihm einen neuen Wert zuweist. Die zu diesen Methoden gehörenden Capability-Konstanten sind **ALLOW\_PICKABLE\_READ** und **ALLOW\_PICKABLE\_WRITE**. In diesem

8530 Zusammenhang ebenso zu erwähnen wäre **ENABLE\_PICK\_REPORTING**, dieses Capability legt fest, dass der Node innerhalb eines `SceneGraphPath`-Objektes aufgeführt wird, der das Ergebnis einer Picking-Operation ist.

```
Node getParent()
```

8535 Es wird das Elternobjekt des aktuellen Nodes zurückgeliefert. Diese Methode kann nur während der Konstruktion eines SceneGraphen verwendet werden. Ist dieser einmal live oder ist er kompiliert worden, so würde die Verwendung von `getParent()` eine `RestrictedAccessException` auslösen.

8540 Wie bereits erwähnt ist die Klasse `Node` ein direkter Abkömmling von `SceneGraphObject`:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
```

8545

## 12.3 Die Klasse Group

Eine der direkt von `Node` abgeleiteten Klassen ist `Group`, deren nahe Verwandte `BranchGroup` und `TransformGroup` bereits gut bekannt sind. Eine gemeinsame

8550 Eigenschaft von `Group`-Nodes ist, dass sie mehrere Children verwalten können, unterhalb einer `Group` kann im `SceneGraph` also verzweigt werden.

Da `Group` erstmalig keine abstrakte Klasse ist, existiert auch ein verwendbarer Konstruktor, zu dem es allerdings nicht viel zu sagen gibt:

8555

```
Group()
```

Interessanter wird es dann schon bei den Methoden, die sich zwangsläufig auch in allen von `Group` abgeleiteten Nodes wiederfinden:

8560

```
void addChild(Node child)
```

Diese Methode ist bereits bekannt, sie erlaubt es, der `Group` ein neues Child hinzuzufügen. In kompilierten `SceneGraph`en oder solchen, die live sind, ist das nur möglich, wenn zuvor die Capability **ALLOW\_CHILDREN\_EXTEND** gesetzt wurde.



8565

```
java.util Enumeration getAllChildren()
```

Der Returnwert dieser Methode ist eine Enumeration, die alle Children des aktuellen Group-Objektes enthält. Mit dieser Methode steht die Capability-Konstante **ALLOW\_CHILDREN\_READ** in Zusammenhang.

8570

```
boolean getAlternateCollisionTarget()
```

```
void setAlternateCollisionTarget(boolean target)
```

Wird eine Group als alternatives Kollisionsziel verwendet, so bedeutet das, dass sie als in ein Kollisionsereignis involviert gemeldet wird, wenn einer ihrer Children in eben  
8575 eine Kollision verwickelt ist. Voreingestellt ist hier der Wert false, was bedeutet, dass wirklich nur die Nodes als kollidiert gemeldet werden, die auch wirklich etwas mit der Kollision zu schaffen haben. Wird die Group als alternatives Ziel verwendet, so wird eine eventuell (nicht) gesetzte **ENABLE\_COLLISION\_REPORTING**-Capability ignoriert, da das Collision-Reporting in diesem Falle implizit aktiviert wird.

8580 Die beiden obigen Methoden erlauben es nun, den aktuellen Zustand in Bezug auf das alternative Kollisionsziel abzufragen oder aber diese Group als solches zu verwenden (target=true) oder die Verwendung wieder zu deaktivieren (target=false).

```
Node getChild(int index)
```

8585 Es wird das durch index spezifizierte Child dieser Group zurückgeliefert. Für diese Methode muß nötigenfalls die Capability **ALLOW\_CHILDREN\_READ** gesetzt sein.

```
int indexOfChild(Node child)
```

Ein solcher Indexwert läßt sich beispielsweise mit dieser Methode ermitteln. Sie  
8590 liefert die Positionsinformation zurück, die angibt, an welcher Stelle innerhalb der Group-eigenen Childliste sich der Node child befindet.

```
void insertChild(Node child, int index)
```

```
void setChild(Node child, int index)
```

8595 Ähnliche Funktionalitäten wie die von diesen Methode bereitgestellten sind ebenfalls bereits bekannt. Sie fügt den Node child an der Position index in die Childliste der Group ein und verschieben alle Nodes von dieser und den nachfolgenden Positionen um eins nach hinten bzw. überschreiben den dort befindlichen Node mit dem neuen child. Die zugehörige Capability-Konstante ist **ALLOW\_CHILDREN\_WRITE**.

8600

```
void removeAllChildren()
```

```
void removeChild(int index)
```

```
void removeChild(Node child)
```

Diese Methoden kümmern sich um das entfernen von bereits vorhandenen Children  
8605 innerhalb der Group. Damit ist es möglich, entweder alle oder das durch index bzw. child spezifizierte Node-Objekt aus der Group-internen Childliste herauszunehmen.

```
int numChildren()
```

8610 Der Zahlenwert, der von dieser Methode zurückgeliefert wird, sagt aus, wie viele Nodes sich derzeit insgesamt unterhalb dieser Group befinden, d.h. wie viele Children diese hat.

```
Bounds getCollisionBounds()
```

```
void setCollisionBounds(Bounds bounds)
```

8615 Collision Bounds sind optionale Bounds, die nur für die Feststellung von Kollisionen verwendet werden können. Es kann durchaus Sinn machen, dafür größere Bounds zu definieren, wenn eine Berührung beispielsweise bereits vor einer echten Kollision mit den Geometriedaten des zugehörigen SceneGraphen festgestellt werden soll. Mittels dieser beiden Methoden ist es nun möglich, die aktuell verwendeten Bounds zu holen oder aber neue festzulegen. In compilierten oder live geschalteten SceneGraphen können sie nur noch dann verwendet werden, wenn zuvor die Capabilities **ALLOW\_COLLISION\_BOUNDS\_READ** bzw. **ALLOW\_COLLISION\_BOUNDS\_WRITE** gesetzt wurden.

8620

8625 Wie zu erwarten und wie auch in der Einleitung dieses Abschnittes bereits gesagt, leitet sich die Group direkt von der Klasse Node ab:

```
java.lang.Object
```

```
 javax.media.j3d.SceneGraphObject
```

8630 

```
 javax.media.j3d.Node
```

```
 javax.media.j3d.Group
```

## 12.4 Die Switch-Group

8635 Nach dem jetzt mit der BranchGroup und der TransformGroup sowie ihrer Basisklasse Group selbst die wichtigsten Group-Nodes bekannt sind, soll noch auf eine weitere Klasse eingegangen werden, die für viele Anwendungszwecke wirklich wichtig ist. Wie es der Name bereits andeutet, ist die Klasse Switch in der Lage, etwas zu schalten. Da es sich um eine Group handelt, ist anzunehmen, dass sie ihre Child-Nodes an- und abschalten kann. Diese Annahme ist richtig, mit der Switch-Group ist es möglich, Nodes bzw. ganze Teil-SceneGraphen aus einer Szene zu entfernen, ohne diese auch aus dem SceneGraphen entfernen zu müssen. Neben dem Default-Konstruktor, der ein Switch-Objekt erzeugt, bei dem alle Children voreingestellt deaktiviert (also in der Szene nicht sichtbar) sind, findet sich auch dieser hier:

8640

8645

```
Switch(int whichChild, java.util.BitSet childMask)
```

Eine Switch-Group enthält eine geordnete Liste von Children, die an Hand ihrer

Indexnummer modifiziert werden können. Dabei ist zu beachten, dass die Reihenfolge dieser Children innerhalb der Liste nichts mit der Reihenfolge zu tun hat, in der sie gezeichnet werden. Was nun mit welchem Child getan werden soll, wird mit den Parametern dieses Konstruktors beeinflusst. `whichChild` gibt an, was getan werden soll. Das passiert mit verschiedenen vorgegebenen Konstanten. **CHILD\_NONE** gibt dabei an, dass keines der Children aktiviert sein soll, **CHILD\_ALL** legt fest, dass sie alle in der Szene sichtbar sein sollen. Für den oben gezeigten Konstruktor ist aber eigentlich nur die Verwendung von **CHILD\_MASK** sinnvoll, denn dieser Wert legt fest, dass mit dem folgenden Parameter `childMask` angegeben wird, welcher der Children – die durch ihre Indexnummer innerhalb der geordneten Switch-eigenen Childliste bestimmt werden – aktiviert und welche deaktiviert sein sollen. Sollen alle Children bei der Konstruktion des Switch-Objektes den gleichen Zustand erhalten, also soll eine der Konstanten **CHILD\_ALL** oder **CHILD\_NONE** verwendet werden, so wäre es sinnvoller, diesen Konstruktor einzusetzen:

```
Switch(int whichChild)
```

8665

Natürlich gibt es auch für die Switch-Group verschiedene Methoden, die es unter anderem auch ermöglichen, die Children dynamisch und während der Programmlaufzeit zu beeinflussen:

```
8670 int getWhichChild()
void setWhichChild(int child)
```

Diese Methoden beziehen sich auf den Modus, in dem die Switch-Group arbeiten soll. Dieser wird mit Hilfe der Konstanten **CHILD\_ALL**, **CHILD\_NONE** und **CHILD\_MASK** angegeben und kann von diesen Methoden ermittelt oder aber auf einen neuen Wert gesetzt werden.

8675

```
java.util.BitSet getChildMask()
void setChildMask(java.util.BitSet childMask)
```

Wurde für `whichChild` der Wert **CHILD\_MASK** angegeben, so ist es möglich, Children dieser Group einzeln zu beeinflussen. Das passiert mit Hilfe eines `java.util.BitSet`-Objektes. Dabei wird der aktuelle Zustand von der Methode `getChildMask()` zurückgeliefert, während es mit `setChildMask()` und einem als Parameter übergebenen `BitSet` möglich ist, den Aktivierungszustand der Children zu ändern. Ein Bit, das auf 1 (bzw. `true`) gesetzt wird, steht dabei für ein sichtbares Child, während der wert 0 (bzw. `false`) das an der zugehörigen Indexposition befindliche Child deaktiviert, also innerhalb der Szene unsichtbar macht.

Für diese beiden Methoden existieren die Capability-Konstanten **ALLOW\_SWITCH\_READ** und **ALLOW\_SWITCH\_WRITE**, die gegebenenfalls gesetzt werden müssen.

8690

Da diese Klasse nicht sehr komplex, aber elementar wichtig ist (das manuelle Entfernen und Wiedereinfügen von Teil-SceneGraphen wäre wesentlich aufwändiger und würde mehr Performance kosten), kann ihre Beschreibung an dieser Stelle mit dem alt

8695 gewohnten Blick auf die Ableitung bereits beendet werden. Die direkte Verwandtschaft mit der Klasse Group war hier eigentlich zu erwarten:

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Group
 javax.media.j3d.Switch
```

## 12.5 AlternateAppearance

8705 Eine der exotischeren Klassen der Java 3D API ist sicher die AlternateAppearance. Wenn Sie hier die verschiedenen Dokumente bemühen, die es zu Java 3D gibt, werden Sie auch nicht unbedingt herausfinden, wozu diese Klasse dient, da die offizielle Aussage lediglich darin besteht, dass sie dazu verwendet werden kann, um die Appearance ausgewählter Nodes zu überschreiben („The AlternateAppearance leaf node is used for overriding the  
8710 Appearance component of selected nodes“). Die Kern-Aussage ist hier tatsächlich das „selected“. Dieses ist jedoch so zu interpretieren, dass es sich um explizit ausgewählte Nodes handelt, und nicht nur um „bestimmte Nodes“. Mit anderen Worten gesagt, soll die AlternateAppearance dazu dienen, eine Art Highlighting, also eine Hervorhebung bestimmter, durchaus auch vom Benutzer ausgewählter Nodes zu realisieren.

8715 Interessant ist einer der Konstruktoren der Klasse AlternateAppearance:

```
AlternateAppearance(Appearance appearance)
```

8720 Dieser erwartet als Parameter ein Appearance-Objekt, dessen Daten verwendet werden, um die Eigenschaften der AlternateAppearance in Bezug auf das 3D-Objekt festzulegen. Und auch bei den Methoden ist zu sehen, dass die AlternateAppearance nur eine Art Container für das eigentliche Appearance-Objekt darstellt, es finden sich hier keine Möglichkeiten, die aus der Klasse Appearance her bekannten Attributes direkt zu setzen.

8725 Die AlternateAppearance selber wird also nicht nur einem einzelnen 3D-Objekt hinzugefügt, sondern in alt bekannter Weise als Node in den SceneGraphen eingehängt und wirkt dann auf alle Shape3D- und Morph-Nodes, die sich innerhalb ihres Einflußbereiches befinden bzw. auf diejenigen Objekte, die sich innerhalb dieser  
8730 Influencing Bounds befinden und die als Scope definiert wurden. Damit die Appearance eines Objektes jedoch wirklich durch diese AlternateAppearance überschrieben werden kann, muß eine weitere Bedingung erfüllt sein: das Überschreiben muß für die betroffenen Shape3D- oder Morph-Nodes mittels eines Aufrufes von `setAppearanceOverrideEnable(true)` aktiviert werden.

8735

Die AlternateAppearance selbst bringt verschiedene Methoden mit, von denen sich auch wieder ein ganzer Block mit dem Scope beschäftigt, einer Festlegung also, auf welche Elemente eines SceneGraphs sich die AlternateAppearance – neben ihren InfluencingBounds - beziehen soll:

8740

```
void addScope(Group scope)
```

Es wird ein neues Element als Scope definiert und zur Klassen-internen Scopeliste hinzugefügt. Die zu dieser Methode gehörende Capability-Konstante ist **ALLOW\_SCOPE\_WRITE**.

8745

```
Group getScope(int index)
```

Es wird das Objekt zurückgeliefert, das sich an der durch index spezifizierten Position in der Scopeliste befindet. Die korrespondierende Capability-Konstante ist **ALLOW\_SCOPE\_READ**.

8750

```
int indexOfScope(Group scope)
```

Es wird die Position zurückgeliefert, an der sich das Objekt `scope` in der Scopeliste befindet.

8755 

```
void insertScope(Group scope, int index)
```

Diese Methode fügt ein neues Group-Objekt in die klasseninterne Scopeliste an der Position `index` ein. Befindet sich dort bereits eine Node, so wird dieser und alle nachfolgenden um eine Position nach hinten verschoben. Für die Verwendung dieser Methode ist unter Umständen das **ALLOW\_SCOPE\_WRITE**-Capability erforderlich.

8760

```
int numScopes()
```

Der Rückgabewert dieser Methode ist die Gesamtanzahl aller bereits als Scope definierten Nodes. Dieser Wert ist 0, wenn zuvor die folgende Methode aufgerufen wurde:

8765 

```
void removeAllScopes()
```

Es werden alle Nodes aus der AlternateAppearance-internen Scopeliste entfernt. Anschließend legen nur noch die Influencing Bounds fest, auf welche Objekte die AlternateAppearance wirken soll.

8770 

```
void removeScope(Group scope)
void removeScope(int index)
```

Diese Methoden entfernen jeweils ein Element aus der Scopeliste. Welches Element entfernt werden soll wird dadurch festgelegt, dass dieser Node selber als Parameter übergeben wird oder aber durch Angabe seiner Position `index` innerhalb der Scopeliste.

8775

```
void setScope(Group scope, int index)
```

Es wird ein neuer Node als Scope definiert und an die Position `index` innerhalb der `AlternateAppearance`-internen Scopeliste gesetzt. Der Unterschied zu `insertScope()` ist, dass Objekte, die sich eventuell bereits an dieser Position befinden, durch das neue scope-Objekt überschrieben werden. Die zu dieser Methode gehörende Capability-Konstante ist wiederum **ALLOW\_SCOPE\_WRITE**.

```
java.util Enumeration getAllScopes()
```

Diese Methode liefert den gesamten Inhalt der klasseninternen Scopeliste zurück. Die einzelnen Nodes sind dabei in dem zurückgegebenen Enumeration-Objekt zu finden. Auch hierfür ist es nötig, die Capability **ALLOW\_SCOPE\_READ** zu setzen, wenn diese Methode in einem compilierten `SceneGraph` verwendet werden soll oder in einem, der live ist.

```
Appearance getAppearance()
```

```
void setAppearance(Appearance appearance)
```

Diese Methoden beziehen sich auf die eigentliche Eigenschaft der Klasse, auf die `Appearance`. Sie ermöglichen es, die aktuell verwendete `Appearance` zu ermitteln oder aber eine neue festzulegen. Die dafür unter Umständen zu setzenden Capabilities sind **ALLOW\_APPEARANCE\_READ** und **ALLOW\_APPEARANCE\_WRITE**.

```
Bounds getInfluencingBounds()
```

```
void setInfluencingBounds(Bounds region)
```

Neben dem Scope legen die `Influencing Bounds` fest, in wie weit die `AlternateAppearance` Einfluß nehmen soll- ist kein Scope definiert, so legen ausschließlich diese `Bounds` den Einflußbereich fest. Mittels dieser Methoden ist es nun möglich, den aktuell spezifizierten Einflußbereich zu ermitteln oder aber einen neuen festzulegen. Die zugehörigen Capability-Konstanten sind hier

**ALLOW\_INFLUENCING\_BOUNDS\_READ** und **ALLOW\_INFLUENCING\_BOUNDS\_WRITE**.

Interessanterweise leitet sich die Klasse `AlternateAppearance` nicht von `Appearance` ab, was allerdings nicht zu erstaunlich ist, wenn man bedenkt, das sie gewissermaßen nur als Container dient:

```
java.lang.Object
```

```
 javax.media.j3d.SceneGraphObject
```

```
 javax.media.j3d.Node
```

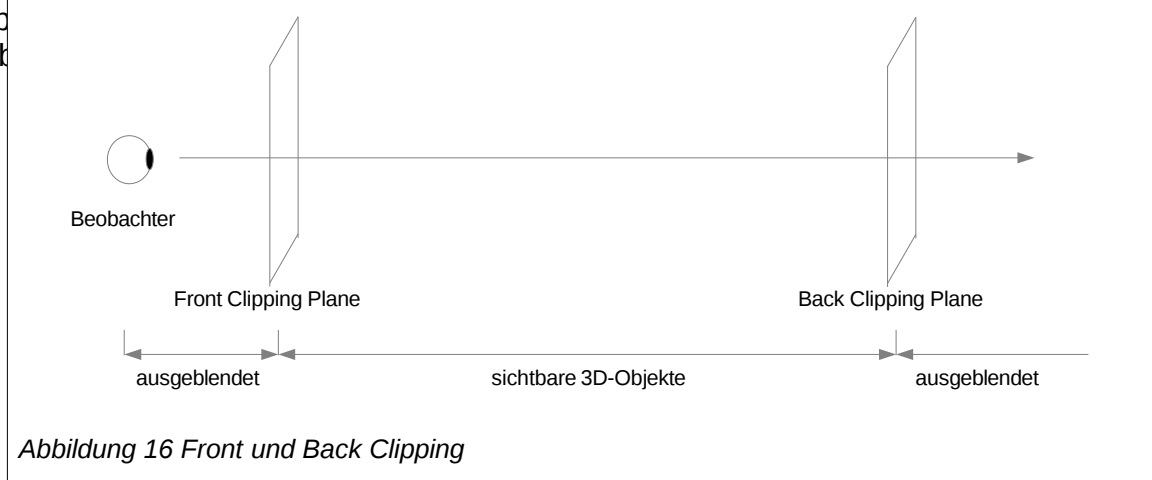
```
8815 javax.media.j3d.Leaf
```

```
 javax.media.j3d.AlternateAppearance
```

## 12.6 Front- und Backclipping

- 8820 Eine wichtige Technik bei 3D-Applikationen, die bisher nur kurz angesprochen wurde, ist das Clipping. Vereinfacht gesagt, bewirkt das Clipping, dass alle 3D-Objekte jenseits spezieller Clipping Planes nicht mehr dargestellt werden. Das ist erforderlich, um bei komplexen Szenen die Performance nicht einbrechen zu lassen. Mit anderen Worten gesagt, wird mit dem Clipping (zumindest mit der Back-Clipping Plane) eine Art künstlicher Horizont erzeugt, jenseits dessen nichts mehr zu sehen ist – mit Ausnahme des Hintergrundes natürlich.
- 8825

- Für Java 3D stellt es sich so dar, dass hier zwei Clipping Planes existieren: die bereits angesprochene für den Hintergrund, die alles „abschneidet“, was eine gewisse Entfernung überschreitet, sowie eine weitere Front-Clipping Plane, die all die Teile von 3D-Objekten
- 8830
- ausblendet
- Beobachter



- 8835 Dieser Aufbau könnte Sie in Versuchung führen, für die Back Clipping Distance einen Wert zu wählen, der für die Szene sinnvoll ist und für die Front Clipping Distance einen sehr kleinen Wert einzusetzen um möglichst alles, was sich dicht am Beobachter befindet, sichtbar werden zu lassen. Das ist jedoch absolut nicht empfehlenswert, da das zu Problemen mit dem Z-Buffer, also mit der Verwaltung der Tiefeninformation innerhalb der
- 8840 Grafikkarte kommen kann. Dieser Z-Buffer verwendet im allgemeinen nur 16- oder 32-Bit-Ganzzahlen, nur ganz wenige, professionelle Grafikkarten setzen hier Fließpunktzahlen ein. Die Hersteller von Consumer-Grafikkarten protzen scheinbar lieber mit vergleichsweise weniger wichtigen Details wie die Größe des Grafikspeichers, der Anzahl der Grafikpipelines oder anderem, statt hier mit einem Fließpunkt-Z-Buffer ein wirklich sinnvolles Feature einzuführen. Diese Beschränkung auf Ganzzahlen bedeutet
- 8845 jedoch, dass die Genauigkeit des Z-Buffers begrenzt und abhängig vom Verhältnis zwischen Front und Back Clipping Distance ist.
- Um so größer dieses Verhältnis zwischen diesen beiden Werten ist, um so ungenauer arbeitet der Z-Buffer, was sich in hässlichen Treppenstufen in den dargestellten Objekten äußert. In gewissen Grenzen lässt sich dieses Problem umgehen, indem von einem 16-Bit-Z-Buffer auf 32 Bit umgeschaltet wird (was sich für viele Grafikkarten einstellen lässt),
- 8850 aber auch das wird irgendwann an seine Grenzen stoßen. Die einzige Möglichkeit besteht darin, das Verhältnis  $r$  zwischen Front und Back Clipping Distance auf einem Wert kleiner als 1000 für einen 16-Bit-Buffer und kleiner als ca. 5000 bei einem 32-Bit-Buffer

8855 gehalten wird, wobei sich „r“ aus der Front Clipping Distance „front“ und der Back Clipping Distance „back“ folgendermaßen berechnet:

$$r = \frac{\text{back}}{\text{front}}$$

8860 Im schlimmsten Fall sollten Sie also davon ausgehen, dass der Z-Buffer bei einem Enduser nur mit 16 Bit Präzision arbeitet, was bedeutet, dass r kleiner als 1000 sein sollte. Das würde bedeuten, dass die Front Clipping Plane bei einer Back Clipping Distance von 200 m in einer Entfernung von 20 cm gesetzt werden müßte – für eine freie Navigation innerhalb einer virtuellen Welt ist dieser Wert nicht gerade klein, da sämtliche  
8865 Bewegungen in einer Entfernung > 20 cm vor einem 3D-Objekt enden müßten, damit der Benutzer von diesem Clipping nichts mehr bemerkt.

Doch wie sind nun die Werte für die beiden Clipping Planes zu setzen? Bei der Verwendung eines SimpleUniverse u, wie es in den obigen Beispielprogrammen zum  
8870 Einsatz kam, würden die folgenden Methodenaufrufe zum Einsatz kommen:

```
u.getViewer().getView().setBackClipDistance(double d)
```

Auf einigen Umwegen wird mittels der Methode `setBackClipDistance()` die Entfernung zur Back Clipping Plane festgelegt. Diese Entfernung d wird dabei in der  
8875 Einheit Meter festgelegt. Die Methode selbst findet sich in der Klasse View, auf die hier über den Umweg der Klasse Viewer zugegriffen wird.

```
u.getViewer().getView().setFrontClipDistance(double d)
```

Die Entfernung d zur Front Clipping Plane wiederum wird mit dieser Methode  
8880 ebenfalls in der Einheit Meter festgelegt. Der Zugriff auf den Viewer des SimpleUniverse u erfolgt wieder mittels des Methodenaufrufes `getViewer()` während dieser die Methode `getView()` zur Verfügung stellt, die letztendlich Zugriff auf das zugehörige View-Objekt und damit auf die Methode `setFrontClipDistance()` erlaubt.

8885 In der Klasse View selbst finden sich unter anderem auch Methoden, mit denen sich die aktuellen Werte ermitteln lassen:

```
double getBackClipDistance()
```

Diese Methode liefert den aktuellen Abstand zur Back Clipping Plane zurück.

8890

```
double getFrontClipDistance()
```

Der Rückgabewert dieser Methode ist wiederum die Front Clipping Distance, also der Abstand zur vorderen Clipping Plane.



## 8895 12.7 Das Canvas3D

Die Schnittstelle zwischen der zweidimensionalen Welt der grafischen Benutzeroberfläche und der dreidimensionalen Welt von Java 3D wird durch ein Objekt gebildet, das bereits intensiv verwendet wurde, bisher aber noch wenig Beachtung gefunden hat: das  
8900 Canvas3D. Dabei handelt es sich – frei übersetzt – um eine Art Leinwand, auf der das 3D-Geschehen dargestellt wird und die in eine Programmoberfläche eingebunden werden kann. Das Canvas3D ist dabei ähnlich wie die AWT-Komponenten ein Heavyweight-Objekt, das Schwierigkeiten bereiten kann, wenn es in eine Oberfläche aus Lightweight-Komponenten eingebunden werden soll. Doch dazu in Kürze mehr.

8905

Bisher wurde immer folgender Konstruktor verwendet, um ein Canvas3D-Objekt zu erzeugen:

```
Canvas3D(java.awt.GraphicsConfiguration graphicsConfiguration)
```

8910

Als Parameter wurde dabei lediglich ein GraphicsConfiguration-Objekt aus dem Paket `java.awt` verwendet, das detaillierte Informationen über das zu verwendende Grafikausgabegerät beinhaltet. Der einfachste Weg, diese Informationen für das aktuell verwendete Gerät (das in der Regel der Monitor sein dürfte) zu erhalten, ist mittels der  
8915 statischen Methode `getPreferredConfiguration()` des `SimpleUniverse`-Objektes. Bis vor kurzem war es unter Umständen auch möglich, hier `null` zu übergeben, seit Java 3D 1.3 ist das jedoch nicht mehr erlaubt. Aus Kompatibilitätsgründen funktioniert das zwar noch wie in den vorangegangenen Versionen, das kann sich aber in Zukunft ändern.

### 8920 12.7.1 Off-Screen Rendering

Der zweite mögliche Konstruktor für ein Canvas3D-Objekt erlaubt einen anderen Arbeitsmodus. Statt das Canvas3D wie bisher sichtbar zu machen und die Szene direkt  
8925 anzuzeigen, ist es im Modus für das off-screen Rendering möglich, das Bild, das das Canvas3D sonst direkt darstellen würde, in Form von Imagedaten zu holen und weiterzuverarbeiten:

```
Canvas3D(java.awt.GraphicsConfiguration graphicsConfiguration,
boolean offScreen)
```

8930

Wird der hier zusätzliche Parameter `offScreen` auf `true` gesetzt und das Canvas3D-Objekt nicht sichtbar gemacht, in dem es der Oberfläche einer Applikation hinzugefügt wird, so arbeitet dieses im off-Screen-Modus. In diesem Modus sind die folgenden  
8935 Methoden von Interesse, die es in der hier gezeigten Reihenfolge erlauben, die Szene off-Screen in ein Image zu rendern:

```
void setOffScreenBuffer(ImageComponent2D buffer)
```

Es wird ein `ImageComponent2D`-Objekt gesetzt, in welches die Imagedaten aus dem off-Screen-Renderingprozess gespeichert werden sollen. Dieses Objekt sollte also  
8940 unbedingt die gleiche Breite und höhe aufweisen, wie das `Canvas3D`.  
Befindet sich das `Canvas3D`-Objekt nicht im off-Screen-Modus, so führt das zu einer `IllegalStateException`.

```
void renderOffScreenBuffer()
```

8945 Diese Methode startet die Berechnung eines Einzelbildes, wobei die Imagedaten in den `offScreenBuffer` aus dem vorhergehenden Methodenaufruf gespeichert werden. Diese Methode arbeitet asynchron, das heißt, sie kehrt sofort zurück, auch wenn der Rendering-Vorgang für das aktuelle Frame noch nicht beendet wurde.  
Auch hierfür ist es erforderlich, dass sich das `Canvas3D` im off-Screen-Modus befindet, da  
8950 anderenfalls wieder eine `IllegalStateException` geworfen wird.

```
void waitForOffScreenRendering()
```

Mittels dieser Methode wird das Ende eines Rendering-Vorganges von einem vorhergehend aufgerufenen `renderOffScreenBuffer()` abgewartet. Wenn diese Methode  
8955 abgearbeitet wurde, so sind die Imagedaten vollständig vorhanden und können weiterverwendet werden.  
Auch hier wird eine `IllegalStateException` geworfen, sollte sich das `Canvas3D`-Objekt nicht im off-Screen-Modus befinden.

```
8960 ImageComponent2D getOffScreenBuffer()
```

Nach einem vollständigen off-Screen-Renderingvorgang liefert diese Methode ein `ImageComponent2D`-Objekt zurück, das die Imagedaten des eben gerenderten Frames beinhaltet. Dessen Methode `getImage()` wiederum gibt als Returnwert ein  
`java.awt.image.BufferedImage` zurück, das anschließend weiterverarbeitet oder  
8965 gespeichert werden kann.

Neben diesen Methoden, die für die Abarbeitung eines off-Screen-Renderzyklus nötig sind, gibt es noch einige, die eher informativen Charakter haben:

```
8970 boolean isOffScreen()
```

Ist der Rückgabewert dieser Methode `true`, so befindet sich das `Canvas3D` im off-Screen-Modus, anderenfalls wird `false` zurückgegeben.

```
boolean isRendererRunning()
```

8975 Diese Methode sagt aus, ob der Renderer gerade arbeitet oder nicht. Wurde die Methode `renderOffScreenBuffer()` aufgerufen und wird das aktuelle Frame noch gezeichnet, so ist der Rückgabewert `true`. Nur wenn dieser Vorgang abgeschlossen ist – also wenn auch die Methode `waitForOffScreenRendering()` beendet ist, so ist der Returnwert `false`.

8980

## 12.7.2 Stereoskopie

8985 Eine weitere hochinteressante Fähigkeit von Java 3D lässt sich ebenfalls über das Canvas3D beeinflussen: die Möglichkeit, stereoskopische Ausgaben zu erzeugen, die bei der Verwendung eines geeigneten Ausgabegerätes (also beispielsweise ein VR-Helm mit zwei unabhängig voneinander ansteuerbaren Displays für jedes Auge oder eine Shutterbrille) einen echten, räumlichen Eindruck hinterlassen.

```
boolean getStereoAvailable()
```

8990 Der Rückgabewert dieser Methode gibt Auskunft darüber, ob das aktuelle verwendete Grafikgerät überhaupt in der Lage ist, Stereobilder zu erzeugen, die zu einer räumlichen Darstellung nötig sind, oder nicht.

```
boolean getStereoEnable()
```

8995 Diese Methode sagt aus, ob das Canvas3D stereoskopische Ausgaben erzeugt oder nicht. Voraussetzung dafür, dass diese Methode `true` zurückliefert, ist nicht nur, dass dieser Modus mittels `setStereoEnable(true)` aktiviert wurde, sondern natürlich auch, dass das aktuell verwendete Grafikausgabegerät Stereobilder unterstützt.

```
9000 void setStereoEnable(boolean flag)
```

Mittels diese Methode ist es möglich, den Stereomode zu aktivieren (`true`) oder ihn wieder zu deaktivieren (`false`). Wird der Stereomodus aktiviert, obwohl kein Grafikausgabegerät angeschlossen ist, dass diesen unterstützt, so passiert schlichtweg nichts, das angezeigte Ergebnis ist nach wie vor monoskopisch.

```
9005 int getMonoscopicViewPolicy()
```

```
void setMonoscopicViewPolicy(int policy)
```

Die View-Policy für den monoskopischen Ausgabemodus legt fest, welchen Blickwinkel das aktuelle Canvas3D darstellen soll. Die Konstanten, die diesen Modus näher spezifizieren, finden sich dabei in der Klasse `View`. **CYCLOPEAN\_EYE\_VIEW** (oder exakter `View.CYCLOPEAN_EYE_VIEW`) ist der voreingestellte Wert, er zeigt den Betrachter die Szene aus einem Blickwinkel, wie ihn ein Zyklop mit nur einem Auge hätte. Dieser Wert ist für den monoskopischen Modus der sinnvollste. Die Konstanten **LEFT\_EYE\_VIEW** und **RIGHT\_EYE\_VIEW** hingegen legen fest, dass der berechnete Blickwinkel dem Blick aus dem linken bzw. dem rechten Auge entspricht, das heißt, diese sind in der horizontalen um ein paar Zentimeter nach links bzw. rechts versetzt. Aber was kann man nun damit anfangen? Neben der Möglichkeit, zwei Canvas3D zu erzeugen und diese auf zwei unterschiedlichen Ausgabegeräten anzuzeigen, die zur Erstellung von Hologrammen oder aber für die beiden bereits erwähnten Displays in einem VR-Helm verwendet werden, bleiben der Fantasie mit diesen Features praktisch keine Grenzen gesetzt. Sie bieten beispielsweise auch die Möglichkeit, Bilder für links und rechts off-Screen zu rendern und diese zu so genannten Anaglyphen zusammenzumischen. Das sind dreidimensionale Bilder, bei denen die Informationen für die beiden Augen farblich getrennt übereinander gelegt werden (das Bild für das linke Auge wird rot gefärbt, das für das Rechte grün) und die anschließend mit einer Rot-Grün-Brille betrachtet werden können. Dort fügen sich die beiden Bilder wieder zusammen und

erzeugen einen recht realistischen Effekt räumlicher Tiefe.

9030 Einen Wermutstropfen gibt es allerdings für Nutzer der DirectX-Version von Java 3D: Der Stereosupport ist nicht verfügbar, wenn diese (proprietäre) Grafikschnittstelle verwendet wird, das funktioniert nur mit OpenGL bzw. GLX.

### 12.7.3 Sonstige Canvas3D-Methoden

9035 Neben den oben bereits beschriebenen Methoden finden sich in der Klasse Canvas3D noch einige mehr, die es wert sind, hier erwähnt zu werden. Für eine vollständige Beschreibung aller Methoden möchte ich jedoch auf die Spezifikation verweisen, da sich hier auch einige sehr spezielle sowie einige von anderen Klassen eigentlich bereits hinlänglich bekannte Methoden finden.

9040

```
java.awt.Rectangle getBounds()
```

```
java.awt.Rectangle getBounds(java.awt.Rectangle rv)
```

Bei den Bounds, die von diesen Methoden zurückgeliefert werden, handelt es sich – wie am Datentyp der Returnwerte zu erkennen – erstmalig nicht um solche, die die Begrenzung eines 3D-Objektes darstellen. Vielmehr beschreiben diese Bounds das Canvas3D-Objekt als GUI-Element, dessen Position sowie Dimensionen.

9045

```
boolean getDoubleBufferAvailable()
```

Double-Buffering ist eine Technik, die für ein flackerfreies Bild sorgt. Das wird realisiert, in dem in einen nicht sichtbaren Buffer gezeichnet wird, während ein zweiter, bereits vollständig gezeichneter Buffer sichtbar ist. Wurden alle Zeichenoperationen beendet, so werden die Buffer getauscht: jetzt ist der neu gezeichnete Buffer zu sehen während in den anderen, zuvor sichtbaren Buffer das nächste Frame gezeichnet wird. Dadurch, dass die Zeichenoperationen – bei denen sich das Bild mehr oder weniger langsam aufbaut - selber nicht sichtbar sind, wird der Eindruck eines Flackerns vermieden.

9055

Der Rückgabewert dieser Methode sagt nun aus, ob das Canvas3D Double-Buffering überhaupt unterstützt (`true`) oder nicht (`false`).

9060 

```
boolean getDoubleBufferEnable()
```

```
void setDoubleBufferEnable(boolean flag)
```

Diese Methoden erlauben es, festzustellen, ob Double-Buffering aktiviert ist oder nicht bzw. sie bieten die Möglichkeit, dieses zu (de)aktivieren. Wird Double-Buffering auf einem Grafikgerät aktiviert, dass dieses nicht unterstützt, so wird ein Aufruf von `setDoubleBufferEnable(true)` ignoriert und bewirkt keine Veränderung.

9065

```
J3DGraphics2D getGraphics2D()
```

Der Rückgabewert dieser Methode ist ein Kontext auf J3DGraphics2D. Wie man an

der Ableitung dieser Klasse sehen kann, ist sie direkt mit Graphics2D verwandt und kann deswegen dazu verwendet werden, um deren 2D-Grafikfunktionalitäten direkt im Canvas3D einzusetzen:

```
9070 java.lang.Object
 java.awt.Graphics
 java.awt.Graphics2D
9075 javax.media.j3d.J3DGraphics2D
```

```
int getHeight()
```

```
int getWidth()
```

```
java.awt.Dimension getSize()
```

9080 Diese Methoden erlauben es, die Größe des Canvas3D in der Einheit Pixel zu ermitteln. Dabei beziehen sich diese auf dessen Höhe, die Breite bzw. die gesamte Dimension, die Höhe und Breite beinhaltet.

```
double getPhysicalHeight()
```

9085 double getPhysicalWidth()

Die Rückgabewerte dieser Methoden sind wieder die Höhe und die Breite des Canvas3D. Allerdings beziehen diese sich auf die reale, physische Größe und werden deswegen in der Einheit Meter angegeben.

9090 boolean getSceneAntialiasingAvailable()

Diese Methode liefert die Information, ob ein globales Scene-Antialiasing, also eine Kantenglättung für die gesamte Szene, verfügbar ist oder nicht. Das SceneAntialiasing selbst läßt sich dann mit Hilfe der Methode setSceneAntialiasingEnable() aus der Klasse View (de)aktivieren, auf die wie immer Zugriff über das SimpleUniverse besteht.

9095

## 12.7.4 Java 3D und Swing

Wie bereits erwähnt ist das Canvas3D eine Heavyweight-Komponente, die sich mit Lightweight-Elementen wie den Swing-Klassen nicht unbedingt verträgt. Das ist auch daran zu erkennen, dass ein Canvas3D Größenänderungen innerhalb von Swing-Benutzeroberflächen nicht mit vollführt und dann andere Elemente der Oberfläche überdeckt. Auch legt sich solch eine Heavyweight-Komponente über JPopup-Menüs und macht diese zumindest teilweise unleserlich.

9105 Normalerweise empfiehlt es sich, Heavyweight- und Lightweight-Komponenten gar nicht erst zu mischen, also beispielsweise generell nur auf AWT oder Swing zu setzen. Im Fall von Java 3D ist das jedoch nicht so einfach, da es keine Lightweight-Repräsentation für das Canvas3D gibt. Also muß das Problem hier anders angegangen werden.

9110 Das Problem mit den Popup-Menüs läßt sich mit einem Einzeiler beheben. Ein Aufruf von

```
JPopupMenu.setDefaultLightWeightPopupEnabled(false)
```

9115 ganz am Anfang der Applikation, also noch bevor irgend welche GUI-Objekte erzeugt werden, sorgt dafür, dass sich alle JPopupMenu-Objekte anschließend wie ein Heavyweight-Element verhalten und wirklich über dem Canvas3D auflappen.

9120 Eine Integration des Canvas3D in eine Oberfläche wird allerdings etwas schwieriger. Um das Problem zu umgehen, dass der Canvas3D Größenänderungen nicht folgt, ist hier ein wenig aktiver Code notwendig.

Als erstes empfiehlt es sich, den Canvas3D selbst in eine JScrollPane einzufügen, bei der die Scrollbars deaktiviert wurden. Ein entsprechendes Codefragment könnte so aussehen:

```
9125 (1) C3DScrollPane.getViewport().add(C3D);
 (2) C3DScrollPane.setBorder(null);
 (3) C3DScrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
 (4) C3DScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_NEVER);
 (5) C3DScrollPane.addPropertyChangeListener(aSymProp);
```

9130

In Zeile 1 wird das Canvas3D-Objekt C3D der JScrollPane C3DScrollPane hinzugefügt. Diese selbst wird mit den Anweisungen in den Zeilen 2 bis 4 praktisch unsichtbar gemacht, in dem ihr kein Border zugewiesen wird und die Scrollbars deaktiviert werden. Die Zeile 5 indes gibt einen Hinweis auf einen Listener, der sich bei Größenänderungen um das Canvas3D-Objekt kümmert. Immer wenn sich die gesamte Oberfläche in einer Form ändert, die eine Größenänderung beim JScrollPane verursacht, so wird ein PropertyChangeEvent ausgelöst:

9135

```
9140 (1) class SymProp implements PropertyChangeListener
 (2) {
 (3)
 (4) public void propertyChange(PropertyChangeEvent e)
 (5) {
 (6) Dimension NewDim=new Dimension(),ActDim;
9145 (7)
 (8) NewDim=C3DScrollPane.getSize();
 (9) if ((NewDim.width>0) && (NewDim.height>0))
 (10) {
 (11) ActDim=C3D.getSize();
9150 (12) NewDim.width-
 =(C3DScrollPane.getInsets().right+C3DScrollPane.getInsets().left);
 (13) NewDim.height-
 =(C3DScrollPane.getInsets().top+C3DScrollPane.getInsets().bottom);
 (14) if ((ActDim.width!=NewDim.width) || (ActDim.height!
9155 =NewDim.height)) C3D.setSize(NewDim);
 (15) }
 (16) }
 (17) }
```

- 9160 Die Bearbeitung dieses Events ist hier etwas aufwändiger gestaltet, im Prinzip läuft diese aber so ab, dass der Canvas3D die neue Größe der JScrollPane erhält. Immer, wenn diese verändert wird, wird die Methode `propertyChange()` in dieser Klasse `SymProp` aufgerufen. In Zeile 8 wird dann die aktuelle (vermutlich neue) Größe der JScrollPane `C3DScrollPane` ermittelt. Sind ihre Dimensionen jeweils größer als 0, so wird in Zeile 11
- 9165 die derzeitige Größe des Canvas3D-Objektes ermittelt. Die Abfrage, ob die JScrollPane überhaupt eine Größe hat, kann für den Zeitpunkt der Erzeugung der Oberfläche wichtig sein. Hier ist es möglich, dass `PropertyChangeEvents` für eine noch nicht sichtbare JScrollPane auftreten, was mit der Abfrage in Zeile 9 abgefangen wird.
- 9170 In den Zeilen 12 und 13 wird die Größe, die das Canvas3D-Objekt bekommen müßte errechnet, in dem die Inset-Werte der JScrollPane abgezogen werden, da ja nur die wirklich benötigten, effektiven Ausdehnungen dieses Objektes interessant sind. Ist nun die neue Größe `NewDim` wirklich unterschiedlich zur bisherigen Größe `ActDim` des Canvas3D, so wird dieses mit dem Aufruf von `setSize()` in Zeile 14 angepaßt, damit es
- 9175 sich wieder nahtlos in die restliche Swing-Oberfläche einfügt.
- Diese spezielle Abfrage der bisherigen und neuen Breite und Höhe ist für einige Grafikumgebungen sinnvoll, da in manchen Fällen der Aufruf von `setSize()` ein leichtes Flackern im Canvas3D verursachen kann, welches bei der häufigen, unnötigen
- 9180 Verwendung dieser Methode doch sehr stören würde. Die `if`-Abfrage in Zeile 14 vermindert dieses mögliche Flackern deswegen auf das allernötigste.

## 12.8 LOD

- 9185 Eine weitere wichtige Technik, die abschließend noch besprochen werden soll, nennt sich „Level of Detail“ und dient wiederum dazu, Performance zu sparen. Der Gedanke dahinter ist, dass mit wachsender Entfernung zu einem 3D-Objekt innerhalb einer Szene naturgemäß immer weniger Details zu erkennen sind, schlichtweg deswegen, weil diese viel zu klein werden, um nicht sichtbar zu sein. Die 3D-Hardware verarbeitet diese
- 9190 unwichtigen weil nicht mehr sichtbaren Details bisher allerdings immer noch, was wieder einen unnötigen Aufwand an Rechenleistung erfordert. Also wäre es naheliegend, für größere Entfernungen vereinfachtere Versionen der verschiedenen 3D-Objekte zu verwenden, an denen verschiedene Details eingespart wurden. Das bedeutet, diese vereinfachten Versionen bestehen zum einen aus weniger Polygonen und benötigen zum
- 9195 anderen auch nicht mehr so hoch auflösendem, große Texturen.

- Für diese Technik steht im Java 3D bereits eine abstrakte, von `Behavior` abgeleitete Basisklasse `LOD` (für „Level of Detail“) bereit, die Grundfunktionalitäten bietet, um verschieden stark detaillierte 3D-Objekte entfernungsabhängig und unter Zuhilfenahme
- 9200 einer `Switch-Group` in einer Szene zu aktivieren. Eine von `LOD` abgeleitete Klasse würde also exakt das tun, was bei komplexeren Szenen viel Rechenleistung einsparen würde: bei wachsendem Abstand zu einem 3D-Objekt dementsprechend vereinfachtere Versionen selbigens sichtbar machen.
- 9205 Da diese Klasse abstrakt ist, existiert kein verwendbarer Konstruktor, der vorgegebene Default-Konstruktor

LOD( )

9210 müßte dementsprechend von einer eigenen Klasse, die von LOD abgeleitet ist, zur Verfügung gestellt werden. Die Methoden, die LOD mitbringt, dienen in erster Linie der Verwaltung der Switch-Nodes, die sich um das zu- und abschalten der verschieden stark detaillierten 3D-Objekte kümmern sollen:

9215 `void addSwitch(Switch switchNode)`

Es wird ein neuer Switch-Node hinzugefügt.

`java.util Enumeration getAllSwitches()`

9220 Der Rückgabewert dieser Methode ist eine Enumeration, die alle Switches beinhaltet, die bereits von diesem LOD-Objekt verwaltet werden.

`Switch getSwitch(int index)`

9225 Diese Methode liefert das Switch-Objekt zurück, dass sich an der durch `index` spezifizierten Position innerhalb der LOD-internen Liste befindet. Ist diese Position mit keinem Objekt besetzt, so ist der Rückgabewert `null`.

`int indexOfSwitch(Switch switchNode)`

9230 Der zurückgegebene Wert bezeichnet die Indexposition des Nodes `switchNode`. Ist das als Parameter übergebene Switch-Objekt nicht in der LOD-internen Liste enthalten, so ist der Rückgabewert `-1`

`void insertSwitch(Switch switchNode, int index)`

9235 Mit dieser Methode ist es möglich, einen neuen Switch `switchNode` an der Position `index` einzufügen. Befindet sich an dieser Stelle bereits ein Switch-Objekt, so werden diese und alle nachfolgenden um eins in der Switchliste nach hinten verschoben.

`int numSwitches()`

9240 Diese Methode gibt an ,wie viele Switch-Nodes sich insgesamt bereits in der Liste befinden.

`void removeAllSwitches()`

9245 Ein Aufruf der Methode `removeAllSwitches()` sorgt dafür, dass die LOD-interne Liste komplett geleert wird, anschließend befinden sich in dieser also keinerlei Switch-Nodes mehr.

`void removeSwitch(int index)`



```
void removeSwitch(Switch switchNode)
```

9250 Diese Methoden ermöglichen es, einen einzelnen Switch aus der Liste zu entfernen. Welches Objekt dabei genau entfernt werden soll, wird mit Hilfe seiner Position index innerhalb der Liste festgelegt oder aber durch die Übergabe einer Referenz auf das zu entfernende Objekt selbst.

```
void setSwitch(Switch switchNode, int index)
```

9255 Diese Methode fügt ebenfalls einen neuen switchNode in die LOD-interne Liste ein. Befindet sich an der Position index jedoch bereits ein Objekt, so wird dieses vom neuen, hiermit als Parameter übergebenen Switch-Node überschrieben.

Wie bereits erwähnt leitet sich LOD von der Klasse Behavior ab, was für eine einfache Implementierung der Level-of-Detail-Funktionalität auch sinnvoll ist:

9260

```
java.lang.Object
 javax.media.j3d.SceneGraphObject
 javax.media.j3d.Node
 javax.media.j3d.Leaf
 javax.media.j3d.Behavior
 javax.media.j3d.LOD
```

### 12.8.1 DistanceLOD

9270 Mit der Klasse LOD aus dem vorigen Abschnitt läßt sich nun nicht all zu viel anfangen, für eine vernünftige, entfernungsabhängig realisierte Level-of-Detail-Funktionalität benötigt es einiges mehr. Sie haben also die Möglichkeit, eine eigene, von LOD abgeleitete Klasse zu implementieren, oder aber mit DistanceLOD eine bereits vorgefertigte Klasse einzusetzen. Wie diese funktioniert, macht der folgende Konstruktor deutlich:

9275

```
DistanceLOD(float[] distances)
```

9280 Dieser erzeugt ein neues DistanceLOD-Objekt an der Position 0,0,0. Als Parameter wird zur Steuerung der entfernungsabhängigen Level-of-Detail-Funktionalität ein Array mit Entfernungsangaben erwartet. Diese Entfernungen müssen mit wachsender Indexnummer ebenfalls größer werden und geben an, bei welcher Entfernung welcher Switch-Node aus der klassenintern verwalteten (und von LOD geerbten) Liste jeweils aktiv geschaltet werden soll. Diese Switch-Nodes müssen dann dementsprechend die für die jeweilige Entfernung vorgesehenen 3D-Objekte beinhalten. Die Entfernungswerte aus diesem Array werden dabei nach folgenden Regeln auf Indexnummern in der Switchliste umgesetzt:

9285

Ist die Entfernung des Beobachters kleiner oder gleich dem Wert von `distances[0]`, so wird der Switch-Node an Indexposition 0 verwendet.

Liegt die Entfernung  $d$  innerhalb des Bereiches, der durch `distances[i-1]` und `distances[i]` angegeben wird (also `distances[i-1] < d <= distances[i]`), so wird der Switch von Position  $i$  aktiviert.

Befindet sich der Beobachter in einer Entfernung, die größer als der Wert, der in `distances[n-1]` angegeben ist, so wird der Switch-Node an Position  $n$  verwendet.

Wie zu sehen ist, besteht ein direkter Zusammenhang zwischen der Länge des übergebenen Arrays und der Anzahl der Switch-Nodes, die von einem DistanceLOD-Objekt verwaltet werden.

Ein weiterer möglicher Konstruktor bietet darüber hinaus auch noch die Möglichkeit, die Position des DistanceLOD-Objektes und damit der von diesem beeinflussten 3D-Objekte festzulegen:

```
DistanceLOD(float[] distances, Point3f position)
```

Neben dem bereits bekannten Parameter `distances`, der die Entfernungsangaben für die einzelnen Switch-Nodes beinhaltet, gibt hier das `Point3f`-Objekt `position` die gewünschte Position der verwalteten 3D-Objekte innerhalb einer virtuellen 3D-Welt an.

Entsprechend der neu hinzugekommenen Funktionalität bietet diese Klasse auch einige zusätzliche Methoden mehr an:

```
double getDistance(int whichDistance)
```

Der Rückgabewert dieser Methode entspricht dem Wert, der sich auch an der Indexposition `whichDistance` innerhalb des vom Konstruktor her bekannten `distance`-Arrays befindet. Sie liefert also die Entfernung zurück, die für den zugehörigen Switch-Node als Schwellwert dient.

```
void setDistance(int whichDistance, double distance)
```

Hierbei handelt es sich gewissermaßen um das Gegenstück zur vorhergehenden Methode. Diese erlaubt es, einem Abstandswert `whichDistance` eine neue Entfernung `distance` zuzuweisen, die anschließend als neuer Schwellwert für den zugehörigen Switch-Node verwendet wird. Für den neuen `distance`-Wert gelten die gleichen Regeln wie für das vom Konstruktor her bekannte `distance`-Array: Der Entfernungswert muß unbedingt zwischen den Entfernungen der Abstände `whichDistance-1` und `whichDistance+1` liegen.

```
void getPosition(Point3f position)
```

```
void setPosition(Point3f position)
```

Die `position` bezieht sich auf die Position des aktuellen DistanceLOD-Objektes. Mit diesen Methoden ist es möglich, deren aktuellen Wert zu bestimmen bzw. einen neuen Wert für die Position festzulegen. Die Koordinaten werden dabei jeweils in das bzw. aus

9330 dem als Parameter übergebenen Point3f-Objekt kopiert.

```
void initialize()
```

Hierbei handelt es sich gewissermaßen um ein Erbstück und um eine für Behaviors elementar wichtige Methode. Sie dient dazu, das initale WakeupCriterion festzulegen und zu aktivieren.

```
int numDistances()
```

Diese Methode liefert die Information zurück, wie viele unterschiedliche Distanz-Schwellwerte von diesem DistanceLOD-Objekt verwaltet werden.

9340

```
void processStimulus(java.util.Enumeration criteria)
```

Auch diese Methode entstammt der Verwandtschaft mit den Behaviors. Sie wird immer dann automatisch aufgerufen, wenn das bei `initialize()` spezifizierte WakeupCriterion erfüllt ist. Hier werden dann abhängig von der aktuellen Entfernung des DistanceLOD-Objektes zum Beobachter die jeweiligen Switch-Nodes ab- bzw. neu zugeschaltet und damit die zugehörigen 3D-Objekte sichtbar gemacht.

Da diese Klasse ebenfalls ein Abkömmling von Behavior ist, ist zu beachten, dass noch einige Schritte mehr notwendig sind, um diese zu aktivieren. So muß ein solches DistanceLOD-Objekt nicht nur live sein, es muß – wie von den Behaviors her bekannt – auch einen Einflußbereich zugewiesen bekommen haben, der festlegt, dass der Behavior nur dann aktiv ist, wenn der Beobachter sich innerhalb eben dieser Scheduling Bounds befindet.

9355 `java.lang.Object`

`javax.media.j3d.SceneGraphObject`

`javax.media.j3d.Node`

`javax.media.j3d.Leaf`

`javax.media.j3d.Behavior`

9360 `javax.media.j3d.LOD`

**`javax.media.j3d.DistanceLOD`**

## 12.9 Sky-Objekte für Panoramahintergründe

9365 Im Folgenden soll das Geheimnis hinter den so extrem echt wirkenden Panoramahintergründen und Himmelsimulationen professioneller 3D-Anwendungen gelüftet werden.

Sollten Sie sich schon einmal daran versucht haben, sind sie sicher über eines der

9370 folgenden Probleme gestolpert: Wenn das Skyobjekt zu hoch angeordnet ist, gibt es  
Probleme mit der Darstellung (es muss schließlich noch innerhalb der Back Clip Distance  
liegen und trotzdem bis zum Horizont reichen). Wenn es aber zu niedrig ist, sieht das  
eher nach einer angemalten Zimmerdecke oder nach den Wänden eines Zimmers aus als  
9375 Objekte oben abgeschnitten.

Die Lösung zu diesen Problemen liefern zwei Tricks, die die Illusion von sehr weit  
entfernten Flächen erzeugen. Um sich das selbst zu verdeutlichen, können Sie sich  
einfach einmal in eine Häuserschlucht oder zwischen Bäume stellen und in den Himmel  
9380 schauen: Wenn Sie nun nach vorne gehen, scheinen alle nahen Objekte wie gewohnt  
nach hinten zu verschwinden, während der Himmel scheinbar mit Ihnen mitgeht. Und  
genau das ist auch einer der Tricks mit den Himmelsobjekten. Die Skyplane wird zum  
einen einfach mit der ViewPlatform mitbewegt, auch in der Höhe. Wie hoch über dem  
aktuellen View die Skyplane tatsächlich ist, ist dabei zweitrangig, solange dieser Abstand  
9385 immer konstant bleibt. Wichtig ist natürlich auch, dass nur die Position der  
ViewPlatform übernommen wird, nicht aber deren Rotation. Das Skyobjekt bleibt dann  
immer an den Achsen der Weltkoordinaten ausgerichtet. Diese Illusion funktioniert  
übrigens auch mit Himmelsobjekten perfekt, die sich nur einen Meter über der Spielerfigur  
befinden!

9390

Fehlt noch der zweite anzuwendende Trick, der das Problem mit den abgeschnittenen,  
hohen Objekten löst. Damit das Skyobjekt sich auch immer hinter den höchsten Objekten  
befindet wird dieses in jedem Frame jedem Frame bei deaktiviertem Z-Buffer einfach als  
erstes Objekt gerendert. Somit erscheinen alle später gerenderten Objekte immer vor und  
9395 damit also unterhalb des Skyobjekts, auch wenn sie höher sind als diese, da sie dort  
keinen Z-Buffer-Eintrag finden.

Die dazu notwendigen Funktionalitäten sind bereits bekannt: eine OrderedGroup, mit der  
die Reihenfolge der Objekte während des Renderns festgelegt wird, in dem das Skyobjekt  
9400 zuerst attached wird sowie eine kleine Manipulation an den RenderingAttributes des  
Skyobjekts:

```
RenderingAttributes().setDepthBufferEnable(false);
RenderingAttributes().setDepthBufferWriteEnable(false);
```

9405

### 12.9.1 RenderingAttributes

Die RenderingAttributes wurden bisher noch nicht näher betrachtet, da die  
Manipulationen, die hier möglich sind, bisher nicht benötigt wurden. Sie werden mittels  
9410 setRenderingAttributes() einem Appearance-Objekt hinzugefügt:

```
public RenderingAttributes()
```

9415 Der Default-konstruktor erzeugt ein `RenderingAttributes`-Objekt mit den folgenden  
Vorgabeparametern: Der Z-Buffer ist aktiviert und kann beschrieben werden, alpha test  
function ist auf **ALWAYS** gesetzt und hat den Wert 0.0 (siehe unten), Vertexfarben werden  
nicht ignoriert und Rasteroperationen sind deaktiviert.

```
9420 public RenderingAttributes(boolean depthBufferEnable, boolean
depthBufferWriteEnable, float alphaTestValue, int
alphaTestFunction)
```

9425 Dieser Konstruktor erwartet einige Werte als Parameter: `depthBufferEnable` legt fest,  
ob der Z-Buffer aktiviert werden soll – zusammen mit `depthBufferWriteEnable` die  
Funktionalität, die die oben beschriebene Skyobjekt-Illusion möglich machen kann. Ein  
deaktivieren des Z-buffers alleine genügt hier übrigens nicht, mit dem Setzen von  
`depthBufferWriteEnable` auf false muss ebenfalls dafür gesorgt werden, dass dieser  
nur lesbar ist und nicht beschrieben werden kann, um eine entsprechende Wirkung zu  
erreichen. `alphaTestValue` und `alphaTestFunction` werden verwendet, um einen  
9430 Vergleich mit den Alphawerten jedes einzelnen Pixels durchzuführen. Ist dieser Test  
erfolgreich, so wird das entsprechende Pixel gezeichnet, schlägt er fehl, wird es nicht  
dargestellt. Die `alphaTestFunction` legt mit einem der folgenden Konstanten die Art  
des Vergleichs fest:

- **ALWAYS** – Pixel werden unabhängig vom Alphawert immer gezeichnet, damit wird der  
9435 Alphavergleich effektiv also abgeschaltet.
- **NEVER** – Pixel werden niemals gezeichnet – das Gegenstück zu **ALWAYS** also.
- **EQUAL** – Pixel werden gezeichnet, wenn deren Alphawert gleich dem angegebenen  
Alpha-Vergleichswert ist.
- **NOT\_EQUAL** – diese Option ist das Gegenstück zur vorhergehenden, hier wird nur dann  
9440 gezeichnet, wenn die Alphawerte ungleich sind.
- **LESS** – es wird immer dann gezeichnet, wenn der Alphawert eines Pixels kleiner ist als  
der angegebene alpha-Vergleichswert,...
- **LESS\_OR\_EQUAL** – wenn er gleich oder größer ist oder ...
- **GREATER** – wenn er größer ist
- 9445 - **GREATER\_OR\_EQUAL** – und letztendlich werden Pixel bei dieser Variante immer dann  
dargestellt, wenn der Vergleich auf gleich oder größer wahr ist

```
9450 public RenderingAttributes(boolean depthBufferEnable, boolean
depthBufferWriteEnable, float alphaTestValue, int
alphaTestFunction, boolean visible, boolean ignoreVertexColors,
boolean rasterOpEnable, int rasterOp)
```

9455 Dieser Konstruktor erweitert die vorhergehend beschriebenen um folgende Parameter und  
damit Modifikationsmöglichkeiten: `visible` legt fest, ob das Objekt sichtbar ist oder nicht,  
`ignoreVertexColors` beeinflusst das Verhalten in Bezug auf eventuell vorhandene  
per-Vertex-Farbdefinitionen, ob diese also ignoriert oder aber ob die entsprechenden  
Farben dargestellt werden sollen und `rasterOpEnable` legt fest, ob logische  
Rasteroperationen für dieses Objekt möglich sein sollen oder nicht. Es ist zu beachten,  
das bei aktivierten Rasteroperationen alle Alphablendingfunktionen deaktiviert werden.  
9460 Für die Rasteroperationen kann mit Hilfe von `rasterOp` schlussendlich die Art der

Operation fest gelegt werden. Hier stehen **ROP\_COPY** und **ROP\_XOR** zur Auswahl. **ROP\_COPY** ist die Vorgabeoperation, hier ist Quelle gleich Ziel während diese bei **ROP\_XOR** Exklusiv-ODER verknüpft werden.

9465 Bei den von den RenderingAttributes zur Verfügung gestellten Methoden finden sich wieder Viele, die verwendet werden können, um die eben beschriebenen Eigenschaften zu manipulieren und zu verändern. Aus diesem Grunde sollen sie hier auch nur noch kurz angerissen werden:

9470 `public void setDepthBufferEnable(boolean state)`

Aktiviert (`state=true`) oder deaktiviert (`false`) den Z- oder Tiefenbuffer

`boolean getDepthBufferEnable()`

9475 Liefert den aktuellen Aktivierungsstatus des Tiefenbuffers zurück. Diese Methode wirft eine `CapabilityNotSetException` wenn das Objekt live oder compiliert ist und die entsprechende `Capability` nicht gesetzt wurde.

`void setDepthBufferWriteEnable(boolean state)`

9480 Macht den Tiefenbuffer les- und schreibbar (`true`) oder nur lesbar (`false`). Während eines Renderingdurchlaufes kann dieses Attribut vom `depthBufferFreezeTransparent`-Attribut im View-Objekt überschrieben werden.

`public boolean getDepthBufferWriteEnable()`

9485 Ermittelt den Zustand des Z-Buffers. Diese sowie die vorhergehende Methode können wieder eine `CapabilityNotSetException` auslösen, wenn die entsprechenden `Capabilities` fehlen.

`void setAlphaTestValue(float value)`

`float getAlphaTestValue()`

9490 Setzt bzw. ermittelt den aktuellen Vergleichswert für den Alphatest. auch diese Methoden benötigen entsprechende `Capabilities`, wenn entsprechende Manipulationen vorgenommen werden sollen.

`void setAlphaTestFunction(int function)`

9495 `int getAlphaTestFunction()`

Es wird die Alphatest-funktion gesetzt bzw. die aktuell verwendete ermittelt. Details hierzu sowie zu Parameter und Returnwert siehe oben.

`void setVisible(boolean visible)`

9500 Das Objekt, dem diese RenderingAttributes zugeordnet sind, kann mit Hilfe dieser

Methode sichtbar (`true`) und unsichtbar (`false`) gemacht werden. Unsichtbare Objekte erscheinen nicht mehr in der Szene, können aber mittels Picking aufgefunden werden. Wichtig in diesem Zusammenhang ist auch die Methode `setVisibilityPolicy()` des aktuellen View-Objekts.

9505

```
boolean getVisible()
```

Ermittelt, ob das Objekt momentan sichtbar ist oder nicht.

```
void setIgnoreVertexColors(boolean ignoreVertexColors)
```

9510 `boolean getIgnoreVertexColors()`

Manipuliert den Status des Flags zum Ignorieren der eventuell gesetzten Vertexfarben bzw. gibt dessen Status zurück.

```
void setRasterOpEnable(boolean rasterOpEnable)
```

9515 `boolean getRasterOpEnable()`

Aktiviert bzw. deaktiviert Rasteroperationen oder gibt den aktuellen Aktivierungsstatus zurück.

```
void setRasterOp(int rasterOp)
```

9520 `int getRasterOp()`

Diese Methoden behandeln die Art der Rasteroperation, Details hierzu siehe die Konstruktorenbeschreibung oben.

9525 Mit diesen beiden Tricks - dem Halten der Position des Skyobjektes an der ViewPlatform und dem gezielten Rendern der Objekte in einer festen Reihenfolge um die Illusion eines hoch über allen Objekten schwebenden Himmels - arbeiten übrigens alle der im folgenden beschriebenen Skyimitationen.

9530 Bevor nun endlich wieder zur Beschreibung der Skyobjekte zurück gekehrt werden kann, soll dieser Abschnitt mit dem obligatorischen Blick auf die Verwandtschaftsverhältnisse der `RenderingAttributes` abgeschlossen werden:

```
java.lang.Object
```

9535 `javax.media.j3d.SceneGraphObject`

```
 javax.media.j3d.NodeComponent
```

```
 javax.media.j3d.RenderingAttributes
```

## 12.9.2 Die Skyplane

9540

Die Skyplane dürfte das am einfachsten zu erklärende Skyobjekt sein. Dabei handelt es sich einfach um eine mit einer Himmelstextur überzogenen Fläche über der ViewPlatform. Wird diese sehr tief gesetzt, um mit einem fernen Horizont abzuschließen, so wirkt das trotz der Z-Buffer-Illusion manchmal sehr beklemmend. In solchen Fällen ist es ratsam, die Plane etwas höher zu setzen und den Horizont mit ein paar Objekten zu kaschieren – oder aber gleich einen anderen Objekttyp für den Himmel zu verwenden.

9545

9550

Da die Textur für eine Skyplane nicht verzerrt werden muss, eignet sich dieses Objekt jedoch gut für Wolkenanimation. Hierzu muss die Textur eine Kachel sein, die endlos aneinandergereiht werden kann. Verschiebt man nun in jedem Frame die Texturkoordinaten, hat man einen dahinziehenden Wolkenhimmel.

9555

Als Objekt für eine Skyplane bietet sich entweder ein sehr flaches und sehr ausgedehntes Cube-Primitive an oder aber ein selbst erzeugtes Shape3D, dessen Geometriedaten im wesentlichen aus einem einzelnen, riesigen Quad bestehen.

## 12.9.3 Skydome und Skysphere

9560

Zwei weitere, in der 3D-Programmierung geläufigen Skyobjekte sollen hier nur kurz angerissen werden. Der Skydome - die Himmelskuppel – ermöglicht ein Panorama bis zum Horizont, das auch weit entfernte Berge usw. darstellt. Dazu krümmt man die Skyplane zu einer Halbkugel, eine Art Käseglocke über der Szene. Der Z-Buffer- und Bewegen-Trick funktionieren auch hier perfekt. Nichts anderes wird realisiert, wenn eine Skysphere zum Einsatz kommt – nur, dass sich hier der Himmel auch unter der ViewPlatform in gleicher Weise wölbt.

9565

9570

Das große Problem dieser beiden Skyobjekte generell sind die Singularität des Himmelspols bzw. der beiden Himmelspole, an denen selbst bei guter Filterung oft Verzerrungen und störende "schmale Pixel" erkennbar sind. Erst recht, wenn das Auge an dieser Stelle Details erwartet.

9575

Stört dieser Effekt nicht, weil es möglich ist, diese Stellen passend zu kaschieren oder der Blick dort hin nicht möglich ist, bietet besonders die Skysphere eine gute Möglichkeit mit wenig aufwändig zu erzeugenden Texturen überzeugende Hintergründe zu schaffen.

9580

Übrigens ... ist es Ihnen aufgefallen? Im Abschnitt über die Backgrounds in Java3D wurde die Skysphere bereits angesprochen. Hier liefert Java3D die passenden Funktionalitäten bereits mit, es sind also keine zusätzlichen Tricks und Manipulationen mit ViewPlatform und Z-Buffer notwendig.



## 12.9.4 Die Skybox

9585 Kommen wir schlussendlich zu dem Skyobjekt, das die besten Ergebnisse liefert, leider aber – speziell bei der Textur - auch am aufwändigsten zu erzeugen ist: Die Skybox. Wie der Name es andeutet, kommt hier ein Würfel zum Einsatz, innerhalb dessen sich die ViewPlatform befindet und an dessen Innenseiten die Landschaft projiziert werden muss.

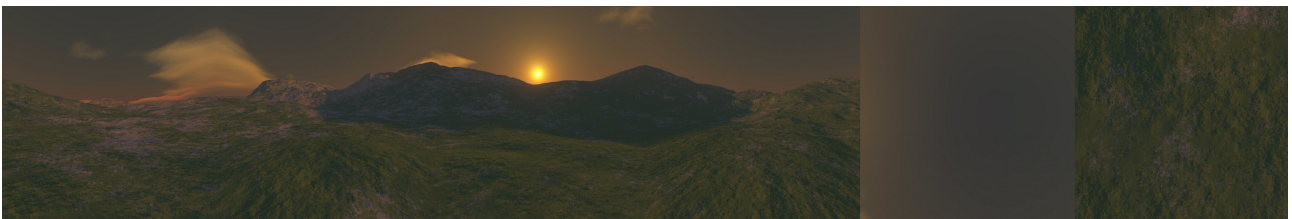
9590 Um eine jetzt sicher aufkommende Frage gleich zu beantworten: Eine Skybox besitzt innen keine sichtbaren Ecken hat, wenn man die richtigen Texturen benutzt. Leider können aber die typischen Panoramatexturen hier nicht verwendet werden - die Verzerrung ist eine andere. Es ist also nötig, jede Würfelseite einzeln zu behandeln, was nahtlos passende Realaufnahmen sehr schwierig macht. Dafür kann man Skybox-Texturen z.B. mit Raytracingprogrammen sehr einfach künstlich erzeugen.

9595 Da für eine Skybox jede Seitenfläche des zu verwendenden Würfels eine andere Textur aufweisen muss, eignet sich hier das Cube-Primitive leider nicht. Wer nicht ein vorgefertigtes Skybox-Objekt (z.B. aus der Java-XTools-Package) verwenden möchte, ist hier also gezwungen, sich ein eigenes Shape3D mit den entsprechenden Eigenschaften und Texturkoordinaten zu erzeugen

9600

Das Erzeugen einer passenden Skybox-Textur ist nicht ganz trivial, im Prinzip aber nichts anderes als die Projektion einer beliebigen 3D-Szene auf die 6 Innenseiten eines Würfels. Um nun eine solche Textur mit z.B. einem Raytracer zu erzeugen, muss man sich zuerst die zu projizierende Szene aufbauen. Ob das nun Einzelbilder von Planeten und Galaxien sind, oder hochaufgelöste, importierte 3D-Objekte, ist dabei egal. Die Szene muss dabei natürlich nur von einem Punkt aus perfekt aussehen: dem Mittelpunkt der Projektion. Das dafür aber in allen Richtungen. Es sollten natürlich nur Objekte zu sehen sein, die in der späteren Szene nicht erreichbar sind, wie z.B. ferne Galaxien, Sterne, Planeten, Wolken, Berge und bei denen der Spieler keine Größen- oder Positionsänderung erwartet, wenn er sich bewegt.

9610



*Beispiel einer zusammen hängenden Skybox-Textur für die verschiedenen Seiten des Würfels (links, vorne, rechts, hinten oben und unten); eine Verwendung dieser Textur erfordert zwingend, dass die Texturkoordinaten der Skybox entsprechend gesetzt sind*

9615

Sind alle Objekte plaziert, setzt man die Kamera des Raytracers in die Mitte der Szene, benutzt ein quadratisches View Frustum mit 90 Grad Öffnungswinkel von Seite zu Seite und rendert das Ganze auf die gewünschte Teiltexturgröße (z.B. 512x512 Pixel). Das insgesamt vier mal jeweils um 90 Grad um die Y-Achse gedreht (für die vier Himmelsrichtungen) sowie nochmal senkrecht nach oben und nach unten. Anschließend werden die sechs Teiltexturen in einem Grafikprogramm zusammengesetzt und können

9620

dann auf das Skybox-Objekt gelegt werden.

## 13 Anhang A – Überblick über die mitgelieferten Sourcen

| <b>Verzeichnisname</b> | <b>Benötigte Files</b>                                                 | <b>Kurzbeschreibung</b>                                                                                                                                                                                         |
|------------------------|------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BackgroundSound        | Universe.java,<br>teppich.png,<br>voegel.wav                           | Demonstriert den Einsatz von Soundnodes zur Erzeugung von Hintergrundsounds mittels des BackgroundSound-Nodes.                                                                                                  |
| Capability             | Universe.java                                                          | Zeigt das Prinzip der Capabilities an Hand einer rotierenden Box, die von einem SpotLight sowie einem PointLight beleuchtet wird und für deren Rotation die Capability ALLOW_TRANSFORM_WRITE gesetzt sein muss. |
| ConeSound              | Universe.java,<br>beep.au,<br>holz.png,<br>speaker.png,<br>teppich.png | Hier kommt mit dem ConeSound ein weiterer Soundnode zum Einsatz. Dessen Wirkung kann mit Hilfe des ebenfalls verwendeten MouseZoom-Behaviors überprüft werden.                                                  |
| DataSharing            | Universe.java                                                          | Es werden Funktion und Wirkungsweise des Datasharings mittels SharedGroup und Link-Node an Hand eines ColorCube demonstriert.                                                                                   |
| Durchsichtiges         | Universe.java                                                          | Hier wird ein durchsichtiges 3D-Objekt                                                                                                                                                                          |
|                        |                                                                        |                                                                                                                                                                                                                 |
|                        |                                                                        |                                                                                                                                                                                                                 |
|                        |                                                                        |                                                                                                                                                                                                                 |
|                        |                                                                        |                                                                                                                                                                                                                 |
|                        |                                                                        |                                                                                                                                                                                                                 |

| <b>Verzeichnisname</b>    | <b>Benötigte Files</b>                                             | <b>Kurzbeschreibung</b>                                                                                                                                                                     |
|---------------------------|--------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Licht                     | Universe.java                                                      | Ein Box-Objekt wird von zwei verschiedenfarbigen Lichtquellen vom Typ PointLight und SpotLight angestrahlt.                                                                                 |
| MouseRotate               | Universe.java                                                      | Die Orientierung eines Cone-Objektes wird mittels eines MouseRotate-Behaviors manipuliert.                                                                                                  |
| MouseRotate-TranslateZoom | Universe.java                                                      | Orientierung und Lage eines Cone-Objektes werden mit Hilfe der Behaviors MouseRotate, MouseTranslate und MouseZoom manipuliert.                                                             |
| NormalGenerator           | Universe.java                                                      | Der NormalGenerator wird verwendet, um einem mittels GeometryInfo erzeugten Shape3D Normals hinzuzufügen. Zusätzlich kommt der Stripifier zum Einsatz, um die Geometriedaten zu optimieren. |
| PickCanvas                | Universe.java                                                      | Es wird Picking mittels des PickCanvas an einem Text3D-Objekt gezeigt.                                                                                                                      |
| PointSound                | Universe.java,<br>box.png,<br>kuh.wav,<br>loon.wav,<br>teppich.png | Es kommen zwei PointSound-Nodes zum Einsatz, zwischen deren Positionen mittels MouseRotate und MouseZoom navigiert werden kann.                                                             |
| RotationInterpolator      | Universe.java                                                      | Ein Box-Objekt führt eine beschleunigte Pendelbewegung aus, die von einem RotationInterpolator sowie dem zugehörigen Alpha-Objekt gesteuert wird.                                           |
| Shape3D                   | Universe.java                                                      | Es wird ein Shape3D-Objekt unter Verwendung eines IndexedTriangleArray erzeugt.                                                                                                             |
| Soundscape                | Universe.java,<br>box.png,<br>loon.wav,<br>teppich.png             | Es wird ein komplexer Raumklang unter Verwendung von Soundscape und AuralAttributes erzeugt.                                                                                                |
| Sternenhimmel             | Universe.java                                                      | Es wird ein Sternenhintergrund erzeugt, in dem ein PointArray als Background-Geometry verwendet wird.                                                                                       |
| Sternenhimmel-Textur      | Universe.java,<br>bg.jpg                                           | Das Beispiel „Sternenhimmel“ wird um eine Textur erweitert, die zusätzlich zum PointArray für den Background-Node verwendet wird.                                                           |
| Texturen                  | Universe.java,<br>tiger.jpg                                        | Eine Sphere wird mit einer Grafik texturiert. Dazu kommen Texture-, TexCoordGeneration- und TextureAttributes-Objekte sowie TextureLoader und Material zum Einsatz.                         |
| ViewPlatform              | Universe.java                                                      | Die Position der ViewPlatform und damit die Kameraposition wird verändert.                                                                                                                  |

## 14 Index

### A

5.5Abstrahlrichtung 63  
6ActionListener 68  
9.1.4addBranchGraph() 142  
12.3addChild() 20, 51, 71, 99, 114, 216  
8.4.2addGeometry() 105, 107, 117  
9.1.4addMouseListener() 140  
12.5addScope() 65, 172, 221  
12.8addSwitch() 232  
9.1.3ALIGN\_CENTER 138f.  
9.1.3ALIGN\_FIRST 138f.  
9.1.3ALIGN\_LAST 138f.  
11.1.5ALLOW\_ANGULAR\_ATTENUATION\_READ 198  
11.1.5ALLOW\_ANGULAR\_ATTENUATION\_WRITE 198  
10.1.2.1ALLOW\_ANTIALIASING\_READ 171  
10.1.2.1ALLOW\_ANTIALIASING\_WRITE 171  
8.4ALLOW\_APPEARANCE\_OVERRIDE\_READ 108  
8.4ALLOW\_APPEARANCE\_OVERRIDE\_WRITE 108  
12.5ALLOW\_APPEARANCE\_READ 108, 222  
12.5ALLOW\_APPEARANCE\_WRITE 106, 108, 222  
11.2.2ALLOW\_APPLICATION\_BOUNDS\_READ 208  
11.2.2ALLOW\_APPLICATION\_BOUNDS\_WRITE 208  
11.2.1ALLOW\_ATTRIBUTE\_GAIN\_READ 204  
11.2.1ALLOW\_ATTRIBUTE\_GAIN\_WRITE 204  
11.2.2ALLOW\_ATTRIBUTES\_READ 208  
11.2.2ALLOW\_ATTRIBUTES\_WRITE 208  
12.2ALLOW\_AUTO\_COMPUTE\_BOUNDS\_READ 72, 215  
12.2ALLOW\_AUTO\_COMPUTE\_BOUNDS\_WRITE 72, 215  
12.2ALLOW\_BOUNDS\_READ 72, 215  
12.2ALLOW\_BOUNDS\_WRITE 72, 215  
11.1.1ALLOW\_CACHE\_READ 181  
11.1.1ALLOW\_CACHE\_WRITE 181  
11.1.2ALLOW\_CHANNELS\_USED\_READ 184  
12.3ALLOW\_CHILDREN\_EXTEND 71, 216  
12.3ALLOW\_CHILDREN\_READ 71, 217  
12.3ALLOW\_CHILDREN\_WRITE 71, 217  
12.2ALLOW\_COLLIDABLE\_READ 72, 215  
12.2ALLOW\_COLLIDABLE\_WRITE 72, 215  
12.3ALLOW\_COLLISION\_BOUNDS\_READ 72, 108, 218  
12.3ALLOW\_COLLISION\_BOUNDS\_WRITE 72, 108, 218  
10.2ALLOW\_COLOR\_READ 166, 174  
10.2ALLOW\_COLOR\_WRITE 166, 174  
11.1.2ALLOW\_CONT\_PLAY\_READ 182  
11.1.2ALLOW\_CONT\_PLAY\_WRITE 182  
11.2.1ALLOW\_DECAY\_FILTER\_READ 204  
11.2.1ALLOW\_DECAY\_FILTER\_WRITE 204  
11.2.1ALLOW\_DECAY\_TIME\_READ 204  
11.2.1ALLOW\_DECAY\_TIME\_WRITE 204  
11.2.1ALLOW\_DENSITY\_READ 176, 205

|                                          |          |  |
|------------------------------------------|----------|--|
| 11.2.1ALLOW_DENSITY_WRITE                | 176, 205 |  |
| 11.2.1ALLOW_DIFFUSION_READ               | 205      |  |
| 11.2.1ALLOW_DIFFUSION_WRITE              | 205      |  |
| 11.1.5ALLOW_DIRECTION_READ               | 198      |  |
| 11.1.5ALLOW_DIRECTION_WRITE              | 198      |  |
| 11.2.1ALLOW_DISTANCE_FILTER_READ         | 205      |  |
| 11.2.1ALLOW_DISTANCE_FILTER_WRITE        | 205      |  |
| 11.1.5ALLOW_DISTANCE_GAIN_READ           | 193, 199 |  |
| 11.1.5ALLOW_DISTANCE_GAIN_WRITE          | 193, 199 |  |
| 10.2.2ALLOW_DISTANCE_READ                | 177      |  |
| 10.2.2ALLOW_DISTANCE_WRITE               | 177      |  |
| 11.1.2ALLOW_DURATION_READ                | 182      |  |
| 11.1.2ALLOW_ENABLE_READ                  | 183      |  |
| 11.1.2ALLOW_ENABLE_WRITE                 | 183      |  |
| 11.2.1ALLOW_FREQUENCY_SCALE_FACTOR_READ  | 205      |  |
| 11.2.1ALLOW_FREQUENCY_SCALE_FACTOR_WRITE | 205      |  |
| 10.1.1ALLOW_GEOMETRY_READ                | 108, 166 |  |
| 10.1.1ALLOW_GEOMETRY_WRITE               | 108, 166 |  |
| 10.1.1ALLOW_IMAGE_READ                   | 166      |  |
| 10.1.1ALLOW_IMAGE_SCALE_MODE_READ        | 166      |  |
| 10.1.1ALLOW_IMAGE_SCALE_MODE_WRITE       | 166      |  |
| 10.1.1ALLOW_IMAGE_WRITE                  | 166      |  |
| 12.5ALLOW_INFLUENCING_BOUNDS_READ        | 174, 222 |  |
| 12.5ALLOW_INFLUENCING_BOUNDS_WRITE       | 174, 222 |  |
| 11.1.2ALLOW_INITIAL_GAIN_READ            | 183      |  |
| 11.1.2ALLOW_INITIAL_GAIN_WRITE           | 183      |  |
| 9.1.3ALLOW_INTERSECT                     | 140      |  |
| 11.1.2ALLOW_IS_PLAYING_READ              | 185f.    |  |
| 11.1.2ALLOW_IS_READY_READ                | 186      |  |
| 6.1.1ALLOW_LOCAL_TO_VWORLD_READ          | 72       |  |
| 11.1.2ALLOW_LOOP_READ                    | 183      |  |
| 11.1.2ALLOW_LOOP_WRITE                   | 183      |  |
| 11.1.2ALLOW_MUTE_READ                    | 183      |  |
| 11.1.2ALLOW_MUTE_WRITE                   | 183      |  |
| 11.1.2ALLOW_PAUSE_READ                   | 184      |  |
| 11.1.2ALLOW_PAUSE_WRITE                  | 184      |  |
| 12.2ALLOW_PICKABLE_READ                  | 72, 216  |  |
| 12.2ALLOW_PICKABLE_WRITE                 | 72, 216  |  |
| 11.1.4ALLOW_POSITION_READ                | 193      |  |
| 11.1.4ALLOW_POSITION_WRITE               | 193      |  |
| 11.1.2ALLOW_PRIORITY_READ                | 184      |  |
| 11.1.2ALLOW_PRIORITY_WRITE               | 184      |  |
| 11.1.2ALLOW_RATE_SCALE_FACTOR_READ       | 184      |  |
| 11.1.2ALLOW_RATE_SCALE_FACTOR_WRITE      | 184      |  |
| 11.2.1ALLOW_REFLECTION_COEFFICIENT_READ  | 206      |  |
| 11.2.1ALLOW_REFLECTION_COEFFICIENT_WRITE | 206      |  |
| 11.2.1ALLOW_REFLECTION_DELAY_READ        | 206      |  |
| 11.2.1ALLOW_REFLECTION_DELAY_WRITE       | 206      |  |
| 11.1.2ALLOW_RELEASE_READ                 | 185      |  |
| 11.1.2ALLOW_RELEASE_WRITE                | 185      |  |
| 11.2.1ALLOW_REVERB_COEFFICIENT_READ      | 207      |  |
| 11.2.1ALLOW_REVERB_COEFFICIENT_WRITE     | 207      |  |

11.2.1ALLOW\_REVERB\_DELAY\_READ 206  
 11.2.1ALLOW\_REVERB\_DELAY\_WRITE 206  
 11.2.1ALLOW\_REVERB\_ORDER\_READ 207  
 11.2.1ALLOW\_REVERB\_ORDER\_WRITE 207  
 11.2.1ALLOW\_ROLLOFF\_READ 207  
 11.2.1ALLOW\_ROLLOFF\_WRITE 207  
 11.1.2ALLOW\_SCHEDULING\_BOUNDS\_READ 185  
 11.1.2ALLOW\_SCHEDULING\_BOUNDS\_WRITE 185  
 12.5ALLOW\_SCOPE\_READ 174, 221f.  
 12.5ALLOW\_SCOPE\_WRITE 174, 221f.  
 10.1.2.1ALLOW\_SIZE\_READ 171  
 10.1.2.1ALLOW\_SIZE\_WRITE 171  
 11.1.2ALLOW\_SOUND\_DATA\_READ 185  
 11.1.2ALLOW\_SOUND\_DATA\_WRITE 185  
 12.4ALLOW\_SWITCH\_READ 219  
 12.4ALLOW\_SWITCH\_WRITE 219  
 8.4.8.1ALLOW\_TRANSFORM\_READ 71, 99, 124  
 13ALLOW\_TRANSFORM\_WRITE 70f., 99, 124, 243  
 11.1.1ALLOW\_URL\_READ 181  
 11.1.1ALLOW\_URL\_WRITE 181  
 11.2.1ALLOW\_VELOCITY\_SCALE\_FACTOR\_READ 207  
 11.2.1ALLOW\_VELOCITY\_SCALE\_FACTOR\_WRITE 207  
 13Alpha 73, 75f., 78, 80f., 85, 244  
 12.9.1alphaTestFunction 237  
 12.9.1alphaTestValue 237  
 12.5AlternateAppearance 25, 106, 220  
 12.9.1ALWAYS 237  
 8.3Ambient 104  
 4.1.3AMBIENT 32  
 4.1.3Ambient Color 30ff.  
 4.1.3Ambient Color 32  
 4.1.3AmbientColor 31  
 10AmbientLight 28f., 163  
 11.1.5Angular Attenuation 194  
 13Appearance 27, 29f., 51, 84, 109, 114f., 121, 236, 243  
 3.1Applet 14  
 3.1Applikation 14  
 11.1.3AudioDevice 187  
 11.3AudioDevice3D 209  
 4.3Auflösung 43  
 7.3Aufwachbedingung 92  
 13AuralAttributes 244  
 9.1.1AWT 93, 136  
 7.3.1AWTEvent 93  
 3.3AxisAngle4f 18  
 B  
 12.9Back Clip Distance 163f., 236  
 4.3.1Backface-Culling 45  
 13Background 25, 164f., 187, 240, 244  
 13BackgroundSound 163, 188f., 192, 243  
 13Behavior 25, 80, 82, 89f., 92, 94, 98ff., 124, 167, 181, 187, 235, 243f.  
 13Behaviors 161, 235, 243

- 4.1.1Beleuchtung 28
- 7.1Beobachter 55, 87, 89
- 7Beobachterposition 87
- 6.2.1Beschleunigung 73, 77
- 6Bewegung 68
- 8.4.8.1Billboard 124
- 4.2.3BLEND 39f.
- 4.3.2BLEND\_ONE 48
- 4.3.2BLEND\_ONE\_MINUS\_SRC\_ALPHA 48
- 4.3.2BLEND\_SRC\_ALPHA 48
- 4.3.2BLEND\_ZERO 48
- 4.3.2BLENDED 47
- 12.2BoundingBox 58f., 137, 215
- 3.4.2BoundingLeaf 25
- 5.3BoundingPolytope 59f.
- 12.2BoundingSphere 28, 54ff., 58, 95, 215
- 12.3Bounds 29, 55, 58f., 106f., 137, 181, 185, 206, 215, 218
- 9.1.5BOUNDS 141, 144
- 13Box 53, 243f.
- 10.1.1BranchGroup 16f., 19f., 23, 51f., 142, 151, 156f., 166
- 4.2.1BufferedImage 35
- C
- 10.1.2.1Canvas3D 14, 92, 140ff., 170
- 13Capabilities 108, 114, 124, 166, 177, 182, 243
- 12.1Capability 69, 210
- 10.2.2Capability Bit 50, 70f., 76, 114, 177
- 12.9.1CapabilityNotSetException 69, 238
- 9.1.5.3Child 19, 24, 71, 157
- 12.4CHILD\_ALL 219
- 12.4CHILD\_MASK 219
- 12.4CHILD\_NONE 219
- 7.3.1ClassCastException 93
- 2.1.2Classpath 9
- 12.1clearCapability() 211
- 12.1clearCapabilityIsFrequent() 211
- 8.4.9.3.1clearData() 132
- 3.4.2Clip 25
- 12.2cloneNode() 211, 213
- 12.2cloneTree() 211f., 214
- 5.1closestIntersection() 55
- 8.4.9.3COLLECT\_STATS 131f.
- 7.3.2.3collidable 95
- 8.4Collision 107
- 12.3Collision Bounds 107f., 218
- 9.2Collision Prevention 101, 161f.
- 6.2.2.2Color 83
- 10.1.2COLOR\_3 109, 111, 149, 169
- 8.4.1COLOR\_4 109, 112
- 8.4.1Color3b 111
- 8.4.1Color3f 109, 111
- 8.4.1Color4f 109
- 13ColorCube 15ff., 53, 243



- 6.2.2.2ColorInterpolator 83
- 8.4.9.1com.sun.j3d 101, 127
- 9com.sun.j3d.util.picking 134
- 7.3.4com.sun.j3d.utils.behaviors 98
- 7.3.6com.sun.j3d.utils.behaviors.keyboard 100
- 7.3.5com.sun.j3d.utils.behaviors.mouse 98, 100
- 8.4.9.3.1com.sun.j3d.utils.geometry 15, 126, 132
- 4.2com.sun.j3d.utils.image 33
- 9.1.5.4com.sun.j3d.utils.picking 148, 151, 158
- 3.1com.sun.j3d.utils.universe 14
- 4.2.3COMBINE 39f.
- 5.1combine() 56
- 8.4.9.1compact() 128
- 12.1compile() 17, 19f., 25, 69f., 114, 166, 171, 182, 210
- 3.2Compilieren 17
- 8.4CompressedGeometry 106
- 6.2.2.1computeTransform() 82
- 13Cone 87, 243f.
- 13ConeSound 194, 199, 243
- 10.1.2COORDINATES 109, 116, 169
- 11.1.3createAudioDevice() 187
- 12.9.2Cube 240
- 4.3.1CULL\_BACK 45, 47
- 4.3.1CULL\_FRONT 45, 47
- 4.3.1CULL\_NONE 45, 47
- 12.7.2CYCLOPEAN\_EYE\_VIEW 227
- 13Cylinder 43, 243
- D
- 5.5Dämpfung 61, 63
- 12.2Dangling Reference 214
- 12.2DanglingReferenceException 214f.
- 4.2.3DECAL 39f.
- 4.3.2.2DecalGroup 51
- 11.2.1Decay 204
- 6.2.1decreasing 76f.
- 6.2.1DECREASING\_ENABLE 77, 79
- 11.2.1Density 205
- 12.9.1depthBufferEnable 51, 237
- 12.9.1depthBufferFreezeTransparent 238
- 12.9.1depthBufferWriteEnable 51, 237
- 3.1destroy() 14
- 3.3.1.2.1detach() 20
- 8.3Diffuse 104
- 4.1.3DIFFUSE 32
- 4.1.3Diffuse Color 31f.
- 11.2.1Diffusion 205
- 10.1.1DirectionalLight 28f., 31, 163f.
- 12.7.2DirectX 8, 169, 228
- 11.1.5Distance Gain 194
- 12.8.1DistanceLOD 233
- 12.7.3Double-Buffering 228
- 4.3.1Drahtgitter 44

- 4.3Drahtgittermodell 44
- 8.4.9.3.1Dreieck 110, 133
- 8.4.9.3Dreiecke 45, 131
- 12.2duplicateNode() 213f.
- 12.1duplicateSceneGraphObject() 211
- E
- 5.4Echtzeit-3D-Rendering 60
- 10.1.1Einflussbereich 164
- 12.8.1Einflußbereich 25, 75, 173, 185, 189, 220, 235
- 5Einflußbereiche 54
- 4.1.3EMISSIVE 32
- 4.2.3Emissive Color 30, 32, 39
- 12.3ENABLE\_COLLISION\_REPORTING 73, 95, 215, 217
- 12.2ENABLE\_PICK\_REPORTING 73, 216
- 12.8Enumeration 65, 92, 106, 217, 232
- 12.9.1EQUAL 237
- 9.1.5.4equals() 56, 158f.
- 13ExponentialFog 171, 175, 243
- 9.1.2ExtrusionShape 136
- 4.2.2EYE\_LINEAR 37f.
- F
- 4.3.1Face-Culling 45, 47
- 4.3.1Faces 45
- 8.4.9.2faceted 131
- 12.1false 78, 212
- 6.2.2.2Farbe 27, 83
- 8.3Farben 104
- 4.3.2FASTEST 40f., 47, 49
- 4.2.4FILTER4 41
- 6.2.1finished() 78
- 8.4.9.2Flächen 130
- 10.2Fog 25, 164, 171
- 9.1.2Font 135, 137
- 9.1.2Font3D 136
- 9.1.2FontExtrusion 135, 137
- 11.1.2Format 185
- 5Framerate 55
- G
- 7GENERATE\_NORMALS 53, 87
- 10.1.2GENERATE\_NORMALS\_INWARD 53, 166
- 10.1.2GENERATE\_TEXTURE\_COORDS 38, 166
- 8.4.9.2generateNormals() 130
- 12.2Geometry 108, 112, 140, 215
- 9.1.5GEOMETRY 141, 144
- 9.1.5GEOMETRY\_INTERSECT\_INFO 144
- 9.1.5.1.1GeometryArray 110f., 114, 117, 120, 126, 147, 149
- 8.4.8GeometryArrays 121
- 13GeometryInfo 127f., 130, 244
- 9.1.5.1.1geometryIsIndexed() 148
- 9.1.5.2.5get() 151f., 156
- 9.1.3getAlignment() 138
- 8.4.8.1getAlignmentAxis() 122, 125

8.4.8.1getAlignmentMode() 123, 125  
 12.3getAllChildren() 217  
 9.1.5.1getAllGeometries() 106, 147  
 12.5getAllScopes() 65, 172, 222  
 12.8getAllSwitches() 232  
 6.2.1getAlphaAtOneDuration() 78  
 6.2.1getAlphaAtZeroDuration() 78  
 12.9.1getAlphaTestFunction() 238  
 12.9.1getAlphaTestValue() 238  
 12.3getAlternateCollisionTarget() 217  
 4.1.3getAmbientColor() 31f.  
 11.1.5getAngularAttenuation() 198  
 11.1.5getAngularAttenuationLength() 198  
 12.5getAppearance() 106, 222  
 8.4getAppearanceOverrideEnable() 106  
 11.2.2getApplicationBounds() 208  
 5.4getAttenuation() 61  
 11.2.1getAttributeGain() 204  
 11.2.2getAuralAttributes() 208  
 8.4.9.3.1getAvgNumVertsPerTri() 132  
 8.4.9.3.1getAvgStripLength() 132  
 7.3.1getAWTEvent( 93  
 7.3.1getAWTEvent() 93  
 12.6getBackClipDistance() 224  
 10.2.2getBackDistance() 177  
 4.3.1getBackFaceNormalFlip() 46  
 9.1.3getBoundingBox() 137, 139  
 12.7.3getBounds() 72, 106, 215, 228  
 12.2getBoundsAutoCompute() 72, 215  
 9.1.5getBranchGroup() 143  
 11.1.1getCacheEnable() 180  
 9.1.4getCanvas() 142  
 12.1getCapability() 70, 211  
 12.1getCapabilityIsFrequent() 211  
 5.1getCenter() 56  
 9.1.3getCharacterSpacing() 139  
 12.3getChild() 71, 217  
 4.3.2.2getChildIndexOrder() 51  
 12.4getChildMask() 219  
 9.1.5.1getClosestIntersection () 147  
 9.1.5.1.1getClosestVertexCoordinates() 148f.  
 9.1.5.1.1getClosestVertexIndex() 148  
 12.2getCollidable() 72, 215  
 8.4getCollisionBounds() 72, 106  
 10.2getColor() 66, 165, 173  
 4.1.3getColorTarget() 31f.  
 5.5getConcentration() 63  
 8.4.8getConstantScaleEnable() 123  
 11.1.2getContinuousEnable() 182  
 4.3.1getCullFace() 46  
 11.2.1getDecayFilter() 204  
 11.2.1getDecayTime() 204

6.2.1getDecreasingAlphaDuration() 78  
 6.2.1getDecreasingAlphaRampDuration() 78  
 11.2.1getDensity() 175, 205  
 12.9.1getDepthBufferEnable() 238  
 12.9.1getDepthBufferWriteEnable() 238  
 11.2.1getDiffusion() 205  
 11.1.5getDirection() 64, 152, 154, 198  
 12.8.1getDistance() 148, 234  
 11.2.1getDistanceFilter() 205  
 11.2.1getDistanceFilterLength() 205  
 11.1.5getDistanceGain() 193, 198  
 11.1.4getDistanceGainLength() 193  
 12.7.3getDoubleBufferAvailable() 228  
 12.7.3getDoubleBufferEnable() 228  
 4.3.2getDstBlendFunction() 48  
 11.1.2getDuration() 182  
 11.1.2getEnable() 37, 66, 182  
 9.1.5.2.4getEnd() 153, 155  
 9.1.1getExtrusionShape() 136  
 9.1.2getFont() 137  
 9.1.3getFont3D() 139  
 9.1.2getFontExtrusion() 137  
 11.2.1getFrequencyScaleFactor() 205  
 12.6getFrontClipDistance() 224  
 10.2.2getFrontDistance() 177  
 4.2.2getGenMode() 38  
 10.1.1getGeometry() 107, 147, 165  
 9.1.5.1getGeometryArray() 129, 147  
 9.1.5.1getGeometryArrays() 147  
 12.7.3getGraphics2D() 228  
 12.7.3getHeight() 229  
 12.9.1getIgnoreVertexColors() 239  
 12.7.1getImage() 35, 165, 226  
 10.1.1getImageScaleMode() 165  
 6.2.1getIncreasingAlphaDuration() 79  
 6.2.1getIncreasingAlphaRampDuration() 79  
 12.5getInfluencingBounds() 66, 173, 222  
 11.1.2getInitialGain() 183  
 11.1.1getInputStream() 180  
 4.1.3getLightingEnable() 32  
 3.4.2getLinks() 25  
 6.1.1getLocalToWorld() 72  
 11.1.2getLoop() 183  
 6.2.1getLoopCount() 79  
 5.2getLower() 59  
 4.2.4getMagFilter() 41  
 4.1.3getMaterial() 31  
 6.2.2.1getMaximumAngle() 83  
 8.4.9.3.1getMaxStripLength() 132  
 4.2.4getMinFilter() 42  
 6.2.2.1getMinimumAngle() 83  
 8.4.9.3.1getMinStripLength() 133

9.1.5getMode() 79, 143  
 12.7.2getMonoscopicViewPolicy() 227  
 11.1.2getMute() 183f.  
 9.1.5.4getNode() 158  
 11.1.2getNumberOfChannelsUsed() 183  
 8.4.9.3.1getNumOrigTris() 133  
 8.4.9.3.1getNumOrigVerts() 133  
 5.3getNumPlanes() 60  
 8.4.9.3.1getNumStrips() 133  
 8.4.9.3.1getNumVerts() 133  
 9.1.5.5getObject() 142, 147, 158, 160  
 12.7.1getOffScreenBuffer() 226  
 9.1.5.2.4getOrigin() 153f.  
 12.2getParent() 216  
 9.1.3getPath() 139  
 11.1.2getPause() 184  
 6.2.1getPauseTime() 80  
 4.2.3getPerspectiveCorrectionMode() 40  
 6.2.1getPhaseDelayDuration() 79  
 12.7.3getPhysicalHeight() 229  
 12.7.3getPhysicalWidth() 229  
 12.2getPickable() 73, 215  
 9.1.5.1getPickShape() 144, 147  
 4.2.2getPlaneQ() 38  
 4.2.2getPlaneR() 38  
 5.3getPlanes() 60  
 4.2.2getPlaneS() 38  
 4.2.2getPlaneT() 38  
 10.1.2.1getPointAntialiasingEnable() 170  
 9.1.5.1.1getPointColor() 148  
 9.1.5.1.1getPointCoordinates() 149  
 9.1.5.1.1getPointNormal() 149  
 10.1.2.1getPointSize() 170f.  
 9.1.5.1.1getPointTextureCoordinate() 149  
 4.3.1getPolygonMode() 47  
 12.8.1getPosition() 62, 139, 193, 234  
 7.3.2.2getPostId() 94  
 12.7getPreferredConfiguration() 225  
 9.1.5.1.1getPrimitiveColorIndices() 149  
 9.1.5.1.1getPrimitiveColors() 149  
 9.1.5.1.1getPrimitiveCoordinateIndices() 149  
 9.1.5.1.1getPrimitiveNormalIndices() 150  
 9.1.5.1.1getPrimitiveNormals() 150  
 9.1.5.1.1getPrimitiveTexCoordIndices() 150  
 9.1.5.1.1getPrimitiveTexCoords() 150  
 9.1.5.1.1getPrimitiveVertexIndices() 150  
 11.1.2getPriority() 184  
 9.1.5.2.3getRadius() 56, 153  
 12.9.1getRasterOp() 239  
 12.9.1getRasterOpEnable() 239  
 11.1.2getRateScaleFactor() 184  
 11.2.1getReflectionCoefficient() 206

11.2.1getReflectionDelay() 206  
 11.1.2getReleaseEnable() 185  
 11.2.1getReverbBounds() 206  
 11.2.1getReverbCoefficient() 206  
 11.2.1getReverbDelay() 206  
 11.2.1getReverbOrder() 207  
 11.2.1getRolloff() 207  
 8.4.8.1getRotationPoint() 123, 125  
 8.4.8getScale() 21, 123  
 4.2.1getScaledImage() 35f.  
 12.7.3getSceneAntialiasingAvailable() 229  
 9.1.5.1getSceneGraphPath() 147  
 11.1.2getSchedulingBounds() 185  
 12.5getScope() 65, 172, 221  
 3.4.1getSharedGroup() 24  
 4.1.3getShininess() 32  
 12.7.3getSize() 229  
 11.1.2getSoundData() 185  
 9.1.5.2.4getSpreadAngle() 64, 154  
 4.3.2getSrcBlendFunction() 48  
 9.1.5getStartPosition() 145  
 6.2.1getStartTime() 79  
 12.7.2getStereoAvailable() 227  
 12.7.2getStereoEnable() 227  
 9.1.3getString() 139  
 8.4.9.3getStripifierStats() 132  
 8.4.9.3.1getStripLengthCounts() 133  
 12.8getSwitch() 232  
 8.4.8.1getTarget() 125  
 9.1.2getTessellationTolerance() 136f.  
 4.2.1getTexture() 35  
 4.2.3getTextureBlendColor() 40  
 4.2.3getTextureMode() 40  
 4.2.3getTextureTransform() 41  
 9.1.4getTolerance() 142  
 8.4.9.3.1getTotalTime() 133  
 8.4.9.3.1getTotalTris() 133  
 9.1.5.4getTransform() 71, 159  
 3.3.1.2.1getTransform3D() 21  
 4.3.2getTransparency() 48  
 4.3.2getTransparencyMode() 48  
 5.2getUpper() 59  
 11.1.1getURLObject() 180  
 11.1.1getURLString() 180  
 12.1getUserData() 212  
 11.2.1getVelocityScaleFactor() 207  
 12.6getView() 224  
 12.6getViewer() 187, 224  
 7.1getViewingPlatform() 16, 87f.  
 7.1getViewPlatformTransform() 88  
 12.9.1getVisible() 239  
 12.4getWhichChild() 219

- 12.7.3getWidth() 229
- 4.1.3Glanzpunkt 31
- 8.4.9.2glatt 130
- 12.7.2GLX 228
- 8.4.9.2Gouraud-Shading 130
- 12.7.3Graphics2D 229
- 3.1GraphicsConfiguration 14
- 12.9.1GREATER 237
- 12.9.1GREATER\_OR\_EQUAL 237
- 3.3.1.2.1Group 20f.
- H
- 3.1Hallo Welt 13
- 9.1.5.4hashCode() 56, 159
- 11.1.3HEADPHONES 187
- 11.1.3Hintergrundgeräusch 188
- I
- 12.9.1ignoreVertexColors 237
- 11.1.1IllegalArgumentException 59, 180
- 3.4.2IllegalSharingException 25
- 12.7.1IllegalStateException 226
- 4.2.1Image 35
- 12.7.1ImageComponent2D 35f., 165, 225f.
- 6.2.1increasing 76f.
- 6.2.1INCREASING\_ENABLE 77, 79
- 8.4.9.1Index 129
- 8.4.7IndexedGeometryArray 110, 113, 117, 120
- 8.4.7IndexedLineArray 106, 120
- 8.4.7IndexedLineStripArray 106, 121
- 10.1.2.1IndexedPointArray 106, 121, 170
- 8.4.6IndexedQuadArray 106, 119f.
- 13IndexedTriangleArray 106, 108, 114, 116ff., 244
- 8.4.6IndexedTriangleFanArray 106, 119f.
- 8.4.4IndexedTriangleStripArray 106, 118
- 12.3indexOfChild() 217
- 8.4indexOfGeometry() 107
- 12.5indexOfScope() 65, 172, 221
- 12.8indexOfSwitch() 232
- 8.4.9.1Indices 110, 114, 116, 128
- 12.5Influencing Bounds 54f., 65f., 82, 172, 174, 220
- 3.1init() 14
- 12.8.1initialize() 81, 91, 235
- 11.1.1InputStream 180
- 12.3insertChild() 52, 217
- 8.4insertGeometry() 107
- 12.5insertScope() 65, 172, 221
- 12.8insertSwitch() 232
- 4.2.2int getFormat() 37
- 7.3.2.4Interpolator 73, 75f., 80ff., 95
- 7.2Interpolatoren 89
- 9.2intersect() 56, 108, 160ff.
- 12.1isCompiled() 212
- 5.1isEmpty() 57

- 12.1.isLive() 212
- 12.7.1.isOffScreen() 226
- 6.2.1.isPaused() 80
- 11.1.2.isPlaying() 185
- 11.1.2.isPlayingSilently() 185
- 11.1.2.isReady() 186
- 12.7.1.isRendererRunning() 226
- 9.1.5.4.isSamePath() 159
- J
- 12.7.3.J3DGraphics2D 228
- 2.1.3.Java WebStart 12
- 12.7.java.awt 14, 136f., 225
- 12.7.3.java.awt.Dimension 229
- 9.1.2.java.awt.Font 137
- 12.7.1.java.awt.image.BufferedImage 226
- 12.7.3.java.awt.Rectangle 228
- 9.1.1.java.awt.Shape 136
- 11.1.1.java.io.InputStream 180
- 12.1.java.lang.Object 212
- 11.1.1.java.lang.String 139, 180
- 11.1.1.java.net.URL 180
- 10.2.java.util 172
- 12.4.java.util.BitSet 219
- 12.8.java.util Enumeration 65, 106, 217, 222, 232
- 6.1.1.javax.media.j3d.Group 71
- 6.1.1.javax.media.j3d.Node 72
- 7.3.2.5.javax.swing.Timer 96
- 2.1.2.2.JDK 9f.
- 12.7.4.JPopupMenu 230
- K
- 7.Kamera 87
- 7.Kameraposition 87
- 7.Kegel 62, 87
- 7.Kegels 87
- 7.2.Keyboard 89
- 7.3.1.KeyEvent 89, 93
- 7.2.KeyListener 89
- 7.3.6.KeyNavigator 100
- 10.1.KeyNavigatorBehavior 100, 163
- 7.3.2.3.Kollision 95
- 12.3.Kollisionen 73, 107, 218
- 9.Kollisionsvermeidung 95, 101, 134
- 8.4.1.Koordinaten 110
- 5.1.Kugel 27, 55
- L
- 12.7.2.LEFT\_EYE\_VIEW 227
- 12.9.1.LESS 237
- 12.9.1.LESS\_OR\_EQUAL 237
- 12.8.Level of Detail 231
- 5.4.Licht 53, 60
- 5.4.Lichtintensität 61
- 5.5.Lichtkegels 63



- 5Lichtquellen 53
- 11.1.2Light 60, 64, 66f., 164, 172, 181
- 4.1.3Lighting 32
- 10.2LinearFog 171
- 8.4.7LineArray 106, 120
- 8.4.7LineStripArray 106, 121
- 13Link 23f., 243
- 2.1.2Linux 9
- 7.2Listener 89
- 12.1live 50, 70f., 82, 165, 210
- 12.8LOD 231f.
- M
- 3.1main() 14
- 13Material 30, 83, 104, 109, 243
- 4Materialeigenschaften 27
- 8.4.9.2Math.toRadians() 131
- 3.3.1.2.2Matrix 21
- 11.1.5MediaContainer 178ff., 185f., 188, 191f., 196f.
- 4.2.4Minification Filter Function 42
- 3.4.2ModelClip 25
- 4.2.3MODULATE 39f.
- 11.1.3MONO\_SPEAKER 187
- 10.1.1Morph 164
- 7.3.5MouseBehavior 100
- 9.1.4mouseClicked() 140
- 9.1.4MouseEvent 141
- 9.1.4MouseListener 140f.
- 13MouseRotate 99f., 167, 187, 243f.
- 7.3.4MouseRotation 98
- 13MouseTranslate 100, 244
- 13MouseZoom 100, 167, 174f., 187, 243f.
- 4.3.2.2moveTo() 52
- 3.3.1.2.2mul() 21
- N
- 7.2Navigation 89
- 12.9.1NEVER 237
- 5nfluencing Bounds 54
- 4.3.2NICEST 40f., 47, 49
- 11.1.5NO\_FILTER 196
- 8.4Node 108
- 9.1.5.4nodeCount() 159
- 3.3Nodes 18
- 4.3.1Nomals 46
- 4.3.2NONE 47f.
- 9.1.5.1.1Normal 111, 129f., 149f.
- 4.2.2NORMAL\_MAP 37
- 13NormalGenerator 126, 129f., 244
- 8.4.9Normals 87, 109, 126
- 8.4.1NORMALS 109, 111
- 12.9.1NOT\_EQUAL 237
- 12.3numChildren() 218
- 12.8.1numDistances() 235

- 8.4numGeometries() 107
- 12.5numScopes() 65, 172, 221
- 12.8numSwitches() 232
- O
- 4.2.2OBJECT\_LINEAR 37f.
- 7.3.3.4ODER 97f.
- 3.1off-screen rendering 14
- 12.7.1off-screen Rendering 225
- 5.5Öffnungswinkel 62f.
- 12.7.2OpenGL 8, 228
- 6.1Optimierung 69
- 12.9OrderedGroup 50ff., 236
- 10.1.1OrientedShape3D 121f., 149, 164
- P
- 6.2.2.2Path 85
- 9.1.3PATH\_DOWN 138f.
- 9.1.3PATH\_LEFT 138f.
- 9.1.3PATH\_RIGHT 138f.
- 9.1.3PATH\_UP 138f.
- 6.2.2.2PathInterpolator 85f.
- 6.2.1Pause 80
- 6.2.1pause() 79
- 6.2Pendelbewegung 73
- 5Performance 54f.
- 11.3PhysicalEnvironment 209
- 6.1.1pickable 73
- 9.1.5.3pickAll( 157
- 9.1.5.3pickAll() 145f., 157
- 9.1.5.3pickAllSorted() 142, 146, 157
- 9.1.5pickAny() 146
- 13PickCanvas 141f., 244
- 9.1.5.3pickClosest() 157
- 9.1.5.2.4PickCone 154f.
- 9.1.5.2.4PickConeRay 144, 154
- 9.1.5.2.4PickConeSegment 144, 155
- 9.1.5.2.3PickCylinder 142f., 152f.
- 9.1.5.2.3PickCylinderRay 144, 153
- 9.2PickCylinderSegment 145, 153, 161
- 13Picking 73, 95, 134, 239, 244
- 9.1.5.1.1PickIntersection 147ff.
- 9.1.5.2.5PickPoint 156
- 9.2PickRay 144f., 148, 151f., 160, 162
- 9.1.5.1.1PickResult 142, 145ff.
- 9.1.5.2.2PickSegment 144f., 148, 152
- 9.2PickShape 143ff., 156f., 161f.
- 9.1.5.4PickTool 143f., 146, 150f., 158
- 9.1.5.1PickTools 146
- 8.4.1Point3d 55, 58, 110
- 8.4.2Point3f 61, 85f., 110, 116f.
- 13PointArray 106, 121, 169f., 244
- 10.1.2PointArrays 169
- 10.1.2.1PointAttributes 169f.

- 13PointLight 31, 60ff., 243f.
- 13PointSound 189ff., 194, 199, 244
- 9.1.1Polygon 102, 136
- 8.4.9.1POLYGON\_ARRAY 127
- 4.3.1POLYGON\_FILL 44, 47
- 4.3.1POLYGON\_LINE 44, 47
- 13POLYGON\_POINT 44, 47, 243
- 13PolygonAttributes 44, 46, 243
- 4.3.1Polygone 33, 44f.
- 8.1Polygonen 102
- 4.3.1Polygonrasterization 46
- 6.2.2.2Position 84f.
- 6.2.2.2PositionInterpolator 84
- 6.2.2.2PositionPathInterpolator 85
- 7.3.2.2postId() 94
- 12.9.2Primitive 27, 53, 87, 105, 149, 158, 240
- 12.8.1processStimulus() 81, 91f., 97, 235
- 12.7.4propertyChange() 231
- 12.7.4PropertyChangeEvent 231
- 4.3.1Punktwolke 44
- 7Pyramide 60, 87
- Q
- 12.9.2Quad 133, 240
- 8.4.9.1QUAD\_ARRAY 127
- 11.1.3QuadArray 106, 119f., 187
- 5.2Quader 58
- 8.4.9.3Quads 120, 131
- 6.2.2.2Quat4f 85f.
- R
- 8.4Raster 106
- 12.9.1rasterOp 237
- 12.9.1rasterOpEnable 237
- 8.1Raumkoordinaten 103
- 5.4Raycasting 60
- 5.4Raytracing 61
- 11.1.2Ready 186
- 7.1Referenz 88
- 11.2.1Reflection Delay 206
- 4.2.2REFLECTION\_MAP 37
- 7reflektieren 87
- 8.3Reflektionsverhalten 30, 104
- 4.1.3Reflexionsverhalten 31
- 12.3removeAllChildren() 52, 217
- 8.4removeAllGeometries() 107
- 12.5removeAllScopes() 66, 173, 221
- 12.8removeAllSwitches() 232
- 12.3removeChild() 52, 217
- 8.4removeGeometry() 107
- 12.5removeScope() 66, 173, 221
- 12.8removeSwitch() 232f.
- 12.9.1RenderingAttributes 51, 236, 238
- 12.7.1renderOffScreenBuffer() 226

4.2.3REPLACE 39  
 12.2RestrictedAccessException 20f., 25, 51f., 65f., 106, 210f., 214ff.  
 6.2.1resume() 80  
 11.2.1Reverbation Bounds 202  
 11.2.1Reverbation Delay 206  
 11.2.1Reverbation Koeffizient 207  
 11.1.2RIFF-WAVE 185  
 12.7.2RIGHT\_EYE\_VIEW 227  
 12.9.1ROP\_COPY 238  
 12.9.1ROP\_XOR 238  
 8.4.8.1ROTATE\_ABOUT\_AXIS 122ff.  
 8.4.8.1ROTATE\_ABOUT\_POINT 122ff.  
 8.4.8ROTATE\_NONE 122f.  
 7.1Rotation 18, 21, 73, 77, 88  
 13RotationInterpolator 73, 75, 78, 81ff., 244  
 6.2.2.2RotationPathInterpolator 85  
 6.2.2.2RotPosPathInterpolator 86  
 6.2.2.2RotPosScalePathInterpolator 86  
 3.3.1.2.2rotX() 21  
 6rotY() 21, 68f.  
 3.3.1.2.2rotZ() 21  
 8.4.9.2rund 130  
 2.1Runtime Environment 8  
 S  
 6.2.2.2Scale 86  
 10.1.1SCALE\_FIT\_ALL 166  
 10.1.1SCALE\_FIT\_MAX 166  
 10.1.1SCALE\_FIT\_MIN 166  
 10.1.1SCALE\_NONE 166  
 10.1.1SCALE\_NONE\_CENTER 166  
 10.1.1SCALE\_REPEAT 166  
 6.2.2.2ScaleInterpolator 84  
 3.3SceneGraph 18  
 12.1SceneGraphObject 210  
 9.1.5.5SceneGraphPath 147, 157f., 160f.  
 12.8.1Scheduling Bounds 75, 82, 93, 185, 235  
 5.6Scope 65  
 4.3.2SCREEN\_DOOR 47  
 2.1SDK 8ff., 101  
 9.1.5.4set() 57, 151, 153, 155f., 159  
 9.1.3setAlignment() 139  
 8.4.8.1setAlignmentAxis() 122, 125  
 8.4.8.1setAlignmentMode() 123, 125  
 6.2.1setAlphaAtOneDuration() 78  
 6.2.1setAlphaAtZeroDuration() 78  
 12.9.1setAlphaTestFunction() 238  
 12.9.1setAlphaTestValue() 238  
 12.3setAlternateCollisionTarget() 217  
 4.1.3setAmbientColor() 31f.  
 11.1.5setAngularAttenuation() 198  
 12.5setAppearance() 106, 114, 222  
 12.5setAppearanceOverrideEnable() 106, 220

11.2.2setApplicationBounds() 208  
 5.4setAttenuation() 61  
 11.2.1setAttributeGain() 204  
 11.2.2setAuralAttributes() 208  
 12.6setBackClipDistance() 224  
 10.2.2setBackDistance() 177  
 11.1.5setBackDistanceGain() 199  
 4.3.1setBackFaceNormalFlip() 46  
 12.2setBounds() 58, 72, 215  
 12.2setBoundsAutoCompute() 72, 215  
 11.1.1setCacheEnable() 180  
 12.1setCapability() 70f., 114, 207, 210  
 12.1setCapabilityIsFrequent() 71, 99, 210  
 5.1setCenter() 56  
 9.1.3setCharacterSpacing() 139  
 12.3setChild() 71, 217  
 4.3.2.2setChildIndexOrder() 51  
 12.4setChildMask() 219  
 12.2setCollidable() 72, 95, 215  
 8.4setCollisionBounds() 72, 106  
 10.2.1setColor() 66, 165, 169, 173, 175  
 8.4.9.1setColorIndices() 112, 129  
 8.4.9.1setColors() 111, 118, 128  
 4.1.3setColorTarget() 31f.  
 5.5setConcentration() 63  
 8.4.8setConstantScaleEnable() 123  
 11.1.2setContinuousEnable() 182  
 10.1.2setCoordinate() 169  
 8.4.9.1setCoordinateIndices() 110, 117, 128  
 8.4.9.1setCoordinates() 110, 117f., 128  
 4.3.1setCullFace() 47  
 11.2.1setDecayFilter() 204  
 11.2.1setDecayTime() 204  
 6.2.1setDecreasingAlphaDuration() 78  
 6.2.1setDecreasingAlphaRampDuration() 78  
 12.7.4setDefaultLightWeightPopupEnabled() 230  
 11.2.1setDensity() 176, 205  
 12.9.1setDepthBufferEnable() 51, 236, 238  
 12.9setDepthBufferWriteEnable() 236  
 12.9.1setDepthBufferWriteEnable() 51, 238  
 11.2.1setDiffusion() 205  
 11.1.5setDirection() 64, 198  
 12.8.1setDistance() 234  
 11.2.1setDistanceFilter() 205  
 11.1.5setDistanceGain() 193, 199  
 12.7.3setDoubleBufferEnable() 228  
 4.3.2setDstBlendFunction() 48  
 11.1.4setEnable() 37, 66, 182, 184, 192  
 9.1.1setExtrusionShape() 136  
 4.2.4setFilter4Func() 42  
 9.1.3setFont3D() 139  
 4.2.2setFormat() 37

11.2.1setFrequencyScaleFactor() 205  
 12.6setFrontClipDistance() 224  
 10.2.2setFrontDistance() 177  
 4.2.2setGenMode() 38  
 10.1.2setGeometry() 105, 108, 165, 169  
 12.9.1setIgnoreVertexColors() 239  
 10.1.1setImage() 165  
 10.1.1setImageScaleMode() 165  
 6.2.1setIncreasingAlphaDuration() 79  
 6.2.1setIncreasingAlphaRampDuration() 79  
 12.5setInfluencingBounds() 28, 64, 66, 164, 171, 173, 222  
 11.1.2setInitialGain() 183  
 11.1.1setInputStream() 180  
 4.1.3setLightingEnable() 32  
 11.1.2setLoop() 183  
 6.2.1setLoopCount() 79  
 5.2setLower() 59  
 4.2.4setMagFilter() 41  
 4.1.3setMaterial() 31  
 6.2.2.1setMaximumAngle() 82f.  
 4.2.4setMinFilter() 42  
 6.2.2.1setMinimumAngle() 82f.  
 6.2.1setMode() 79  
 9.1.5setMode() 141, 143  
 12.7.2setMonoscopicViewPolicy() 227  
 11.1.2setMute() 183f.  
 9.1.5.4setNode() 159  
 9.1.5.4setNodes() 159  
 7setNominalViewingTransform() 16, 87  
 8.4.9.1setNormalIndices() 111, 129  
 8.4.1setNormals() 111  
 8.4.9.1setNormals() 118, 129  
 9.1.5.4setObject() 159  
 12.7.1setOffScreenBuffer() 225  
 9.1.3setPath() 139  
 11.1.2setPause() 184  
 4.2.3setPerspectiveCorrectionMode() 40  
 6.2.1setPhaseDelayDuration() 79  
 12.2setPickable() 73, 216  
 4.2.2setPlaneQ() 38  
 4.2.2setPlaneR() 38  
 5.3setPlanes() 60  
 4.2.2setPlaneS() 38  
 4.2.2setPlaneT() 38  
 10.1.2.1setPointAntialiasingEnable() 170  
 10.1.2.1setPointSize() 170  
 4.3.1setPolygonAttributes() 46  
 4.3.1setPolygonMode() 47  
 12.8.1setPosition() 62, 139, 193, 234  
 11.1.2setPriority() 184  
 5.1setRadius() 56  
 12.9.1setRasterOp() 239

12.9.1setRasterOpEnable() 239  
 11.1.2setRateScaleFactor() 184  
 11.2.1setReflectionCoefficient() 206  
 11.2.1setReflectionDelay() 206  
 11.1.2setReleaseEnable() 185  
 11.2.1setReverbBounds() 206  
 11.2.1setReverbCoefficient() 206  
 11.2.1setReverbDelay() 202, 206  
 11.2.1setReverbOrder() 207  
 11.2.1setRolloff() 207  
 6setRotation() 22, 69  
 8.4.8.1setRotationPoint() 122f., 125  
 9.1setScale() 21, 123, 135  
 12.7.3setSceneAntialiasingEnable() 229  
 11.1.2setSchedulingBounds() 82, 185  
 12.5setScope() 66, 222  
 9.1.5setShape() 144  
 9.1.5setShapeConeRay() 144  
 9.1.5setShapeConeSegment() 144  
 9.1.5setShapeCylinderRay() 144  
 9.1.5setShapeCylinderSegment() 145  
 9.1.4setShapeLocation() 141, 143  
 9.1.5setShapeRay() 145  
 9.1.5setShapeSegment() 145  
 3.4.1setSharedGroup() 24  
 4.1.3setShininess() 32  
 11.1.2setSoundData() 185  
 5.5setSpreadAngle() 64  
 4.3.2setSrcBlendFunction() 48  
 6.2.1setStartTime() 79  
 12.7.2setStereoEnable() 227  
 9.1.3setString() 139  
 12.8setSwitch() 233  
 8.4.8.1setTarget() 125  
 4.2.2setTexCoordGeneration() 36  
 4.2setTexture() 34  
 4.2.3setTextureAttributes() 39  
 4.2.3setTextureBlendColor() 40  
 8.4.9.1setTextureCoordinateIndices() 113, 129  
 8.4.9.1setTextureCoordinates() 112f., 118, 129  
 4.2.3setTextureMode() 40  
 4.2.3setTextureTransform() 41  
 9.1.4setTolerance() 141f.  
 9.1.5.4setTransform() 18, 71, 159  
 6setTransform3D() 21, 68  
 3.3.1.2.2setTranslation() 22  
 4.3.2setTransparency() 48  
 4.3.2setTransparencyAttributes() 47  
 4.3.2setTransparencyMode() 49  
 5.2setUpper() 59  
 11.1.1setURLObject() 180  
 11.1.1setURLString() 180

- 12.1setUserData() 212
- 11.2.1setVelocityScaleFactor() 207
- 12.9.1setVisibilityPolicy() 239
- 12.9.1setVisible() 238
- 12.4setWhichChild() 219
- 9.1.1Shape 136
- 13Shape3D 25, 104f., 108, 114ff., 121f., 142, 147, 149, 151, 158, 160ff., 164, 169, 240, 244
- 13SharedGroup 23, 25, 172, 189, 243
- 3.4Sharing 23
- 4.1.3Shininess 31f.
- 13SimpleUniverse 14, 88, 187, 209, 225, 243
- 3.3.1.2.2Skalierung 21
- 12.9.4Skybox 241f.
- 12.9.3Skydome 240
- 12.9.2Skyplane 236, 240
- 12.9.3Skysphere 240
- 8.4.9.2smooth 131
- 2.1Software Development Kit 8
- 11.1.5Sound 178, 181f., 188f., 192, 194
- 11.1.1SoundException 179
- 13Soundscape 25, 200, 244
- 8.3Specular 104
- 4.1.3Specular Color 31ff.
- 5.1Sphäre 55
- 13Sphere 27, 166f., 243f.
- 5.5Spot 62
- 5.5Spotlight 63
- 13SpotLight 62, 64, 243f.
- 11.1.3STEREO\_SPEAKERS 187
- 7.3Stimulus 92
- 13Stripifier 131f., 244
- 8.4.9.3.1StripifierStats 132
- 8.4.9.3stripify() 132
- 4.2Strukturen 33
- 11.1.2Sun-Audio 185
- 12.4Switch 172, 218
- T
- 9.2Terrain Following 101, 161f.
- 8.4.1TexCoord2f 109
- 8.4.1TexCoord3f 109
- 8.4.1TexCoord4f 109
- 13TexCoordGeneration 36, 38, 244
- 13Text3D 106, 134f., 138, 140, 244
- 13Textur 35, 244
- 13Texture 35, 41, 244
- 9.1.5.1.1TEXTURE\_COORDINATE\_2 37, 109, 112f., 149
- 8.4.1TEXTURE\_COORDINATE\_3 37, 109, 113
- 8.4.1TEXTURE\_COORDINATE\_4 37, 109, 113
- 4.2.4Texture2D 41
- 4.2.4Texture3D 41
- 13TextureAttributes 39, 244



13TextureLoader 35f., 244  
 4.2.1TextureLoaders 35  
 4.2Texturen 33  
 11.1.3Texturkoordinaten 35f., 38, 103, 112, 187  
 11.1.5Tiefpaß 194  
 6.2.2.1Timer 68, 82  
 8.4.9.2toRadians() 131  
 5.1transform() 57  
 9.1Transform3D 18, 20f., 39, 75, 88, 99, 121, 135  
 8.4.8Transformation 21, 84, 121  
 10.2TransformGroup 18, 20, 71, 88, 99, 121, 124, 135, 172  
 6.2.2.2TransformInterpolator 84f.  
 6.2.2.2TransformInterpolators 84  
 13TransparencyAttributes 47, 49f., 84, 243  
 6.2.2.2TransparencyInterpolator 84  
 4.3transparente Objekte 42  
 6.2.2.2Transparenz 47, 49, 84  
 8.4.1TriangeArray 110  
 8.4.9.1TRIANGLE\_ARRAY 127  
 8.4.9.1TRIANGLE\_FAN\_ARRAY 127  
 8.4.9.1TRIANGLE\_STRIP\_ARRAY 127  
 8.4.4TriangleArray 106, 117f.  
 8.4.5TriangleFanArray 106, 119  
 8.4.4TriangleStripArray 106, 118  
 U  
 5Umgebungslicht 28, 53  
 7.3.3.4UND 97f.  
 10.1.1Universe 14, 165  
 7.1Universum 14, 88  
 12.2updateNodeReferences() 212, 214  
 4.2.1URL 35  
 7.3.2.3USE\_BOUNDS 95  
 7.3.2.3USE\_GEOMETRY 95  
 8.2UV-Koordinaten 103  
 V  
 6.2.1value() 80  
 8.4.8Vector3f 122  
 5.3Vector4d 59  
 6.2Verschiebung 21, 73  
 10.1.2Vertex 44, 102, 110, 128, 169  
 8.4.2Vertexkoordinaten 116  
 8.4.2Vertices 44, 102f., 110, 114, 116  
 4.3.1Vielecke 45  
 4.3.1Vierecke 45  
 12.9.1View 227, 229, 238f.  
 11.1.3Viewer 187  
 7.3.2.1ViewingPlatform 16, 88ff., 93

- 7.3.3.4WakeupAnd 97f.
- 7.3.3.3WakeupAndOfOrs 97
- 7.3.3.3WakeupCondition 96f.
- 13WakeupCriterion 89, 91f., 96f., 235, 243
- 7.3.3wakeupOn() 92, 96
- 7.3.2.1WakeupOnActivation 93
- 13WakeupOnAWTEvent 92f., 243
- 7.3.2.2WakeupOnBehaviorPost 94
- 7.3.2.3WakeupOnCollisionEntry 94
- 7.3.2.3WakeupOnCollisionExit 94
- 7.3.2.3WakeupOnCollisionMovement 94
- 7.3.2.4WakeupOnElapsedFrames 95
- 7.3.2.5WakeupOnElapsedTime 96
- 7.3.3.3WakeupOr 97
- 7.3.3.4WakeupOrOfAnds 98
- 2.1.3WebStart 12
- 10.1.2Windows 9, 169
- Y
- 4.2.1Y\_UP 35
- Z
- 12.9.3Z-Buffer 240
- 4.3.2.2Z-Order 49f.
- 4.3Zylinder 43
- 6.1.1 71
- 4.2.4 Filter Function 41
- 4.1.3 SPECULAR 32