Goerdeler-Gymnasium Paderborn

Facharbeit

Thema: Die Klassenbibliothek Java 3D™ am Beispiel eines einfachen Planetensystems

Fach: Informatik
Kurs: GK
Fachlehrer: Herr Hilwerling

Autor: Daniel Dreibrodt
XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX

Abgabe: Donnerstag, 13.03.2008

1 Eigens angepasste Version eines Bildes vom Java-Maskottchen „Duke“ mit Helm.
Quelle des Originalbildes: siehe http://duke.dev.java.net/images/JDK6/index.html (15.02.08 15:00)

Inhaltsverzeichnis

Einleitung 3
1 Einführung 3
2 Struktur und Anwendung 3
2.1 Die Grundstruktur 3
2.1.1 VirtualUniverse und Locale 4
2.1.2 Gruppen 4
2.1.3 Blätter des Szenengraphen 5
2.1.4 Darstellung auf dem Bildschirm 6
2.1.5 Vereinfachungen durch ein SimpleUniverse 8
2.2 Erzeugung einfacher Körper 8
2.3 Das Aussehen von Körpern 8
2.3.1 Material 8
2.3.2 Textur 9
2.3.3 Transparenz 10
2.4 Transformation 10
2.4.1 Verschiebung 11
2.4.2 Rotation 11
2.4.3 Skalierung 11
2.5 Animation 12
2.5.1 Alpha 12
2.5.2 SchedulingBounds 13
2.6 Beleuchtung 13
2.6.1 AmbientLight 13
2.6.2 DirectionalLight 13
2.6.3 PointLight 14
2.6.4 SpotLight 14
3 Anwendungsbeispiel 14
3.1 Einleitung 14
3.2 Aufbau 14
Zusammenfassung 17
Literaturverzeichnis 18
Bildquellen 18
Technische Hilfsmittel 18
Selbstständigkeitserklärung 19

Einleitung

In der vorliegenden Facharbeit möchte ich mich mit den Möglichkeiten der dreidimensionalen Darstellung in Java mithilfe von Java 3D befassen. Es sollen die Grundlagen zur Anwendung der Klassenbibliothek Java 3D vermittelt und diese abschließend anhand eines Anwendungsbeispiels erläutert werden.

Das Beispielprogramm befindet sich zusammen mit dessen Quelltext, sowie den benutzten Quellen und dieser Facharbeit im PDF-Format auf der beigefügten CD.

1 Einführung

Java 3D ist eine Sammlung von Klassen, die die Java API um die Möglichkeit erweitern, die 3D-Grafikbeschleunigung von Grafikkarten anzusprechen. Somit ermöglicht sie es, dreidimensionale Szenen in Echtzeit mit Java darzustellen.

Bis zur Version 1.4 gab es zwei Varianten von Java 3D, eine die als Schnittstelle zur 3D-Beschleunigung der Grafikkarte OpenGL nutzte und eine die DirectX Graphics benutzte.

Seit der Version 1.5 liegt Java 3D nur noch in einer Version vor, welche sowohl OpenGL als auch DirectX ansprechen kann. Standardmäßig wird OpenGL als 3D-Beschleunigung verwendet, da dies standardmäßig auf vielen Betriebssystemen verfügbar ist, DirectX jedoch nur auf Windows.[1] Falls jedoch DirectX 9.0 oder höher installiert ist, kann auch DirectX als 3D-Beschleunigung verwendet werden, indem man `-Dj3d.rend=d3d` der Java VM als Attribut übergibt.[2]

Voraussetzung zur Installation von Java 3D ist ein installiertes Java Runtime Environment der Version 1.5 oder höher. Ein passendes Java Runtime Environment erhält man auf der Java Homepage bei Sun (http://java.sun.com/javase/downloads/).

Die aktuelle Version von Java 3D, welche zurzeit 1.5.1 ist, erhält man auf der Java 3D Homepage bei java.net (http://java3d.dev.java.net/binary-builds.html).

2 Struktur und Anwendung

2.1 Die Grundstruktur

Dreidimensionale Szenen sind in Java 3D in einem Graphen mit baumartiger Struktur aufgebaut. Diesen Graphen nennt man Szenengraph. Im Folgenden werden die verschiedenen Element dieses Szenengraphen und dessen Aufbau erläutert.

1 siehe Wikipedia – Die Freie Enzyklopädie, Artikel DirectX, im Folgenden [1] benannt
2 siehe Dr. Andrew Davidson, Pro Java™ 6 3D Game Development, Online-Version, S. 3, im Folgenden [2] benannt

2.1.1 VirtualUniverse und Locale

An oberster Stelle des Szenengraphen steht ein Objekt der Klasse *VirtualUniverse*. Diese Klasse repräsentiert genau eine dreidimensionale Umgebung, folglich können in ihr enthaltene Objekte nicht gleichzeitig Teil eines anderen *VirtualUniverse* sein. Das *VirtualUniverse* enthält eine Sammlung von Objekten des Typs *Locale*.[1]

Solch ein *Locale* repräsentiert eine sehr präzise, dreidimensionale Koordinate, welche als Koordinatenursprung für alle Objekte, die an dessen Graphen hängen, dient.[2] Dieses Koordinatensystem ist ein rechtshändiges, wie es in Abbildung 1 dargestellt ist[3], in welchem alle Koordinaten als `float`-Werte angegeben werden.

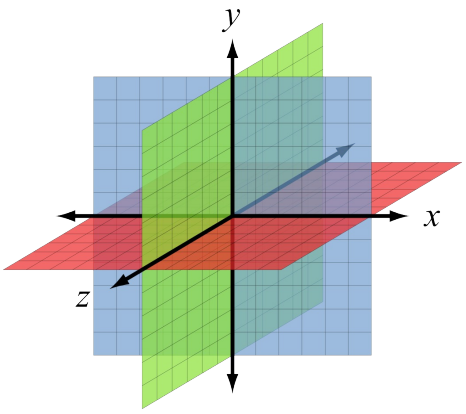

Abbildung 1: Ein rechtshändiges Koordinatensystem

2.1.2 Gruppen

Dreidimensionale Körper werden in Java 3D nicht direkt an ein *Locale* angebracht. Sie werden in sogenannten *Groups* gespeichert.

Eine *Group* im Allgemeinen erlaubt es mehrere dreidimensionale Objekte und Funktionen zu gruppieren und somit zusammen anzusprechen und zu verarbeiten. *Groups* können ferner auch weitere *Groups* enthalten[4]

Die beiden wichtigsten Unterklassen von *Group* sind *BranchGroup* und *TransformGroup*.

BranchGroup

Die Klasse *BranchGroup* erweitert die *Group* zum einen um die Möglichkeit, sie an ein *Locale* anzuhängen, und zum anderen um die Möglichkeit, die in ihr enthaltenen Objekte zu „kompilieren“.[5] Als einzige *Group*, die einem Locale hinzugefügt werden kann, ist die *BranchGroup* ein essentieller Bestandteil einer Szene.

Die Kompilierung einer *BranchGroup* mithilfe der Methode `BranchGroup.compile()` beschleunigt den Darstellungsprozess, indem gleiche Eigenschaften von enthaltenen Objekten zusammengefasst werden. Dies ist jedoch nur

1 siehe Java 3D 1.5.1 API Dokumentation, javax.media.j3d.VirtualUniverse Documentation, im Folgenden [3] benannt
2 siehe [3], javax.media.j3d.Locale Documentation
3 siehe Wikimedia Commons – Image:3D_coordinate_system.svg
4 siehe [3], javax.media.j3d.Group Documentation
5 siehe [3], javax.media.j3d.BranchGroup Documentation

bei großen Szenen mit vielen gleichartigen Objekten nützlich, da ansonsten kaum eine Verbesserung der Leistung auftritt.[1]

TransformGroup

Die *TransformGroup* ist eine spezielle *Group*, welche es ermöglicht, die in ihr enthaltenen Bestandteile einer bestimmten Transformation (Ort, Größe, Rotation etc.), repräsentiert durch ein Objekt des Typs *Transform3D*, zu unterziehen.[2] Auf die Möglichkeiten die eine *TransformGroup* bietet wird in Kapitel 2.4 näher eingegangen.

2.1.3 Blätter des Szenengraphen

Innerhalb von einer *Group* können nun die eigentlichen Objekte die eine Szene bestimmen abgelegt werden. Dies sind die Blätter des baumartigen Szenengraphen und sind dementsprechend Unterklassen von *Leaf*.[3] Im folgenden werden die wichtigsten Blattklassen kurz erläutert.

Shape3D

Die Klasse *Shape3D* ist die wohl wichtigste Blattklasse, da sie die eigentlichen dreidimensionalen Körper darstellt. Die Form eines durch eine *Shape3D* repräsentierten Körpers wird durch ein Objekt des Typs *Geometry* festgelegt, welches ein Attribut der *Shape3D* ist. Ein weiteres, optionales Attribut einer *Shape3D* ist ein Objekt des Typs *Appearance*, welches, wie schon der Name sagt, das Aussehen des Körpers festlegt.[4]

Auf die Verwendung der Klasse *Appearance* wird im Kapitel 2.3 näher eingegangen.

Behavior

Die Klasse *Behavior* ist eine abstrakte Oberklasse, welche die Möglichkeit bietet auf eine bestimmte Ausgangsbedingung hin, andere Objekte innerhalb des Szenengraphen anzusprechen und zu verändern. Mit Unterklassen von *Behavior* werden unter anderem Animation und benutzergesteuerte Navigation realisiert.[5]

Wie man *Behaviors* nutzt um Animationen zu erzeugen wird in Kapitel 2.6 erläutert.

Background

Wie der Name dieser Klasse bereits aussagt, wird mit einem *Background*-Objekt der Hintergrund der Szene festgelegt. Der Hintergrund einer Szene kann entweder aus einer Farbe, einem Bild oder einem Körper bestehen. Somit wird einem *Background* bei der

1 siehe J3D.ORG – Tutorials – BranchGroup.compile(), im Folgenden [4] benannt
2 siehe [3], javax.media.j3d.TransformGroup Documentation
3 siehe [3], javax.media.j3d.Leaf Documentation
4 siehe [3], javax.media.j3d.Shape3D Documentation
5 siehe [3], javax.media.j3d.Behavior Documentation

Erzeugung entweder ein Farbwert des Typs *Color3f*, ein Bild vom Typ *ImageComponent2D* oder ein Körper verpackt in einer *BranchGroup* übergeben.[1]

Wenn ein Hintergrundbild gewählt wurde, wird es auf die Größe des dargestellten Bereichs proportional skaliert und dann im Hintergrund gezeichnet. Handelt es sich um einen Körper, wird dieser auf eine Einheitskugel projiziert, so dass er immer in der gleichen Größe dargestellt wird unabhängig von der Position des Betrachters.

Light

Light ist eine abstrakte Klasse, welche eine Lichtquelle repräsentiert, und somit essentiell für jedes Java 3D Programm. Auf die Benutzung von *Lights* wird im Kapitel 2.6 näher eingegangen.[2]

Sound

Die abstrakte Klasse *Sound* ermöglicht es, in der Szene Tonsignale abzuspielen.

Die zwei Unterklassen *BackgroundSound* und *PointSound* erlauben es, wie es schon der Name sagt, entweder Töne überall in der Szene gleicht laut erklingen zu lassen oder von einem bestimmten Ort in der Szene aus abzuspielen. Bei einem *PointSound* wird also die Position des Betrachters ausgewertet, um die Lautstärke und Richtung aus der die Audiosignale ausgegeben werden zu errechnen.[3]

2.1.4 Darstellung auf dem Bildschirm

Zur Darstellung der Szene auf dem Bildschirm bietet Java 3D, analog zur Klasse *Canvas* der Standard Java API, mit welcher zweidimensional auf den Bildschirm gezeichnet werden kann, die Klasse *Canvas3D*. Da diese eine Unterklasse von *java.awt.Component* ist, lässt sie sich nahtlos in herkömmliche, auf AWT oder Swing basierende Programme einfügen.[4]

Damit ein *Canvas3D* die Informationen bekommt, die er benötigt um eine Szene darzustellen, muss er von einem sogenannten *View* gesteuert werden.

Ein *View* verwaltet, wie die Szene auf einem *Canvas3D* dargestellt wird. Zum Beispiel wird die Entfernung bis zu welcher die Szene dargestellt werden soll oder ob Kantenglättung benutzt werden soll hier festgelegt.

1 siehe [3], javax.media.j3d.Background Documentation
2 siehe [3], javax.media.j3d.Light Documentation
3 siehe [3], javax.media.j3d.Sound Documentation
4 siehe [3], javax.media.j3d.Canvas3D Documentation

Aber sowohl *View* als auch *Canvas3D* befinden sich außerhalb des Szenengraphen und benötigen deshalb eine Anschlussstelle an die eigentliche Szene. Diese Anschlussstelle ist die *ViewPlatform*.

Um nun die Kommunikation zwischen *Canvas3D* und der *ViewPlatform* herzustellen, muss dem *View*, der sozusagen als Vermittler dient, sowohl der *Canvas3D* als auch die *ViewPlatform* bekannt sein.[1]

Die *ViewPlatform* ist ein Objekt innerhalb des Szenengraphen, das festlegt, von wo aus die Szene betrachtet wird. Ferner bestimmt sie über die Blickrichtung aus der betrachtet wird und über die Größe des Blickfelds.[2] Damit der Betrachterstandort veränderbar ist, muss folglich die *ViewPlatform* innerhalb einer *TransformGroup* angelegt werden. Um die eventuell mehreren *ViewPlatforms* und die eigentlichen Körper einer Szene zu trennen ist es sinnvoll, die *ViewPlatforms* enthaltenden *TransformGroups* in einer eigenen *BranchGroup* direkt dem *Locale* einer Szene hinzuzufügen.

Somit ergibt sich folgende Struktur für ein allgemeines Java 3D Programm[3]:

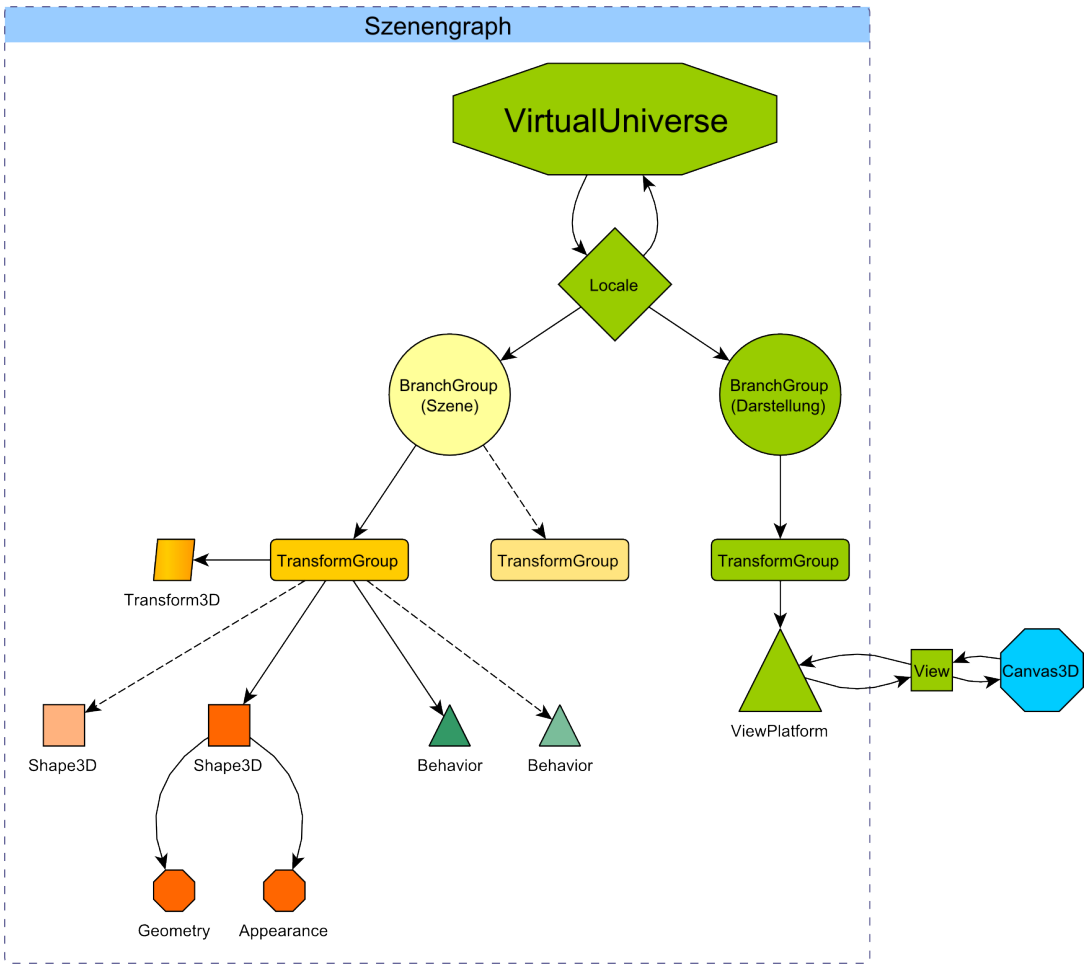

Abbildung 2: Aufbau eines typischen Java 3D Programms

1 [3], javax.media.j3d.View Documentation
2 [3], javax.media.j3d.ViewPlatform Documentation
3 Vergleiche [2], S. 6

2.1.5 Vereinfachungen durch ein SimpleUniverse

Die Klasse *SimpleUniverse*, eine Unterklasse von *VirtualUniverse,* vereinfacht den Erstellungsprozess für einen Szenengraphen sehr stark. Mit Erzeugung eines *SimpleUniverse* wird sofort der gesamte, in Abbildung 2 grün gezeigte, für die Darstellung der Szene auf dem Bildschirm benötigte Teil eines Szenengraphen automatisch erstellt.[1] Somit ist es möglich mit nur zwei Anweisungen bereits eine leere Szene, die in einem *Canvas3D* dargestellt wird, zu erzeugen:

```
Canvas3D c3d = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
SimpleUniverse sUni = new SimpleUniverse(c3d);
```

2.2 Erzeugung einfacher Körper

Die Package *com.sun.j3d.utils.geometry* bietet unter anderem auch diverse Klassen zur Erzeugung primitiver geometrischer Objekte, die dann im Szenengraphen platziert werden können.[2]

Unter Angabe der Maße und des Aussehens können mit diesen Hilfsklassen Quader, Zylinder, Kegel und Kugeln erzeugt werden. Die entsprechenden Klassen heißen *Box*, *Cylinder*, *Cone* und *Sphere*.

Da die Möglichkeiten zur Modellierung komplexerer Körper in Java 3D sehr umfangreich sind, wird darauf an dieser Stelle nicht eingegangen.

2.3 Das Aussehen von Körpern

Das äußerliche Erscheinungsbild von Körpern wird in Java 3D mithilfe der Klasse *Appearance* geregelt. Mit ihr können unter anderem das Material, die Textur und die Transparenz eines Körpers definiert werden. Im folgenden werden diese drei wichtigsten Eigenschaften erläutert.

Um einem Körper anschließend die gewünschte *Appearance* zuzuweisen, muss sie entweder im Konstruktor angegeben werden oder nachträglich per `setApperance(Appearance a)` gesetzt werden.

2.3.1 Material

In der Klasse *Material* wird gespeichert, wie ein Körper auf Beleuchtung reagiert, sozusagen aus welchem Material er besteht.

1 siehe Frank Klawonn, Grundkurs Computergrafik mit Java, 1. Auflage, Wiesbaden: Vieweg 2005, S. 127f., im Folgenden [5] benannt
2 siehe [3], j3d.utils.geometry Package summary

Die Eigenschaften eines Materials werden durch die fünf folgenden Attribute festgelegt, die dem Konstruktor folgendermaßen übergeben werden:

```
new Material(Color3f ambientColour, Color3f emissiveColour,
             Color3f diffuseColor, Color3f specularColor,
             float shininessValue);
```

ambientColour

„Die Farbe `ambientColour` gibt an, wie viel Streulicht das Material reflektiert.“[1] Das heißt, wenn in der Szene eine Streulichtquelle vorliegt gibt dieser Farbwert an, in welcher Intensität das von der Quelle ausgehende Licht reflektiert wird.

emissiveColour

Dieser Wert legt die Eigenleuchtkraft des Materials fest, das heißt die Farbe, in der der Körper an den Stellen erscheint, auf die kein Licht fällt.

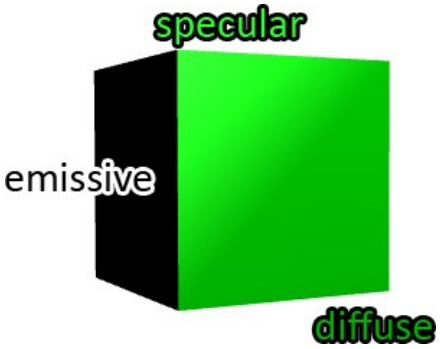

Abbildung 3: Auswirkung der verschiedenen Attribute eines Materials

diffuseColor

Dieser Farbwert gibt an, mit welcher Intensität einfallendes Licht vom Körper gestreut wird.

specularColor

Dieser Wert wiederum gibt an, wie stark einfallendes Licht direkt reflektiert wird.

shininessValue

„Der letzte Parameter `shininessValue` ist ein `float`-Wert zwischen 1 und 128, der angibt, wie glänzend die Oberfläche ist. Je größer dieser Wert ist, desto glänzender ist die Oberfläche.“[2]

Wenn nun ein *Material* nach Wunsch erstellt wurde, fügt man es der *Appearance* via `setMaterial(`**`Material`** `m)` hinzu.

2.3.2 Textur

Um der Oberfläche eines Körpers eine Textur zuzuweisen, gibt es die Klasse *Texture*. Um ein Bild als Textur zu laden gibt es den *TextureLoader*, mit dem man ein Bild aus dem Dateisystem laden und in ein *Texture*-Objekt umwandeln kann.

1 siehe [5], S. 218
2 siehe [5], S. 218

Somit kann man eine Textur über folgende Anweisung erstellen[3]:

```
Texture t = new TextureLoader(String dateipfad, ImageObserver
                observer).getTexture();
```

Der *ImageObserver* kann auch auf **null** gesetzt werden, er ist nicht zwingend zur Erzeugung der Textur nötig.

Anschließend fügt man die Textur der Appearance mit `setTexture(Texture t)` hinzu.

2.3.3 Transparenz

Um die Transparenz eines Körpers festzulegen, bietet Java 3D die Klasse *TransparencyAttributes*. In ihr wird festgelegt mit welcher Methode die Transparenz dargestellt wird und zu welchem Grad der Körper sichtbar sein soll. Auf die verschiedenen Methoden zur Darstellung von Transparenz wird in dieser Arbeit nicht eingegangen, da die *TransparencyAttributes* es auch ermöglichen, je nach vorhandener Hardware die beste Methode automatisch auszuwählen.

Der Grad der Transparenz wird in einem `float`-Wert von 0 (komplett sichtbar) bis 1 (komplett transparent) festgelegt.

Um die am besten aussehende Transparenz für einen Körper festzulegen benutzt man folgende Anweisung:

```
TransparencyAttributes ta = new TransparencyAttributes(
                    TransparencyAttributes.NICEST, float transparenzgrad);
```

Die *TransparencyAttributes* einer *Appearance* stellt man via `setTransparencyAttributes(TransparencyAttributes ta)` ein.[2]

2.4 Transformation

Der in der dreidimensionalen Modellierung essentielle Bereich der Transformation wird in Java 3D mithilfe der Klassen *TransformGroup* und *Transform3D* geregelt. Ein *Transform3D*-Objekt stellt eine Transformation dar, die auf alle Elemente einer *TransformGroup* angewendet wird.

Mit einem *Transform3D* sind alle Transformationen möglich, die durch eine Matrix dargestellt werden können. Auf diese komplexe Ebene von *Transform3D* wird hier aber nicht näher eingegangen. Vielmehr werden im folgenden einfache Methoden zur Verschiebung, Rotation und Größenveränderung vorgestellt.

3 siehe [3], com.sun.j3d.utils.image.TextureLoader Documentation
2 siehe [3], javax.media.j3d.TransparencyAttributes Documentation

Damit man die folgenden Transformationen und auch die später beschriebenen Animationen auch verändern bzw. anwenden kann, nachdem die Szene gestartet ist, muss die entsprechende *TransformGroup* darauf vorbereitet werden. Dies wird mit der Methode `setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE)` der entsprechenden *TransformGroup* erreicht.

2.4.1 Verschiebung

Um eine Verschiebung, auch Translation genannt, auf einen Körper anzuwenden, muss ein Vektor erstellt werden, der diese Verschiebung repräsentiert. Um ein Objekt z. B. eine Einheit in Richtung der *x*-Achse, zwei Einheiten in y-Richtung und eine Einheit nach hinten auf der *z*-Achse zu verschieben muss zunächst folgender Vektor erstellt werden:

```
Vector3f v = new Vector3f(1.0f, 2.0f, -1.0f);
```

Der gewünschte Verschiebungsvektor wird dann dem *Transform3D*-Objekt hinzugefügt via `setTranslation(Vector3f v)`.[1]

2.4.2 Rotation

Ein *Transform3D*-Objekt bietet drei Methoden um eine Rotation um die Achsen des Koordinatensystems durchzuführen. Der Winkel um den rotiert wird, wird hierbei in Radianten angegeben.

Die entsprechenden Methoden lauten:

```
rotX(float winkel), rotY(float winkel), rotZ(float winkel)
```

Man kann auch um eine beliebige andere Achse rotieren. Diese Achse wird durch einen Vektor (x, y, z) repräsentiert. Die entsprechende Rotation wird dann über die Methode `set(new AxisAngle4d(float x, float y, float z, float winkel)` gesetzt.[2]

2.4.3 Skalierung

Um Objekte um einen bestimmten Faktor zu skalieren, bietet ein *Transform3D*-Objekt die Methode `setScale(float faktor)`. Wenn man in x-, y- und z-Richtung verschieden stark skalieren möchte kann man der Methode `setScale` auch einen Vector3f übergeben, der dann die Skalierungsfaktoren in die drei Richtungen angibt.[3]

1 siehe [5], S. 122
2 siehe [5], S. 122
3 siehe [5], S. 122

2.5 Animation

Alle Animationen in Java 3D werden von sogenannten *Interpolator*-Objekten geregelt.[1] Diese werden zu einer *TransformGroup* hinzugefügt und beeinflussen alle Element die sich in dieser Gruppe befinden.

Es gibt verschiedene Interpolatoren, wie z. B. Den *PositionInterpolator* zur linearen Bewegung entlang einer Achse oder den *RotationInterpolator* zur Rotation um eine Achse. Um den Anfangs- und Endstatus von solchen Interpolatoren zu setzen, geht man analog zur Verwendung von Transformationen vor, weshalb hier nicht näher auf diesen Aspekt eingegangen wird. Vielmehr wird im folgenden darauf eingegangen, wie der zeitliche Ablauf einer Animation gesteuert wird.

2.5.1 Alpha

Von welchem Zeitpunkt an, wie lange, wie oft, wie schnell und in welche Richtung die Animationen laufen sollen, wird dem jeweiligen *Interpolator* mit einem Objekt des Typs *Alpha* mitgeteilt.

Die Anzahl der Wiederholungen der Animation wird mit dem `float`-Wert `loopCount` angegeben, wobei -1 unendlich vielen Wiederholungen entspricht.

In welche Richtung die Animation zwischen Anfangs- und Endstatus ablaufen soll wird durch die Ganzzahl `mode` angegeben, welche aus den Konstanten Alpha.INCREASING_ENABLE und Alpha.DECREASING_ENABLE gebildet wird. Die erstere Konstante allein bewirkt, dass die Animation von Anfangsstatus bis Endstatus abläuft. Die zweite Konstante bewirkt das genaue Gegenteil. Die Summe aus beiden Konstanten ermöglicht eine Schleife vom Anfangsstatus über Endstatus wieder hin zum Anfangsstatus.

Der zeitliche Ablauf einer Animation wird durch mehrere `long`-Werte angegeben, jeweils in Millisekunden. Der Wert `triggerTime` gibt an, nach wie vielen ms ab Programmstart die Animation starten soll. Nach dem Start der Animation wird dann noch `phaseDelayDuration` ms lang gewartet, bis die eigentliche Animation beginnt. Dann wird für `increasingAlphaDuration` ms das Objekt von seinem Anfangsstatus in seinen Endstatus gebracht. Dabei wird durch den Wert `increasingAlphaRampDuration` festgelegt, wie lange in dieser „increasing"-Phase die Animation braucht, um auf volle Animationsgeschwindigkeit linear zu beschleunigen und wieder abzubremsen. Wenn der Endstatus erreicht wurde, wird dort für

1 siehe [5], S. 135

`alpaAtOneDuration` ms gewartet. Die Werte `decreasingAlphaDuration`, `decreasingAlphaRampDuration` und `alphaAtZeroDuration` gelten entsprechen für die Animationsphase vom Endstatus wieder hin zum Anfangsstatus.[1]

2.5.2 SchedulingBounds

Damit eine Animation überhaupt dargestellt wird muss einem Interpolator ein Wirkungsbereich zugeteilt werden. Dieser Bereich, in Form einer Kugel, legt fest, unter welchen Umständen die Animation überhaupt berechnet und angezeigt werden soll. Befindet sich der Betrachter innerhalb dieses Wirkungsbereichs wird die Animation berechnet.[2]

Sollte z. B. der Interpolator `i` im Umkreis von r Einheiten um den Punkt (x, y, z) wirksam sein, würde dies durch folgende Anweisung festgelegt:

```
i.setSchedulingBounds(new BoundingSphere(new Point3d(x, y, z), r));
```

2.6 Beleuchtung

In Java 3D gibt es vier verschiedene Arten von Lichtquellen, welche im folgenden kurz erläutert werden.

2.6.1 AmbientLight

Mithilfe der Klasse *AmbientLight* kann man einer Szene Streulicht hinzufügen.

```
AmbientLight al = new AmbientLight(Color3f farbe);
```

Den Wirkungsbereich eines AmbientLight setzt man, ähnlich wie bei Interpolatoren mit einer *BoundingSphere*. Dieser wird bei *AmbientLight* und allen anderen Lichtquellen mit der Methode `setInfluencingBounds(BoundingSphere bs)` gesetzt.[3]

2.6.2 DirectionalLight

Mit *DirectionLight* erstellt man eine Quelle paralleler Lichtstrahlen. Hierzu muss nicht nur die Farbe des Lichts, sondern auch die Richtung der Lichtstrahlen in Form eines Vektors angegeben werden.[4]

```
DirectionalLight dl = new DirectionalLight(Color3f farbe,
                                           Vector3f richtung);
```

1 siehe [5], S. 136 ff.
2 siehe [5], S. 140
3 siehe [5], S. 207
4 siehe [5], S. 207

2.6.3 PointLight

Ein *PointLight* repräsentiert eine punkförmige Lichtquelle, die zur Erzeugung die Angabe von Farbe, Ursprung und Dämpfung benötigt.[1]

```
PointLight pl = new PointLight(Color3f farbe, Point3f ursprung,
                               Point3f daempfung);
```

2.6.4 SpotLight

Ein Scheinwerfer wird mithilfe der Klasse *SpotLight* realisiert. Zusätzlich zu den Angaben zur Farbe, dem Ursprung, der Dämpfung, der Richtung und des Öffnungswinkels benötigt er auch eine Angabe zur Konzentration. Bei einer Konzentration von 0 „nimmt die Lichtintensität bis zum Rand des Kegels nicht ab, um dann abrupt auf Null zu sinken. Der Wert 120 liefert einen Scheinwerfer, dessen Lichtstrahl sehr stark in der Mitte des Lichtkegels konzentriert ist.“[2]

```
SpotLight sl = new SpotLight(Color3f farbe, Point3f ursprung,
                             Point3f daempfung, Vector3f richtung,
                             float oeffnungswinkel, float konzentration);
```

3 Anwendungsbeispiel

3.1 Einleitung

Als Beispiel zur Anwendung von Java 3D soll ein Programm entworfen werden, welches ein einfaches Planetensystem darstellt, in dem die Umlaufbahn eines Trabanten um einen Planeten realitätsnah dargestellt werden soll. Der Benutzer soll die Möglichkeit haben Masse von Planet und Trabant, deren Abstand und die Startgeschwindigkeit des Trabanten zu bestimmen.

3.2 Aufbau

Als Hauptklasse des Anwendungsbeispiels dient die Klasse *Main*. Sie ist vom Typ *JFrame* und verwaltet die Darstellung der 3D-Szene auf dem Bildschirm mithilfe eines *Canvas3D*-Objektes und die gesamte Benutzeroberfläche.

Die anderen beiden Klassen aus denen das Anwendungsbeispiel besteht sind *Scene* und *MoonMover*. Beide werden in *Main* erzeugt und angesprochen.

In *Scene* wird der gesamte Szenengraph verwaltet. Zunächst wird ein *SimpleUniverse* `sUni` erstellt. In einer *BranchGroup* werden dann zwei *TransformGroups* `planet_tfg` und `moon_tfg` erzeugt.

1 siehe [5], S. 208
2 siehe [5], S. 208

Mit `createPlanet(planet_tfg)` und `createMoon(moon_tfg)` werden Planet und Mond erzeugt und dann in ihre entsprechenden *TransformGroups* eingefügt.

`createPlanet` erstellt zunächst die *Appearance* für den Planeten. Dieser wird dann eine aus der Datei `erde.png` erzeugte Textur und ein grünblau erscheinendes Material zugewiesen. Danach wird ein *RotationInterpolator* samt *Alpha*-Werten erzeugt, der die Erde sich endlos drehen lässt.

`createMoon` erzeugt analog zu `createPlanet` eine *Appearance* aus einer Mondtextur und einem weiß erscheinendem Material. Dann wird ein *Transform3D* erzeugt, was eine Verschiebung des Mondes auf seine Anfangsposition darstellt. Diese wird dann auf `moon_tfg` angewandt.

Anschließend wird in *Scene* mit `addLight` dem Szenengraphen eine *BranchGroup* mit einem *DirectionalLight* in hellgelber Farbe und geringer Intensität zugefügt.

Abschließend wird der Szene noch ein Sternenhintergrund festgelegt, die Betrachterposition so eingestellt, dass alles im Abstand von einer Einheit vom Ursprung entfernt ist sichtbar ist, und die Betrachterposition mithilfe eines *OrbitBehaviors* mit der Maus veränderbar gemacht.

Wenn die Szene erstellt wurde, wird, sobald der Benutzer auf einen Startbutton in der von *Main* geregelten Benutzeroberfläche geklickt hat, ein *MoonMover*-Thread erzeugt und gestartet.

Dem *MoonMover*-Thread werden bei der Erzeugung die Masse von Planet und Mond, den Abstand zwischen Mond und Erde, die Anfangsgeschwindigkeit des Mondes und die *TransformGroup* in der sich der Mond befindet übergeben.

Die Bewegungen des Mondes werden durch Veränderungen des auf die Mond-*TransformGroup* angewandten *Transform3D*s durchgeführt. Um diesen Vorgang zu vereinfachen wendet `moveMoonTo(float[] coords)` eine solche Transformation auf die Mond-*TransformGroup* an, das sich der Mond an der durch `coords` angegebenen Position befindet.

Nach der Erzeugung des *MoonMover*-Threads wird der Mond zunächst auf den Richtigen Abstand zur Erde bewegt.

Sobald der Thread gestartet wurde, wird dann auf Basis von physikalischer Gleichungen Schritt für Schritt im Abstand von 1 ms die nächste Position des Mondes berechnet und der Mond dorthin bewegt. Dies geschieht, solange die Variable `stop` **false** ist.

Wenn der Benutzer auf den Stopp-Knopf in der Benutzeroberfläche klickt, wird die Berechnung und Bewegung per Aufruf der Methode `stopThread`, welche `stop` auf **true** setzt, angehalten.

Wie dies alles im Detail realisiert wurde, lässt sich im Programmquelltext auf der beigefügten CD in Erfahrung bringen.

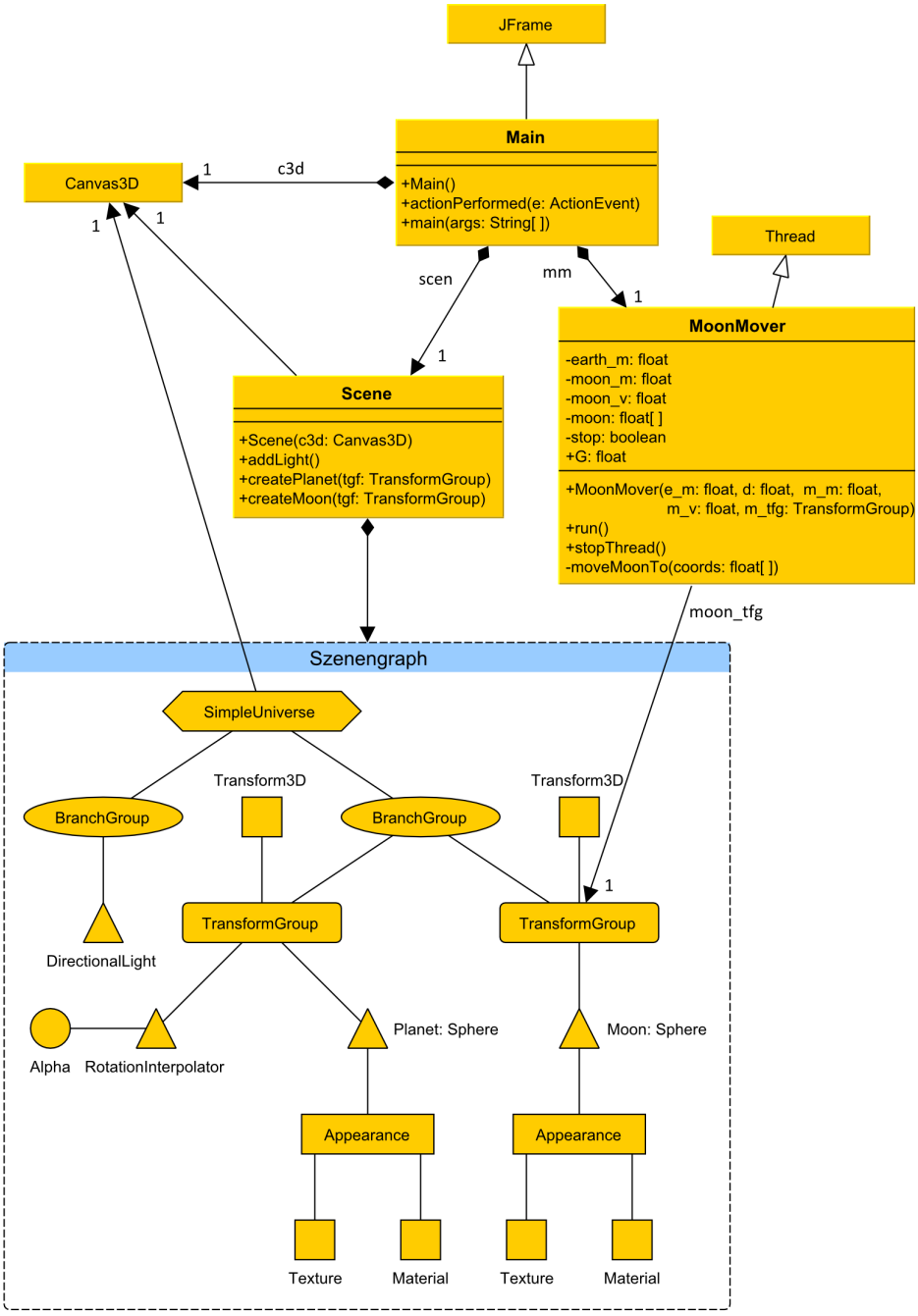

Abbildung 4: Aufbau der Beispielanwendung

Zusammenfassung

Allein mit den in dieser Arbeit vorgestellten Elementen von Java 3D, lassen sich bereits komplexe Szenen realisieren. Dank der noch viel umfangreicheren Java 3D API lassen sich folglich auch mit Java hochwertige Programme mit dreidimensionaler Darstellung erstellen.

Literaturverzeichnis

[1] **Wikipedia – Die freie Enzyklopädie – Artikel DirectX**
Stand 06.03.2008 13:00
http://de.wikipedia.org/wiki/DirectX

[2] **Andrew Davidson, Pro Java™ 6 3D Game Development**
Online Version, Stand 06.03.2008 13:00
http://fivedots.coe.psu.ac.th/~ad/jg2/ch01/ch01.pdf

[3] **Java 3D™ 1.5.1 API Documentation**
Stand 06.03.2008 13:00
http://download.java.net/media/java3d/javadoc/1.5.1/

[4] **J3D.ORG – Tutorials – BranchGroup.compile()**
Stand 06.03.2008 13:00
http://java3d.j3d.org/tutorials/quick_fix/compile.html

[5] **Frank Klawonn, Grundkurs Computergrafik mit Java**
1. Auflage, Wiesbaden: Vieweg 2005, ISBN 3-528-05919-2

Bildquellen

Abb. 1 **Java-Maskottchen „Duke“ mit Helm**
Stand 15.02.2008 15:00
http://duke.dev.java.net/images/JDK6/index.html

Abb. 2 **Wikimedia Commons – Image:3D coordinate system.svg**
Stand 06.03.2008 16:30
http://commons.wikimedia.org/wiki/Image:3D_coordinate_system.svg

Erdtextur **Wikimedia Commons – Image:Whole world - land and oceans.jpg**
Stand 11.03.2008 18:30
http://commons.wikimedia.org/wiki/Image:Whole_world_-_land_and_oceans.jpg

Mondtextur **Wikimedia Commons – Image:Moonmap from clementine data.png**
Stand 11.03.2008 18:30
http://commons.wikimedia.org/wiki/Image:Moonmap_from_clementine_data.png

Sterne **HubbleSite – Plot of Planetoid Sedna's Apparent Motion through Space from 2003 to 2005**
Stand 11.03.2008 19:00
http://hubblesite.org/gallery/album/solar_system_collection/pr2004014b/

Technische Hilfsmittel

Die Abbildungen 1, 2 und 3 und auch die Texturen wurden mit Adobe Photoshop CS 2 bearbeitet.

Zur Erstellung der Abbildungen 2 und 4 wurde der Graphenzeichner yEd und das Vektorzeichenprogramm Inkscape verwendet.

Zur Erstellung des Anwendungsbeispiels wurde der Quelltexteditor SciTE benutzt.

Diese Facharbeit wurde mit OpenOffice.org Writer 2.3 erstellt.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegebenen Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Ort: Paderborn

Datum: 13.03.2008

Unterschrift: _____

(Daniel Dreibrodt)