Computergrafik - Übungen
W. Kurth, T. Mangoldt

Java 3D
Einstiegs-Tutorial
Teil 1

Java 3D:

- Application Programming Interface (API) für Java
- Erweiterung von Java um Klassenbibliotheken, die als Interface für ein 3D-Grafik- und Soundsystem dienen
- gehört zu den Java Media APIs (neben Java Shared Data Toolkit, Java Advanced Imaging, Java 2D, Java Media Framework etc.)

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

Java3D 1.2 API Specification (mit Einführung, Beschreibung der Klassen, Beschreibung der beim Download mitgelieferten Beispiele):
http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dguide/index.html

systematische Klassendokumentation:
http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dapi/index.html

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

Ein Java 3D - Programm besteht (teilweise) aus Objekten der Java 3D - Klassen, die ein virtuelles Universum beschreiben, welches dann gerendert wird:
Kern-Klassen (Java 3D core classes) in **`javax.media.j3d`**
Hilfs-Klassen (utility classes) in **`com.sun.j3d.utils`**

außerdem werden benutzt:
`java.awt` (Abstract Windowing Toolkit) für die Ausgabe der gerenderten Ergebnisse
`javax.vecmath` für Vektor- und Matrizenrechnung

typische Import-Statements für eine Java 3D - Anwendung:

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
```

Java 3D benutzt das *Szenengraph-Konzept*
– aber erweitert gegenüber VRML

2 Arten von Beziehungen zwischen Java 3D-Klassen-Instanzen:

- Eltern-Kind-Beziehung (nicht verwechseln mit Vererbung!) – entspricht **`children`**-Relation in VRML
- Referenzierung: entspricht der Verwendung von Knoten in Feldern anderer Knoten in VRML, z.B. Felder **`material`**, **`proxy`**

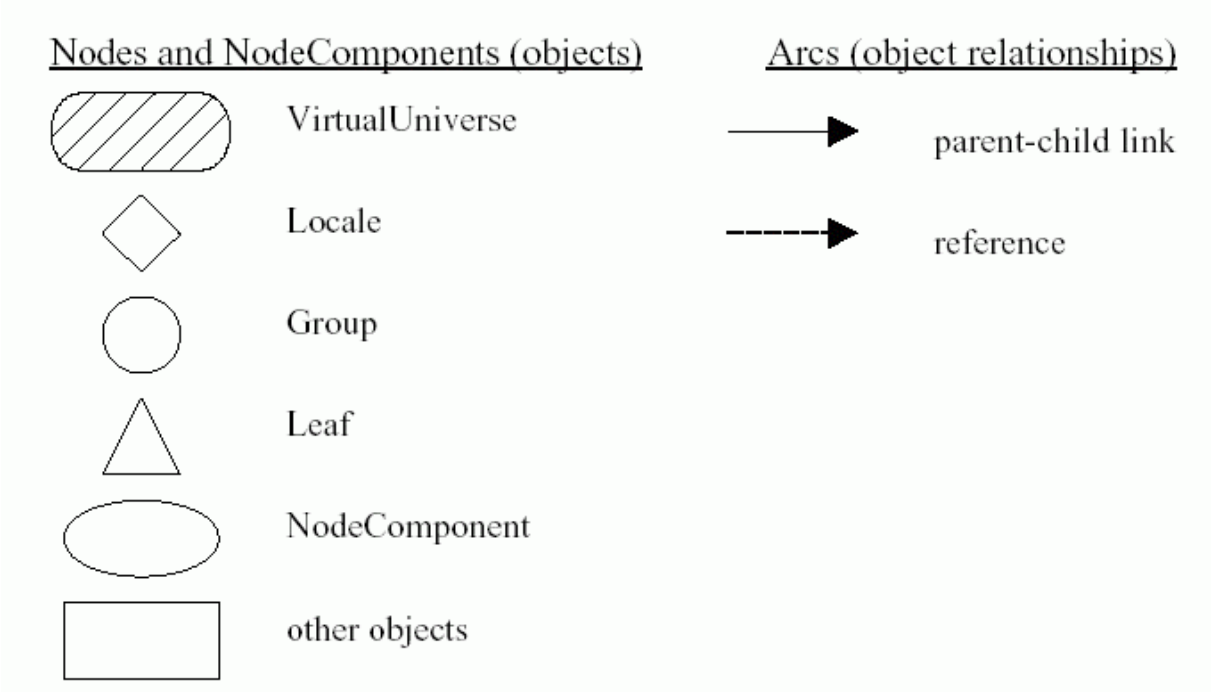
$\Rightarrow$ Szenengraph = gerichteter azyklischer Graph (DAG), bei Weglassen der Referenzierungskanten sogar ein Baum

Wurzel des Szenengraphen: **`Locale`**-Objekt (spezifiziert Referenzpunkt im virtuellen Universum)

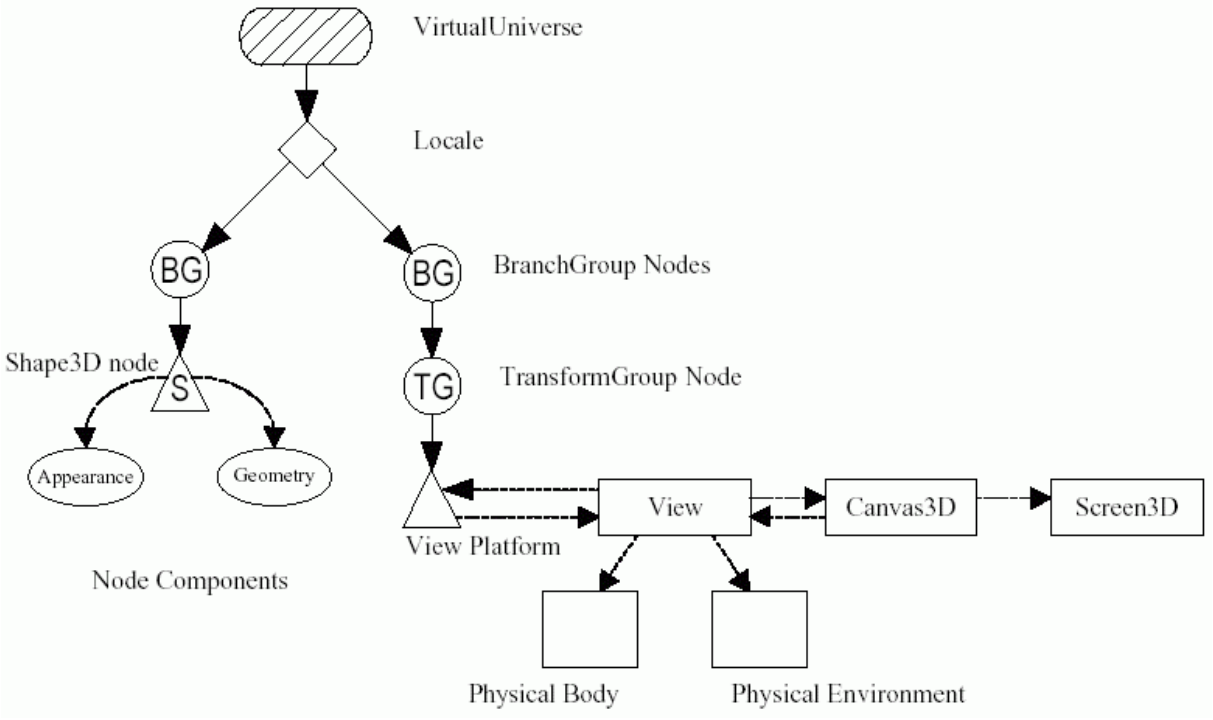
jeder Blattknoten des Szenengraphen hat eindeutigen Pfad von der Wurzel (ohne Referenzierungskanten), dieser spezifiziert vollständig die *Zustandsinformation* des Blattknotens: Position, Orientierung, Größe.

eine Reihe von Methoden wird bereitgestellt, um Szenengraphen zu manipulieren.

Konventionen für Szenengraph-Skizzen in den folgenden Beispielen:



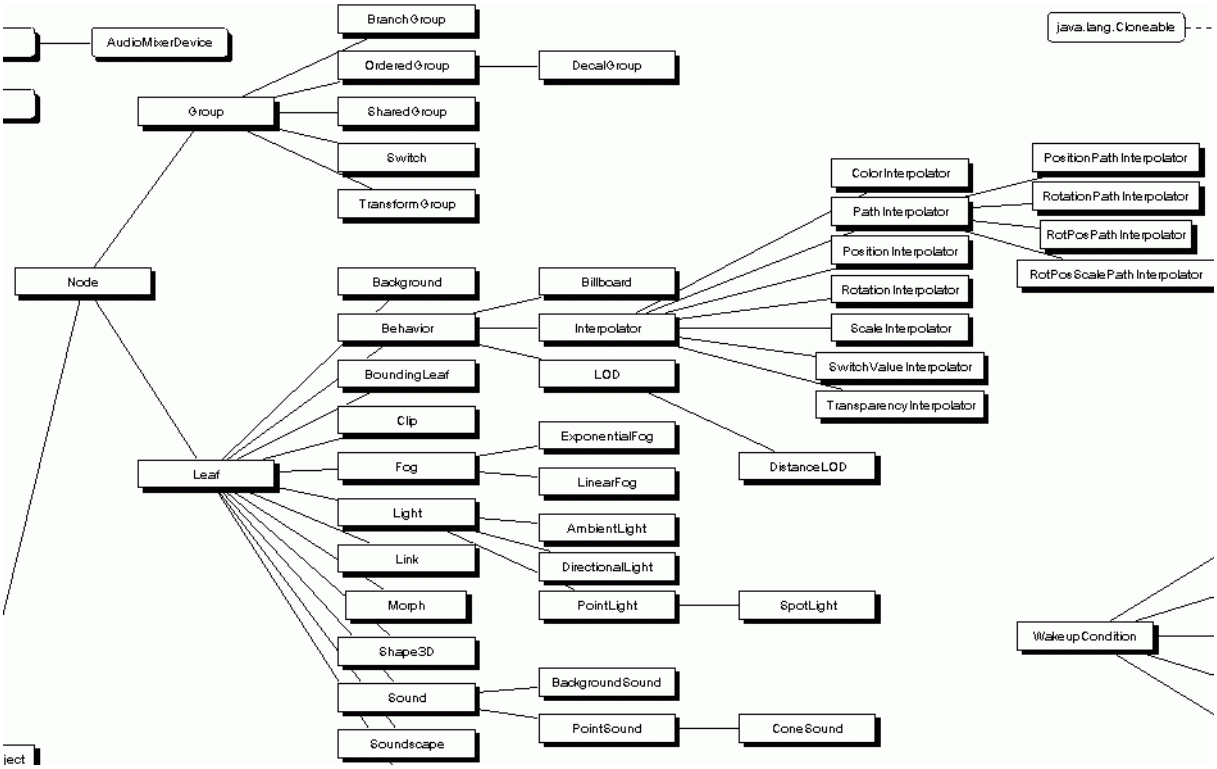
Beispiel eines Szenengraphen in Java 3D:



Die meisten Knoten eines Szenengraphen (außer der Wurzel: **`VirtualUniverse`**, und **`Locale`**) sind Instanzen (von Subklassen) der **`SceneGraphObject`**-Klasse.

Ausschnitt aus der Klassenhierarchie von Java 3D:

`SceneGraphObject` – **`Node`** – **`Group`** – ....
– **`Leaf`** – ....



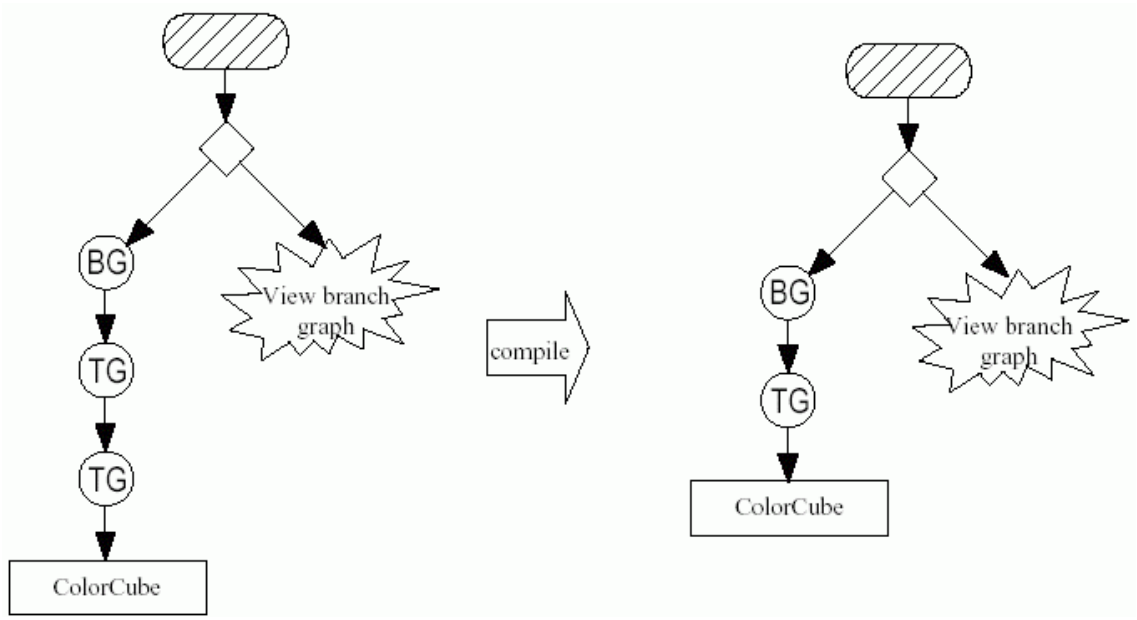
man beachte die Ähnlichkeiten, aber auch Erweiterungen gegenüber VRML!

spezielle **`Group`**-Knoten (Subklassen von **`Group`**):

- **`BranchGroup`** (entsprechend **`Group`**-Knoten in VRML)
- **`TransformGroup`** (entspr. **`Transform`**-Knoten in VRML)

Die **`BranchGroup`**-Klasse besitzt eine **`compile`**-Methode, die vor dem Rendering des Szenengraphen angewendet werden kann. Aufruf dieser Methode überführt den gesamten darunterliegenden Teilbaum des Szenengraphen in eine interne Repräsentation für Java 3D, die ggf. optimiert ist.

Beispiel für effizientere Repräsentation in der internen Darstellung: Zusammenfassung von adjazenten **`TransformGroup`**- (TG-) Knoten

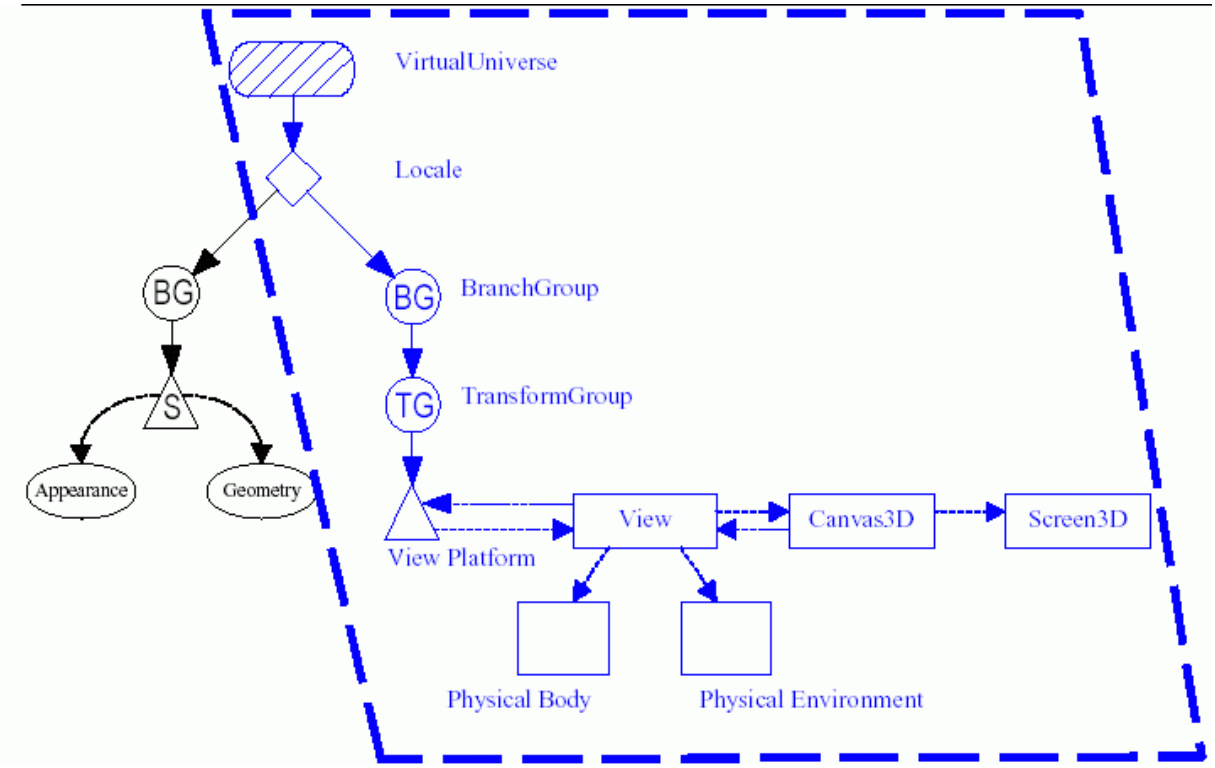


Grundschema zur Erstellung von Java 3D - Programmen:

1. Erzeugen eines **`Canvas3D`**-Objekts
   (die **`Canvas3D`**-Klasse ist abgeleitet von der **`Canvas`**-Klasse des Abstract Windowing Toolkit (AWT))
2. Erzeugen eines **`VirtualUniverse`**-Objekts
3. Erzeugen eines **`Locale`**-Objekts, dies wird verknüpft mit dem **`VirtualUniverse`**-Objekt
4. Erzeugen eines *view branch*-Graphen ("rechter Flügel" des Szenengraphen):
   a. Erzeugen eines **`View`**-Objekts
   b. Erzeugen eines **`ViewPlatform`**-Objekts
   c. Erzeugen eines **`PhysicalBody`**-Objekts
   d. Erzeugen eines **`PhysicalEnvironment`**-Objekts
   e. Verknüpfung von **`ViewPlatform`**, **`PhysicalBody`**, **`PhysicalEnvironment`** und **`Canvas3D`**-Objekt mit dem **`View`**-Objekt
5. Erzeugen eines oder mehrerer *content branch*-Graphen ("linker Flügel" des Szenengraphen)
6. Kompilieren dieser beiden *branch*-Graphen
7. Einfügen beider Teilgraphen in das **`Locale`**-Objekt

Da wir den *view branch*-Graphen nur in standardisierter, einfacher Form benutzen, können wir eine Utility für vereinfachte Generierung von Szenengraphen benutzen, die **`SimpleUniverse`**-Klasse (in **`com.sun.j3d.utils.universe`**).

Durch Erzeugen eines **`SimpleUniverse`**-Objekts wird ein "minimales virtuelles Universum" bereitgestellt, das den Inhalt des gestrichelt berandeten Bereichs in Standardform bereitstellt:



Damit vereinfacht sich das obige allgemeine Schema für die Programmerstellung zu:

1. Erzeugung eines **`Canvas3D`**-Objekts
2. Erzeugung eines **`SimpleUniverse`**-Objekts, das auf das vorher erzeugte **`Canvas3D`**-Objekt Bezug nimmt

2a. Anpassen des **`SimpleUniverse`**-Objekts

3. Konstruktion des *content branch*
4. Compilieren des *content branch*
5. Einfügen des *content branch* in das **`Locale`**-Objekt des **`SimpleUniverse`**-Objekts.

Erstes Beispielprogramm:

Darstellung eines Würfels mit farbigen Seitenflächen, der um 2 Achsen gedreht wurde, damit 3 seiner Seitenflächen sichtbar sind

(es wird ein **ColorCube**-Objekt verwendet, das schon als Utility bereitsteht)

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.ColorCube;
import javax.media.j3d.*;
import javax.vecmath.*;

public class FarbWuerfel1 extends Applet
   {
   public FarbWuerfel1()
      {
      setLayout(new BorderLayout());
      GraphicsConfiguration config =
        SimpleUniverse.getPreferredConfiguration();
      Canvas3D canvas3D = new Canvas3D(config);
      add("Center", canvas3D);
      BranchGroup szene = macheSzenengraph();
      szene.compile();
      // Verwendung der Utility-Klasse
      // SimpleUniverse:
      SimpleUniverse universum =
           new SimpleUniverse(canvas3D);
      universum.getViewingPlatform(
           ).setNominalViewingTransform();
      universum.addBranchGraph(szene);
      } // end Konstruktor

   public BranchGroup macheSzenengraph()
      {
      BranchGroup objWurzel = new BranchGroup();

      // Transformation,
      // Komposition von 2 Rotationen:
```

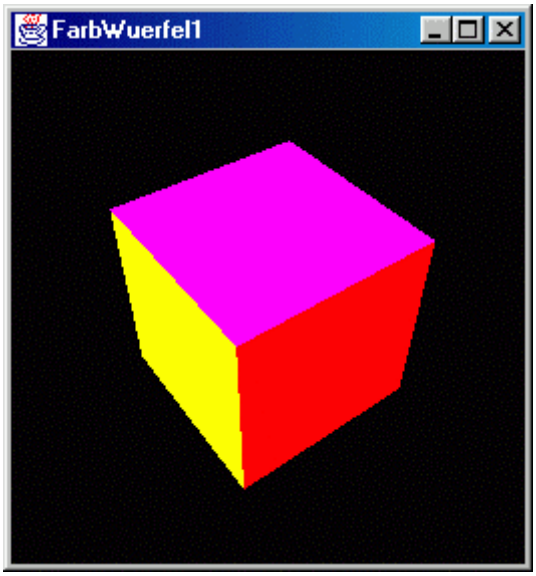
```
    Transform3D drehung = new Transform3D();
    Transform3D drehung2 = new Transform3D();
    drehung.rotX(Math.PI / 4.0d);
    drehung2.rotY(Math.PI / 5.0d);
    drehung.mul(drehung2);
    TransformGroup objDreh =
         new TransformGroup(drehung);

    objDreh.addChild(new ColorCube(0.4));
    objWurzel.addChild(objDreh);
    return objWurzel;
    } // end macheSzenengraph-Methode

 public static void main(String[] args)
    {
    Frame frame = new MainFrame(new FarbWuerfel1(),
                  256, 256);
    }
 } // end class FarbWuerfel1
```

man beachte:

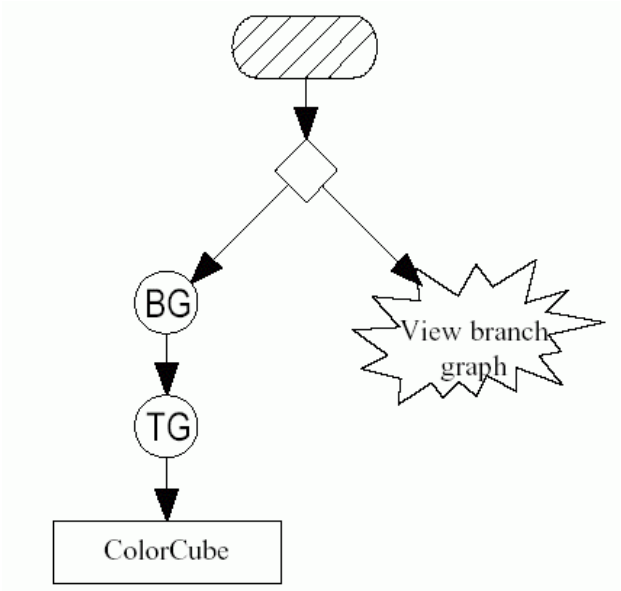
- als einziges zusätzliches **`import`** neben den oben schon angegebenen wird hier **`com.sun.j3d.utils.geometry.ColorCube`** gebraucht
- die Funktion **`main`** wird gebraucht, um das Programm als Anwendung laufen zu lassen (die Klasse **`Applet`** stellt die Fensterverwaltung bereit)
- im Konstruktor von **`FarbWuerfel1`** wird das oben angegebene vereinfachte Schema der Szenenerstellung abgearbeitet – dieser Teil kann für die meisten weiteren Beispiele übernommen werden
- die Konstruktion des *content branch* ist in die Methode **`macheSzenengraph`** ausgelagert
- es gibt **`Transform3D`**-Objekte, die keine Knoten sind, sondern nur zu deren Spezifikation (nämlich von **`TransformGroup`**-Knoten) benötigt werden
- diese entsprechen geometrischen (affinen) Transformationen und verfügen über eine Methode **`mul`** zur Komposition (hier für die Nacheinanderanwendung der beiden Rotationen)

- **`addChild`** ist die Methode von **`Group`**-Knoten, um Kindknoten anzufügen
- am Schluss wird die Wurzel des zusammenhängenden *content branch* an den Konstruktor zurückgegeben und wird dort mit **`addBranchGraph`** eingebaut.

Das Ergebnis sollte etwa so aussehen (Fensterlayout möglicherweise betriebssystemabhängig):



dem Beispiel zugrundeliegender Szenengraph:



Capabilities

Im Vergleich zu VRML ist Java 3D besser abgesichert:
Visuelle Objekte können zur Laufzeit nur verändert werden (z.B. in Animationen), wenn vorher entsprechende "Capabilities" gesetzt wurden.
Jedes **`SceneGraphObject`** hat eine Liste von *capability bits*.

Direkter Zugriff auf diese Tabelle (wird seltener gebraucht):

Methoden von **`SceneGraphObject:`**
`void clearCapability(int bit)`
löscht das spezifizierte *capability bit*
`boolean getCapability(int bit)`
stellt das spezifizierte bit zur Verfügung
`void setCapability(int bit)`
setzt das spezifizierte bit

intuitiverer Zugriff:
<Instanz-Name>.**`setCapability`**
(<Klassenname>.<capability-Name>)

dabei ist <capability-Name> einer der für die jeweilige Klasse vordefinierten capability-Namen (in Großbuchstaben).

Capabilities von **`Group`**:

`ALLOW_CHILDREN_EXTEND`
erlaubt das Anfügen neuer Kindknoten
`ALLOW_CHILDREN_READ`
`ALLOW_CHILDREN_WRITE`
erlaubt die Änderung der Referenzen auf die Kind-knoten (Überschreiben)

Capabilities von **`TransformGroup`**: zusätzlich

`ALLOW_TRANSFORM_READ`
`ALLOW_TRANSFORM_WRITE`

Beispiel:

```
TransformGroup drehobjekt =
        new TransformGroup();
drehobjekt.setCapability
        (TransformGroup.ALLOW_TRANSFORM_WRITE);
```

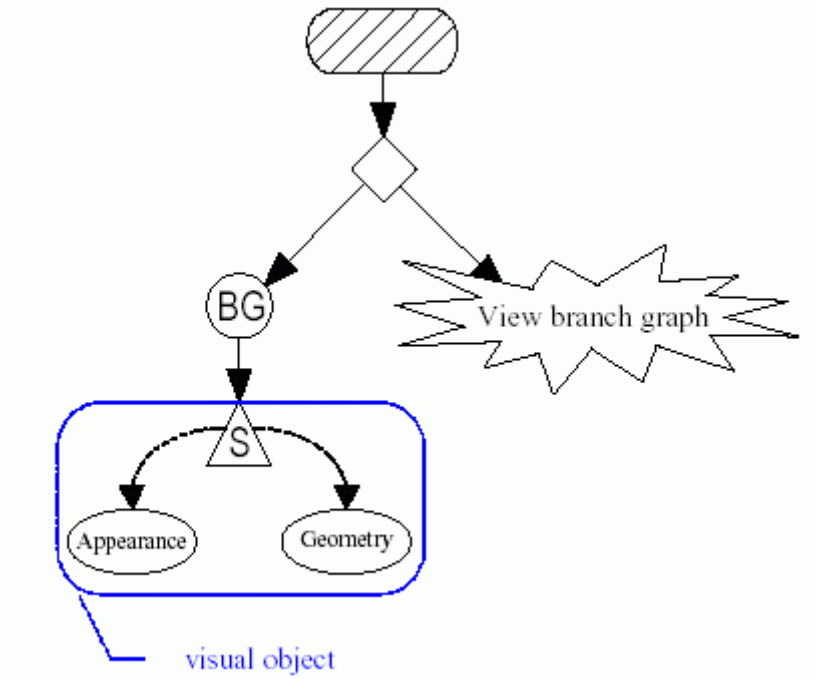
generiert neue **`TransformGroup`**-Instanz und gibt die Möglichkeit, in Animationen ihre Parameter zu überschreiben.

`Shape3D`-*Objekte* entsprechen den **`Shape`**-Knoten in VRML. Ihre Capabilities:

`ALLOW_GEOMETRY_READ`
`ALLOW_GEOMETRY_WRITE`
`ALLOW_APPEARANCE_READ`
`ALLOW_APPEARANCE_WRITE`
`ALLOW_COLLISION_BOUNDS_READ`
`ALLOW_COLLISION_BOUNDS_WRITE`

Definition sichtbarer Objekte mit **`Shape3D`**

Wie in VRML werden Geometrie und Erscheinungsbild durch **`Geometry`**- und **`Appearance`**-Knoten, auf die referenziert wird, näher spezifiziert.



Mögliche Aufrufe des **`Shape3D`**-Konstruktors:

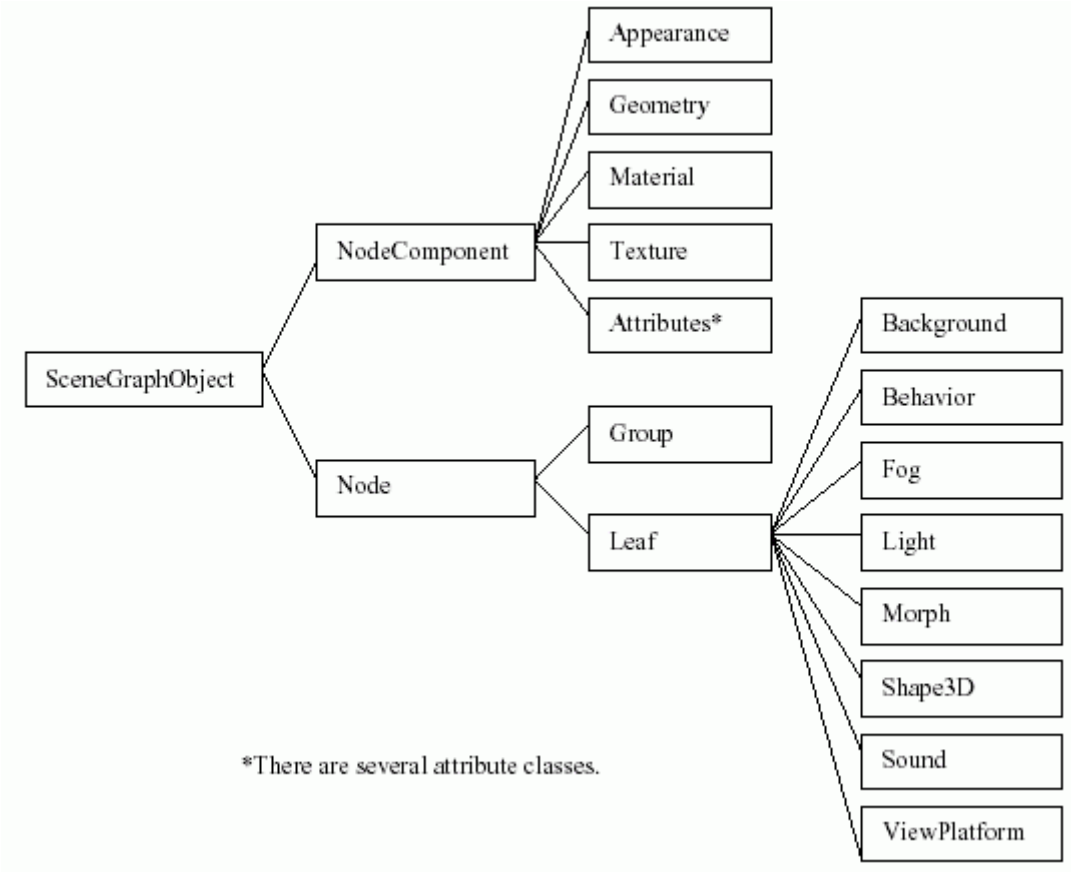
```
Shape3D()
Shape3D(Geometry geometrieknoten)
Shape3D(Geometry geometrieknoten,
            Appearance appknoten)
```

Methoden zum Hinzufügen von entsprechenden Knoten, wenn der **`Shape3D`**-Knoten schon vorhanden ist:

```
void setGeometry(Geometry geometrieknoten)
void setAppearance(Appearance appknoten)
```

sowie entsprechende **`get`**-Methoden (analog).

`Geometry` und **`Appearance`** sind Subklassen von **`NodeComponent`**:



Ausschnitt aus der Klassenhierarchie von Java 3D

Eigene, spezielle visuelle Objekte lassen sich als Subklasse von **`Shape3D`** selbst definieren und in Java 3D-Programmen benutzen, etwa nach folgendem Schema:

```
public class SichtbaresObj extends Shape3D
   {
   private Geometry sio_geom;
   private Appearance sio_app;
   public SichtbaresObj()
      {
      sio_geom = erzeugeGeometrie();
      sio_app = erzeugeApp();
      this.setGeometry(sio_geom);
      this.setAppearance(sio_app);
      }
   private Geometry erzeugeGeometrie()
      { /* Code, um Default-Geometrie
        zu erzeugen */ }
   private Appearance erzeugeApp()
      { /* Code, um Default-Erscheinungsbild
        zu erzeugen */ }
   } // Ende von SichtbaresObj-Klasse
```

Die so def. **`SichtbaresObj`**-Klasse kann genauso einfach benutzt werden wie **`ColorCube`** im ersten Programmierbeispiel (s.o.).
Wenn z.B. **`objWurzel`** eine Instanz von **`Group`** ist, erzeugt der folgende Code ein **`SichtbaresObj`** und fügt es als Kindknoten im Szenengraph an **`objWurzel`** an:

```
objWurzel.addChild(new SichtbaresObj() );
```

Bei Verwendung "vordefinierter" Möglichkeiten gibt es drei Varianten der Geometrie-Spezifikation in Java 3D:

- Verwendung von Geometrie-Primitiven
- Verwendung von *boundary-representation*-Objekten (analog zu **`IndexedLineSet`** etc. in VRML)
- Verwendung eines *geometry loader* zum Laden von Objekten aus anderer Software / anderen Spezifikationssprachen

Verfügbare Loader:

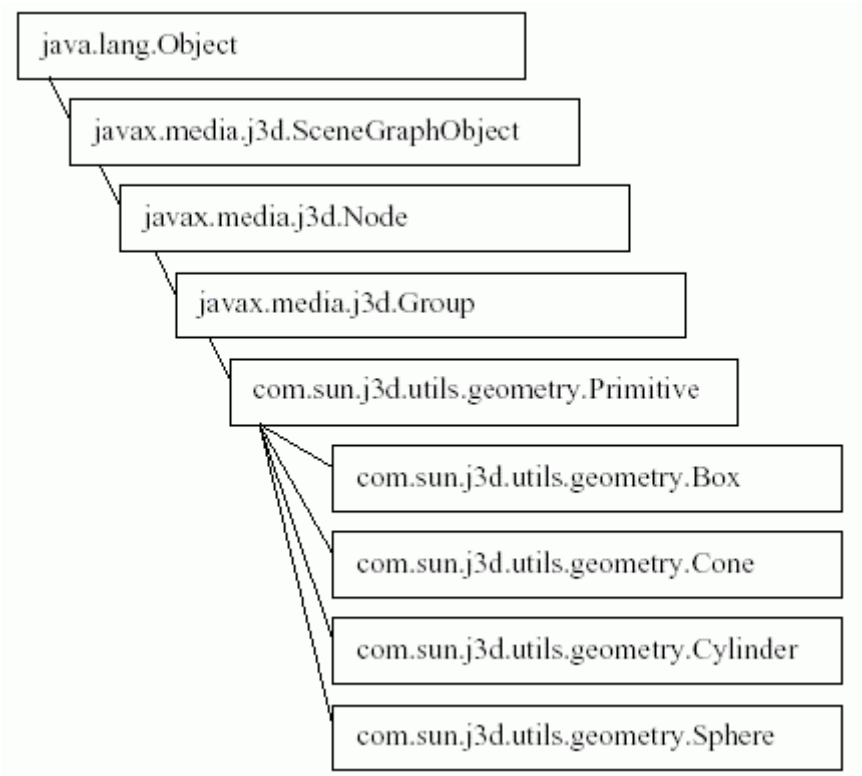
File Format	Description
3DS	3D-Studio
COB	Caligari trueSpace
DEM	Digital Elevation Map
DXF	AutoCAD Drawing Interchange File
IOB	Imagine
LWS	Lightwave Scene Format
NFF	WorldToolKit NFF format
OBJ	Wavefront
PDB	Protein Data Bank
PLAY	PLAY
SLD	Solid Works (prt and asm files)
VRT	Superscape VRT
VTK	Visual Toolkit
WRL	Virtual Reality Modeling Language

Verwendung geometrischer Primitive:

Wie in VRML gibt es **`Box`**, **`Cone`**, **`Cylinder`** und **`Sphere`**.
Die Parameter unterscheiden sich etwas.
`Box`, **`Cone`** und **`Cylinder`** setzen sich (intern) aus mehr als einem Shape3D-Objekt zusammen.

Package: **`com.sun.j3d.utils.geometry`**

Einordnung in die Klassenhierarchie:



`Box`:
Default-Seitenlänge 2, Positionierung mit dem Mittelpunkt im Zentrum des Koordinatensystems.
Konstruktor-Aufrufe:

```
Box()
Box(float xdim, float ydim, float zdim,
Appearance app)
```

`Cone`:
Default-Radius 1, Höhe 2, Zentrum im Mittelpunkt der bounding box, Zentralachse = *y*-Achse.
Konstruktor-Aufrufe:

```
Cone()
Cone(float radius, float hoehe)
```

`Cylinder`:
Defaults wie für **`Cone`**. Konstruktor-Aufrufe:

```
Cylinder()
Cylinder(float radius, float height, Appearance
app)
```

`Sphere:`
Default-Radius 1, Mittelpunkt im Zentrum des Koordinatensystems.
Konstruktor-Aufrufe:
`Sphere()`
`Sphere(float radius)`
`Sphere(float radius, Appearance app)`

Verknüpfung mit einem **`Appearance`**-Knoten bei schon instanziiertem Objekt (für **`Box`**, **`Cone`**, **`Cylinder`** und **`Sphere`**):

Methode
`void setAppearance(Appearance app)`

Zugriff auf die Einzelkomponenten (Seitenflächen) dieser "Primitiv-Objekte":

Methode
`Shape3D getShape(int id)`

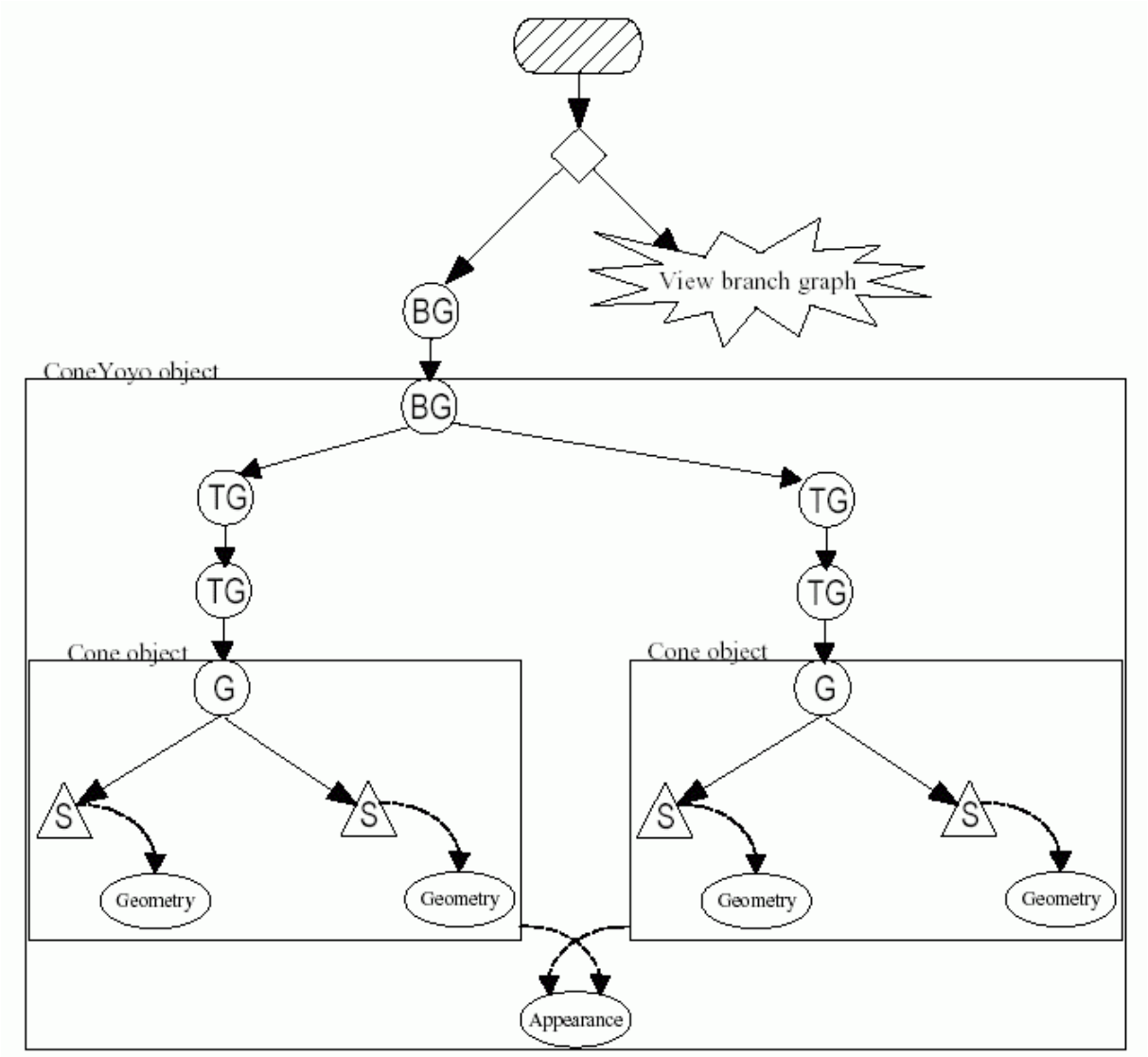
`id` ist ein Index, der festlegt, welche Einzelkomponente ausgewählt wird.

Verknüpfung von **`Appearance`** mit nur einer Einzel-komponente:

`void setAppearance(int id, Appearance app)`

Beispiel-Programm:
Erzeugung einer Jojo-Figur aus 2 Kegeln

Zugehöriger Szenengraph:



Die Strukturen innerhalb der inneren Rechtecke (Cone object) werden im Programm nicht explizit konstruiert.
Die Struktur innerhalb des äußeren Rechtecks wird im Beispielprogramm durch "**`kegelJojo`**" referenziert (man könnte hierfür eine neue Klasse definieren, das ist im Beispielprogramm nicht geschehen):

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.Cone;
import javax.media.j3d.*;
import javax.vecmath.*;

public class Jojo1 extends Applet
   {
   public Jojo1()
      {
      setLayout(new BorderLayout());
      GraphicsConfiguration config =
        SimpleUniverse.getPreferredConfiguration();
      Canvas3D canvas3D = new Canvas3D(config);
      add("Center", canvas3D);
      BranchGroup szene = macheSzenengraph();
      szene.compile();
      SimpleUniverse univ = new
         SimpleUniverse(canvas3D);
      univ.getViewingPlatform(
               ).setNominalViewingTransform();
      univ.addBranchGraph(szene);
      } // end Konstruktor

   public BranchGroup macheSzenengraph()
      {
      BranchGroup objWurzel = new BranchGroup();
      BranchGroup jojo = kegelJojo();
      objWurzel.addChild(jojo);
      return objWurzel;
      } // end macheSzenengraph-Methode

   public BranchGroup kegelJojo()
      {
      BranchGroup jojoBG = new BranchGroup();
      Transform3D dreh = new Transform3D();
      Transform3D verschieb = new Transform3D();
      Appearance jojoApp = new Appearance();

      dreh.rotZ(Math.PI/2.0d);
```

```
    TransformGroup jojoD1 = new
       TransformGroup(dreh);

    verschieb.set(new Vector3f(0.1f, 0f, 0f));
    TransformGroup jojoV1 = new
       TransformGroup(verschieb);

    Cone kegel1 = new Cone(0.6f, 0.2f);
    kegel1.setAppearance(jojoApp);

    jojoBG.addChild(jojoV1);
    jojoV1.addChild(jojoD1);
    jojoD1.addChild(kegel1);

    dreh.rotZ(-Math.PI/2.0d);
    TransformGroup jojoD2 = new
       TransformGroup(dreh);

    verschieb.set(new Vector3f(-0.1f, 0f, 0f));
    TransformGroup jojoV2 = new
       TransformGroup(verschieb);

    Cone kegel2 = new Cone(0.6f, 0.2f);
    kegel2.setAppearance(jojoApp);

    jojoBG.addChild(jojoV2);
    jojoV2.addChild(jojoD2);
    jojoD2.addChild(kegel2);

    jojoBG.compile();
    return jojoBG;
    } // end kegelJojo

public static void main(String[] args)
    {
    Frame frame = new
       MainFrame(new Jojo1(), 500, 400);
    // x=500, y=400: Fenster-Ausmasse (Pixel)
    }
} // end class Jojo1
```

Ergebnis: