

# B6. 3D-Computergrafik mit Java

B6.1 Grundlagen der 3D-Computergrafik   
(teilweise Wiederholung zu Vorlesung Digitale Medien)

B6.2 Einführung in Java 3D

B6.3 Animation

B6.4 Geometrie, Material, Beleuchtung

Literatur:

Alan Watt: 3D Computergrafik, 3. Auflage, Pearson Studium 2002

# Dreidimensionale Darstellung

- Dimensionenkonflikt:
  - Die reale Welt ist dreidimensional
  - Bilddarstellungen (wie bisher betrachtet) sind zweidimensional
    - » Verdeckte Ansichten und Details
- Dreidimensionale Darstellung:
  - Setzt Modell mit den Informationen in allen drei Dimensionen voraus
    - » Alle möglichen Ansichten ohne Informationsverlust
- Anwendungsbereiche für dreidimensionale Darstellung:
  - Virtuelle Welten, „Cyberspace“
  - Ingenieur Anwendungen:
    - » CAD (z.B. Maschinenbau)
    - » Designmodelle von Produkten
    - » (Interaktive) Architekturmodelle
  - Produktpräsentation
  - Geovisualisierung
  - Animation im Film (Trickfiguren in klassischem Film, Vollanimation)

# 3D-Eingabegeräte

- Zeigegeräte: Siehe früheres Kapitel der Vorlesung
- 3D-Scanner:
  - Abtastung realer Objekte z.B. mit Ultraschall, Laser
- Regelmässige Abtastung von 2D-Daten:
  - Z.B. Bildgebende Verfahren der Medizin
    - » Computertomographie (Röntgenbilder)
    - » Magnetresonanztomographie (Kernspin-Prinzip)
  - Errechnung eines „Volumendatensatzes“
    - » Nicht nur Aussenansicht, sondern Information über alle möglichen Schnitte
    - » Filterung anhand verschiedener Kriterien, z.B. Farb-Schwellwerte



**HDS3000**

*Leica*  
Geosystems



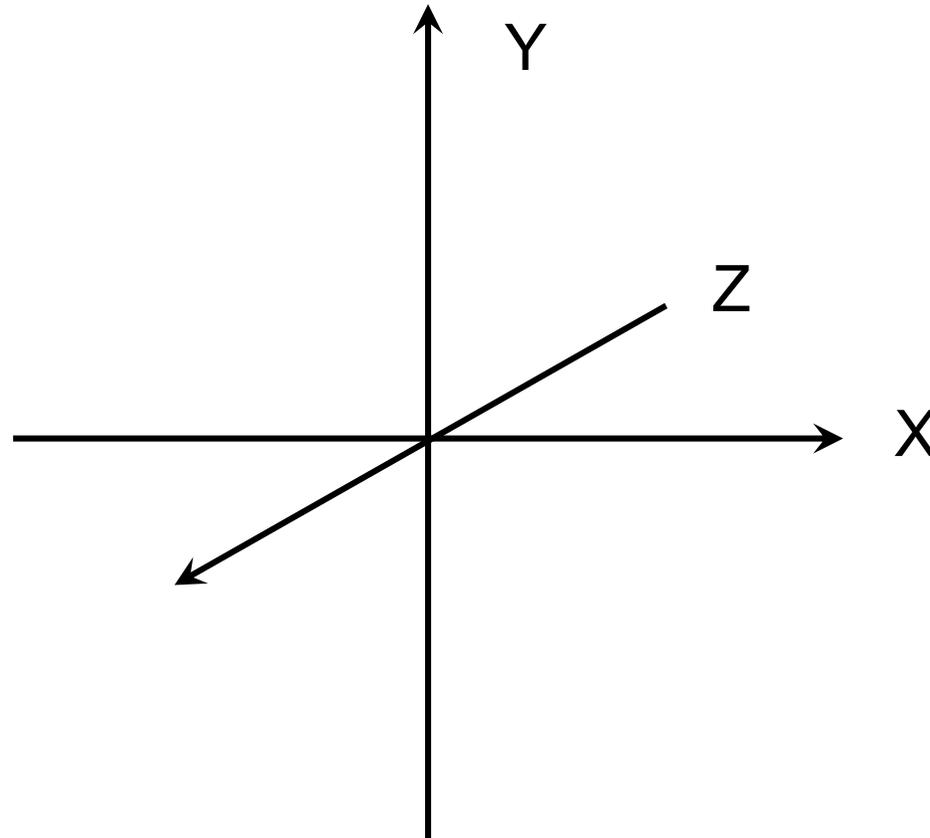
Indeed Amira

# 3D-Ausgabegeräte

- Echte 3D-Darstellung:
  - 3D-Drucker (erzeugt Gegenstände)
- Stereo-Vision:
  - Separate Bildinformation für die beiden Augen
  - Polarisations- und Farbbrillen
  - Shutterbrillen (abwechselnd ein Auge abgedunkelt)
- Immersive Verfahren:
  - *Head Mounted Display (HMD)*
  - Spezialräume mit Rundum-Projektion (*Cave*)
  - Anpassung der Darstellung an Position und Bewegung des Betrachters durch *tracking*
- Einfachste Darstellungsform:
  - 2D-Rendering in „Fenster zur Welt“
  - 2D-Darstellung wird flexibel aus 3D-Daten berechnet

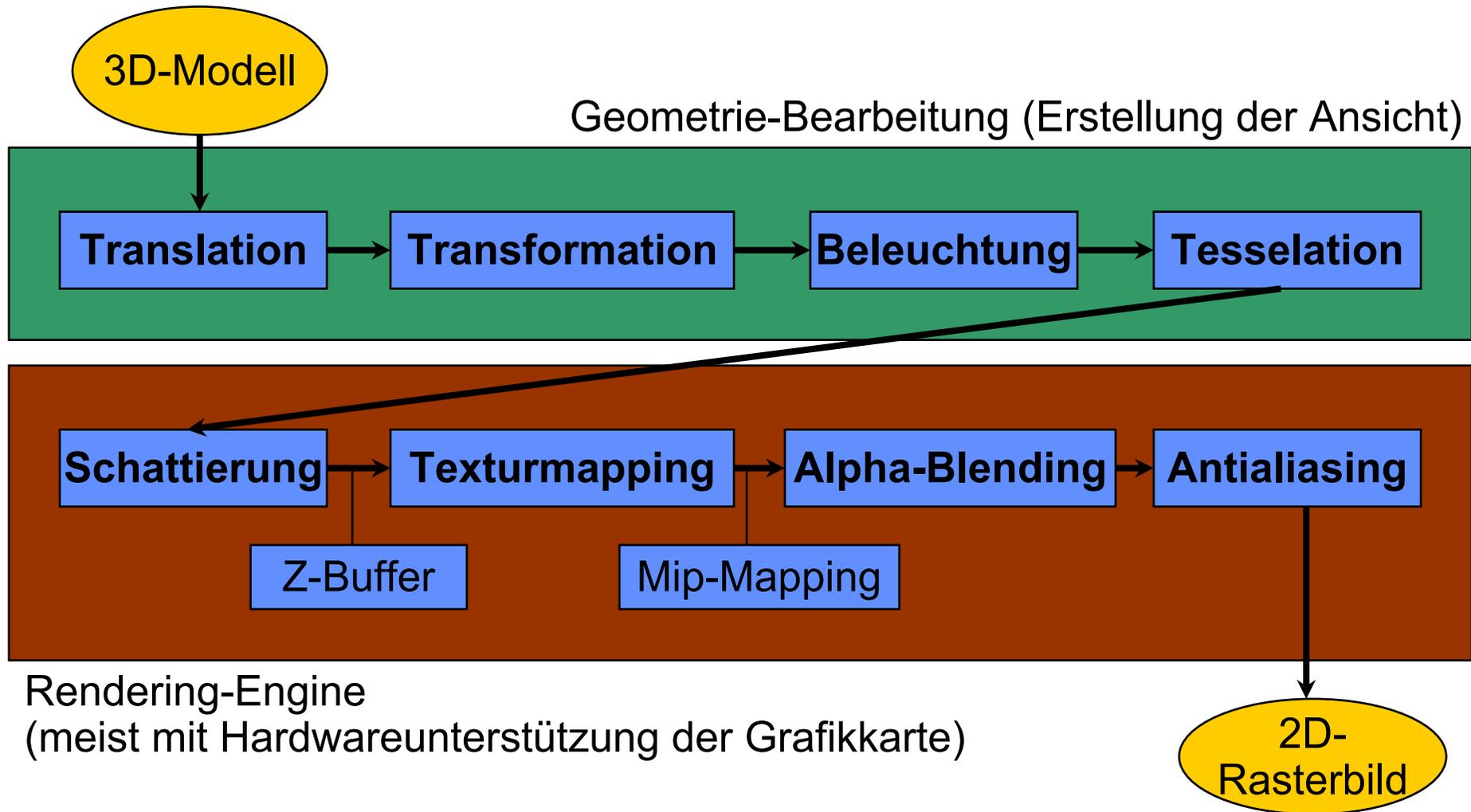


# 3D-Koordinatensystem



Kartesisches Koordinatensystem  
Merkhilfe: „Rechte-Hand-Regel“

# 3D-Rendering-Pipeline



# Transformation

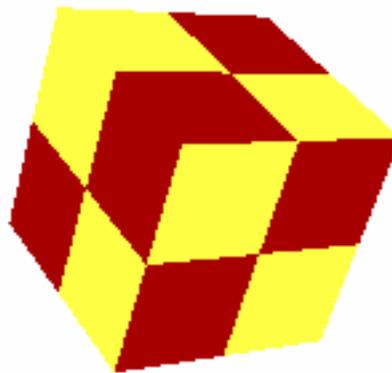
- Änderung der Objektposition
  - Verschieben (oft auch *translation* genannt)
  - In 3 Freiheitsgraden
- Änderung der Objektausrichtung
  - Rotation
  - In 3 Freiheitsgraden
- Änderung der Objektgröße
  - Skalierung
  - 1 Freiheitsgrad bei Erhaltung der Proportionen
  - 3 Freiheitsgrade bei Verzerrung
- Bewegung der Betrachterposition in einer virtuellen Welt:
  - Obige Operationen treten (kombiniert) extrem häufig auf
  - Schnelle Implementierung wichtig

# Z-Buffer

- Z-Buffer speichert für jeden Bildpunkt des 2D-Bildes die niedrigste Entfernung zu einem Objekt
- Beschleunigung des Rendering:
  - Offensichtlich verdeckte Objekte bzw. Objektteile müssen nicht betrachtet werden
- Grössere Wahlfreiheit bei der Abarbeitung des Rendering
  - Hintergrundbild muss nicht unbedingt zeitlich vor den Vordergrundobjekten gerendert werden
- Hardwareunterstützung in Grafikkarten

# Textur-Mapping

- Textur:
  - Muster oder Bild, das auf Oberflächen von 3D-Objekten gelegt wird
  - Kann oft Anzahl der benötigten Polygone drastisch reduzieren
- Textur-Mapping:
  - Darstellung der Flächen eines Objekts mit Textur
  - Erweiterung: Perspektivische Korrektur der Textur
- Mip-Mapping:
  - Textur in mehreren Auflösungen verfügbar (und automatisch passende Fassung ausgewählt)



# B6. 3D-Computergrafik mit Java

B6.1 Grundlagen der 3D-Computergrafik

B6.2 Einführung in Java 3D



B6.3 Animation

B6.4 Geometrie, Material, Beleuchtung

## Literatur:

- D. Selman: Java 3D Programming, Manning 2002
- A. E. Walsh, D. Gehringer: Java 3D API Jump-Start, Prentice-Hall 2001
- <http://java.sun.com/products/java-media/3D/>
- <http://java.sun.com/developer/onlineTraining/java3d/>
- <http://java.sun.com/docs/books/java3d/>
- <http://www.j3d.org>, <http://www.java3d.org/>

# Java 3D

- Java 3D API
  - Ist eine Schnittstelle zum Erstellen von Programmen, die dreidimensionale Grafik darstellen können und Interaktion mit dreidimensionaler Grafik ermöglichen
  - Ist (wie Java 2D) auf höherer Abstraktionsebene angesiedelt als maschinennahe Grafik-Schnittstellen wie DirectX und OpenGL
- Wichtige Eigenschaften:
  - Verwendung von Nebenläufigkeit (Threads), wo möglich
  - Benutzung der Szenengraph-Abstraktion (analog zu VRML)
  - Erlaubt die Erzeugung von eigenständigen Programmen oder Applets zur Einbindung in Webseiten
- Pakete:
  - `javax.media.j3d`
  - `com.sun.j3d.utils` *Java 3D utility classes*

# Szenengraph

- Baumstruktur aus Knotenobjekten (*Node objects*)
  - Terminologie stark an VRML angelehnt
  - Die Darstellung eines einzelnen Knotens hängt von dem *Pfad* von der Wurzel des Szenengraphs bis zum Knoten ab
    - » Transformations-Knoten
    - » Zusatzknoten mit weiterer Information, z.B. über Texturen
  - Sharing von Knoten (d.h. Directed Acyclic Graph = DAG) nur unter sehr speziellen Umständen erlaubt
- Java-Szenengraph ist eine Datenstruktur
  - Kann dynamisch modifiziert werden
  - Teilweise sehr umfangreich
    - » Optimierungen

# Grafische Darstellung von Szenengraphen

## Nodes and NodeComponents (objects)



VirtualUniverse



Locale



Group



Leaf



NodeComponent



other objects

## Arcs (object relationships)



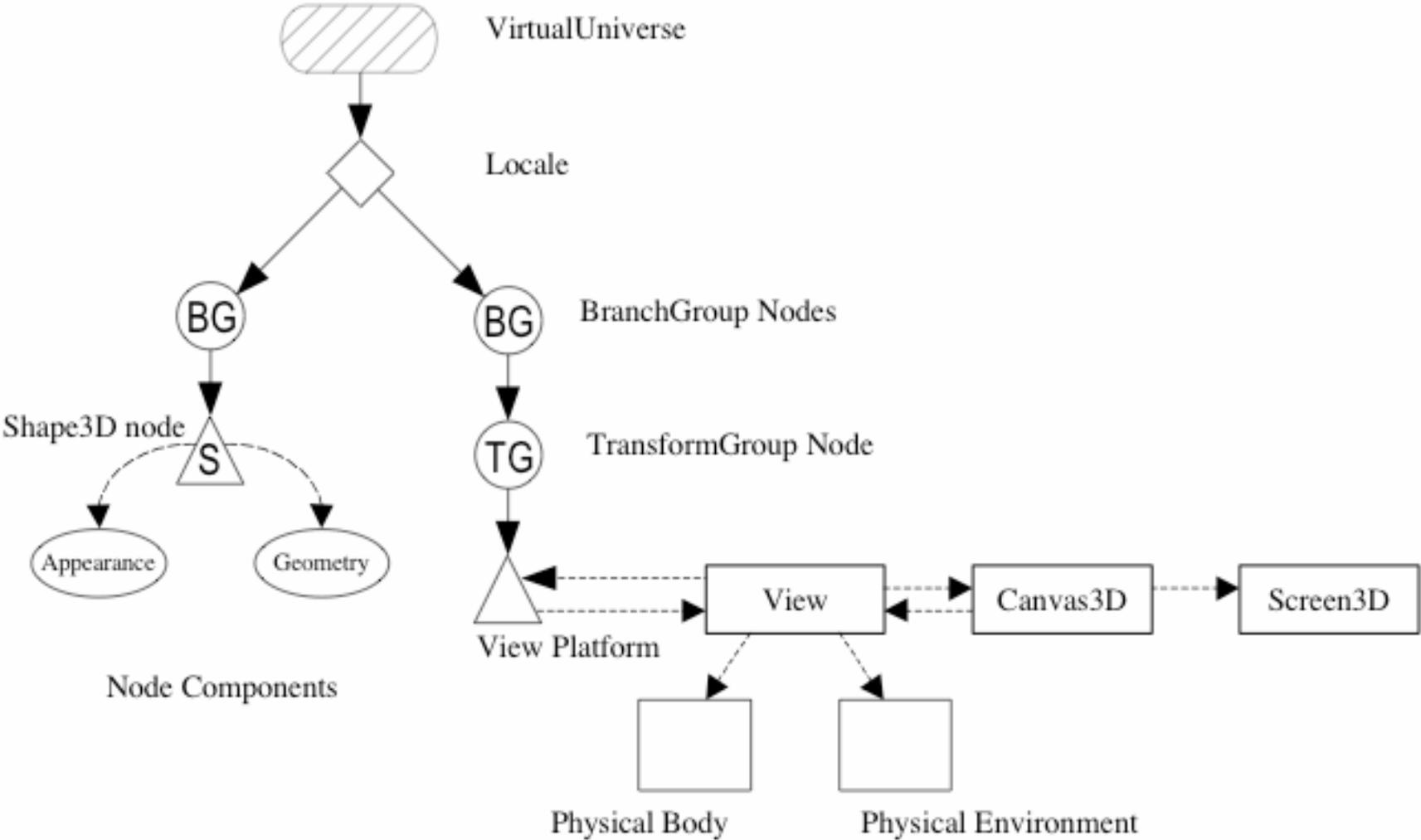
parent-child link



reference

- VirtualUniverse: Global root
- Locale: „A landmark to determine the location of visual objects in the virtual universe“

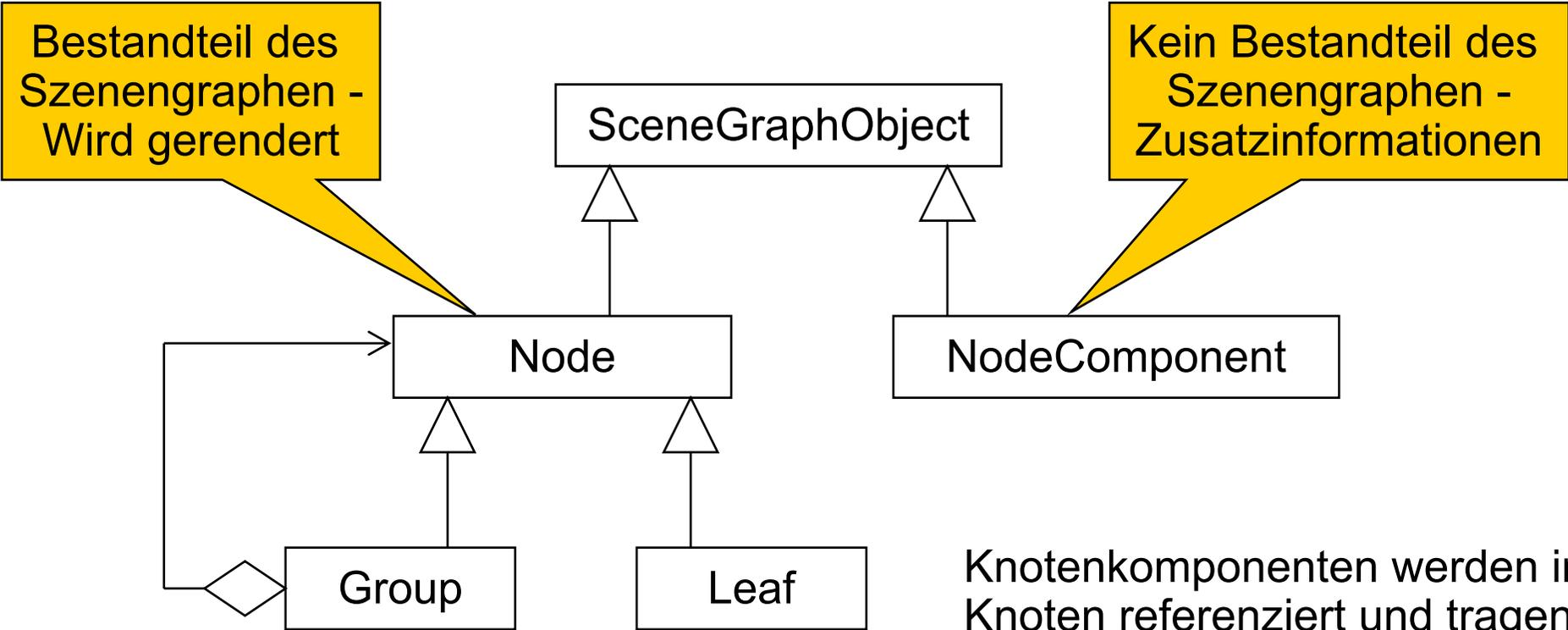
# Beispiel eines Szenengraphen



# Terminologie für Szenengraphen

- *VirtualUniverse*:
  - Üblicherweise die Wurzel des Szenengraphen
- *Locale*:
  - Referenzpunkt im Universum (z.B. für Raum, Gebäude)
  - Typischerweise hat ein *Locale*-Knoten zwei Untergraphen:
    - » *Ansichts-Szenengraph* (*view branch graph*)
    - » *Inhalts-Szenengraph* (*content branch graph*)
- *BranchGroup*:
  - Allgemeiner Begriff für die Wurzel eines Untergraphen
  - Einzig zulässige Kinder für *Locale*-Knoten
- *Node*: Beliebiger Knoten in den Untergraphen
- *Group*: Gruppierungsknoten
- *Leaf*: Blattknoten, z.B. für elementare Objekte, Beleuchtung, Ton
- *NodeComponent*: Knoten, die Zusatzeigenschaften darstellbarer Knoten definieren (Geometrie, Material, Textur, ...)

# Klassenstruktur



Knotenkomponenten werden in Knoten referenziert und tragen indirekt zum Erscheinungsbild bei, z.B. Material, Appearance, Geometry

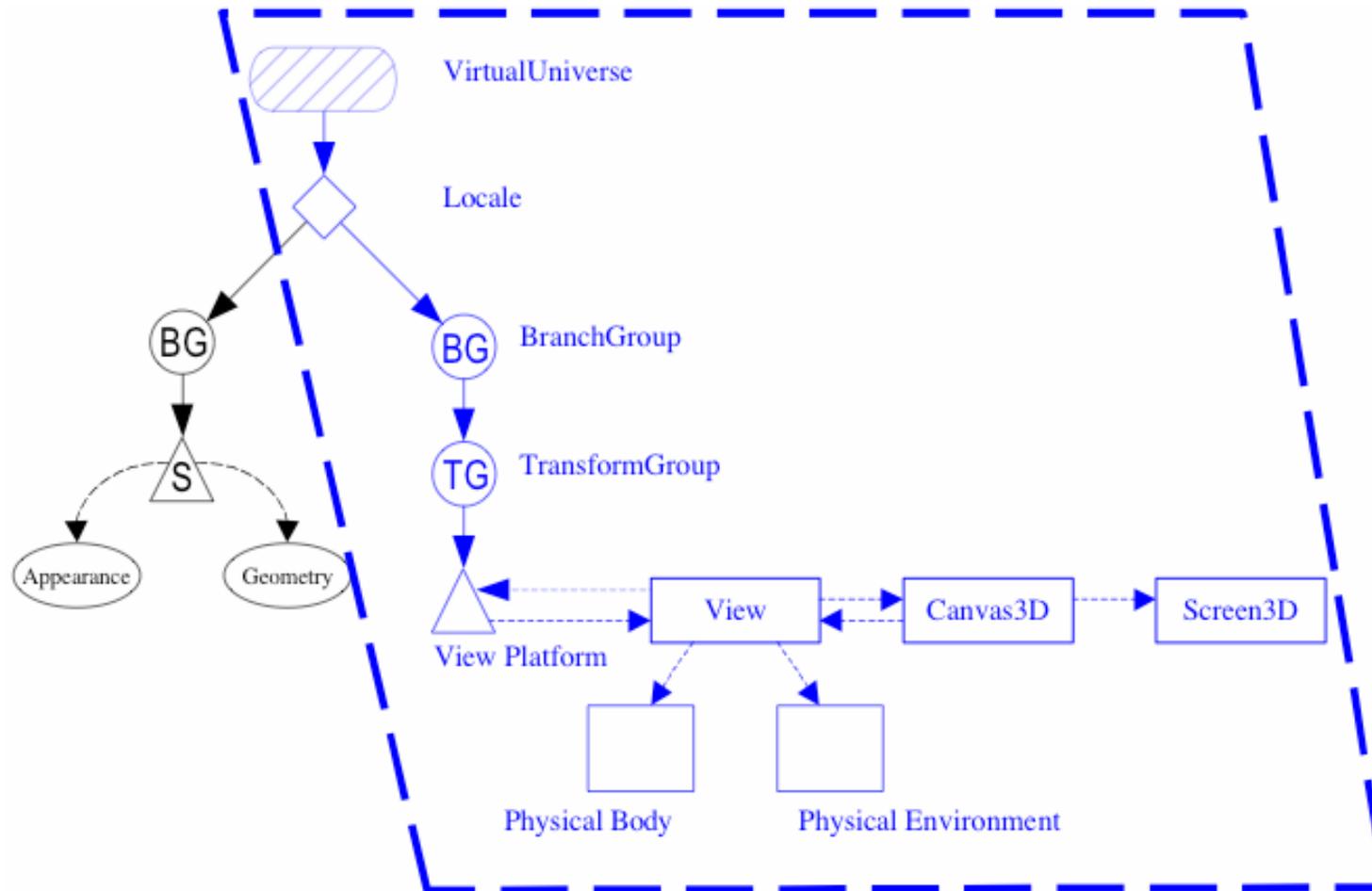
*Composite-*  
Entwurfsmuster

# Basisrezept für Java 3D-Programme

1. Erzeuge ein `Canvas3D`-Objekt
2. Erzeuge ein `VirtualUniverse`-Objekt
3. Erzeuge (mindestens) ein `Locale`-Objekt, binde es an das `VirtualUniverse`-Objekt
4. Erzeuge einen Ansichts-Szenengraph
  - a. Erzeuge ein `View`-Objekt
  - b. Erzeuge ein `ViewPlatform`-Objekt
  - c. Erzeuge ein `PhysicalBody`-Objekt
  - d. Erzeuge ein `PhysicalEnvironment`-Objekt
  - e. Binde `ViewPlatform`-, `PhysicalBody`, `PhysicalEnvironment`- und `Canvas3D`-Objekte an `View`-Objekt an
5. Erzeuge einen Inhalts-Szenengraph
6. Compiliere Untergraphen
7. Füge Untergraphen zum `Locale`-Objekt hinzu

# SimpleUniverse

- Eine Hilfsklasse aus `com.sun.j3d.utils` enthält bereits wesentliche Teile eines Standard-Universums:



# Beispiel: HelloJava3Da.java

```
public class HelloJava3Da extends Applet {

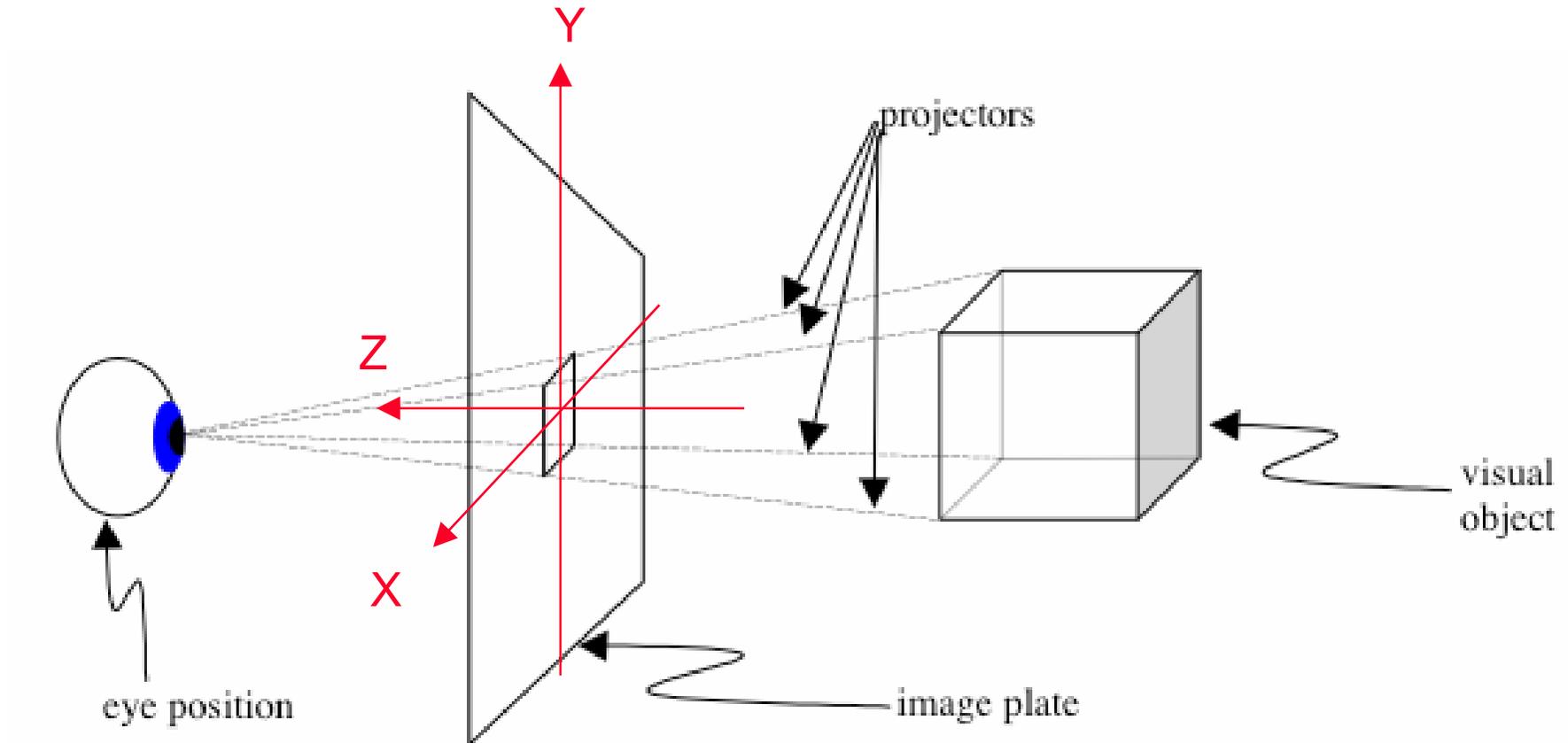
    public HelloJava3Da() {
        setLayout(new BorderLayout());
        Canvas3D canvas3D = new Canvas3D(null);
        add("Center", canvas3D);
        BranchGroup scene = createSceneGraph();
        scene.compile(); //Optional
        SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
        simpleU.getViewingPlatform().setNominalViewingTransform();
        simpleU.addBranchGraph(scene);
    }

    public BranchGroup createSceneGraph() {
        BranchGroup objRoot = new BranchGroup();
        objRoot.addChild(new ColorCube(0.4));
        return objRoot;
    }

    public static void main(String[] args) {
        Frame frame = new MainFrame(new HelloJava3Da(), 256, 256);
    }

}
```

# Konzeptionelle Grundsituation beim Rendering



# Hilfsfunktionen

In SimpleUniverse:

- **ViewingPlatform getViewingPlatform()**
  - Das SimpleUniverse enthält bereits ein **ViewingPlatform**-Objekt
  - Liefert das **ViewingPlatform**-Objekt des SimpleUniverse
- **void addBranchGraph(BranchGroup bg)**
  - Fügt einen Inhalts-Szenengraph zum **Locale**-Objekt des SimpleUniverse hinzu

In ViewingPlatform:

- **void setNominalViewingTransform()**
  - Verschieben der **ViewingPlatform** nach hinten
    - » Um 2,41 m zurück auf (0, 0, 2.41)  
(so dass 2 x 2 m große Objekte (Koordinaten von -1 bis +1) gut sichtbar sind)

# Live und Compiled

- Teilbäume des Szenengraphen können drei Zustände haben
- „Tot“:
  - Szenengraph, der nicht in ein `Local`-Objekt eingebunden ist
- *Live*:
  - Szenengraph ist direkt oder indirekt an ein `Local`-Objekt angebunden
  - Betreffender Szenengraph wird gerendert
  - Änderungen von Eigenschaften sind nur über vorher definierte *capabilities* möglich
- *Compiled*:
  - `BranchGroup`-Objekte können mit `compile()` in eine Form überführt werden, die effizienter gerendert werden kann
  - Empfehlenswert als letzter Schritt vor der *Live*-Schaltung
- Testprädikate:
  - `isLive()`, `isCompiled()`

# Import-Statements für HelloJava3D.java

```
import java.applet.Applet;  
import java.awt.BorderLayout;  
import java.awt.Frame;  
import java.awt.event.*;  
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.*;
```

# ColorCube

- Geometrie-Objekt
  - Aus `sun.com.j3d.utils.geometry`
  - Spezialfall von `Shape3D`
  - Quader mit verschiedenfarbigen Flächen
- Standardort:
  - Mittelpunkt = Koordinatenursprung
- Standardgrösse:
  - 2 x 2 x 2 m
- Konstruktor:
  - `ColorCube(double scale)`
    - `scale` gibt einen Skalierungsfaktor an
    - `ColorCube(0.4)` :  
Ecken an den Koordinatenpositionen +/- 0.4 m

# Transformations-Objekte

- In Analogie zu VRML werden Transformationen durch Knoten im Szenengraphen beschrieben
- **Transform3D**:
  - Von Knoten referenziert (ähnlich zu Knotenkomponenten), nicht direkt Bestandteil des Szenengraphen
  - Repräsentiert 4 x 4-Matrix von Gleitkommazahlen
  - Standard-Konstruktor erzeugt Identitäts-Transformation
  - `void rotX(double angle)`, analog `rotY`, `rotZ`: stellt Matrix als Rotation gegen den Uhrzeigersinn ein
  - Multiplikation (`mul`-Operation) kombiniert Transformationen
  - `void set(Vector3f translate)`: stellt Matrix als Translation um gegebenen 3D-Vektor (Float) ein
- **TransformGroup**:
  - Knoten, Bestandteil des Szenengraphen
  - Beschreibung der genauen Transformation durch **Transform3D**-Objekte
  - **Transform3D**-Objekt kann übergeben werden:
    - » Als Konstruktorargument
    - » Mit `setTransform(Transform3D t)`

# Beispiel: HelloJava3Db.java

```
public BranchGroup createSceneGraph() {  
  
    BranchGroup objRoot = new BranchGroup();  
  
    Transform3D rotate = new Transform3D();  
    Transform3D tempRotate = new Transform3D();  
    rotate.rotX(Math.PI/4.0d); // 45 Grad  
    tempRotate.rotY(Math.PI/5.0d); // 36 Grad  
    rotate.mul(tempRotate);  
    TransformGroup objRotate =  
        new TransformGroup(rotate);  
  
    objRoot.addChild(objRotate);  
    objRotate.addChild(new ColorCube(0.4));  
    objRoot.compile();  
  
    return objRoot;  
}
```

# Capabilities

- Szenengraph *live* oder *compiled*
  - ==> Eigenschaften von Objekten festgeschrieben
  - Ermöglicht Optimierungen beim Rendering
- *Capability-Bits*:
  - Definiert für jedes Szenengraph-Objekt
  - Setzen des Bits bedeutet Änderbarkeit der entsprechenden Eigenschaft

```
void clearCapability(int bit)
```

```
Boolean getCapability(int bit)
```

```
void setCapability(int bit)
```

- Beispiel: Capabilities für TransformGroup-Knoten

```
ALLOW_TRANSFORM_READ
```

```
ALLOW_TRANSFORM_WRITE
```

```
ALLOW_CHILDREN_READ
```

```
ALLOW_CHILDREN_WRITE
```

```
ALLOW_CHILDREN_EXTEND
```

# B6. 3D-Computergrafik mit Java

B6.1 Grundlagen der 3D-Computergrafik

B6.2 Einführung in Java 3D

B6.3 Animation



B6.4 Beleuchtung

## Literatur:

- D. Selman: Java 3D Programming, Manning 2002
- A. E. Walsh, D. Gehringer: Java 3D API Jump-Start, Prentice-Hall 2001
- <http://java.sun.com/products/java-media/3D/>
- <http://java.sun.com/developer/onlineTraining/java3d/>
- <http://java.sun.com/docs/books/java3d/>
- <http://www.j3d.org>, <http://www.java3d.org/>

# Animation: Behavior

- Mit **Behavior**-Objekten können Objekte dynamisches Verhalten aufweisen (Animation)
- *Scheduling region*:
  - Bereich des Raums, in dem sich das Verhalten abspielt
  - Verhalten wird erst berechnet, wenn sich das sogenannte *activation volume* des Betrachterstandpunkts (der **ViewingPlatform**) mit der *scheduling region* des Verhaltens überschneidet
  - Zweck: Reduzierung des Berechnungsaufwandes
  - Einfache Möglichkeit zur Erzeugung einer *scheduling region*:
    - » Kugel **BoundingSphere(Point3d center, double radius)**
    - » **setSchedulingBounds(Bounds region)**
    - » **BoundingSphere**-Objekt als **region** übergeben
- Einfache Behavior-Objekte: *Interpolatoren*
  - Analog zu VRML
  - Beispiel **RotationInterpolator**
  - Verhalten abhängig von einem Parameter, der zwischen 0 und 1 variiert wird
    - » Java3D-Bezeichnung: *Alpha-Wert*

# Zeitabhängige Werte in Java 3D

- Generierung von zeitabhängigen Alpha-Parameterwerten
  - Objekt der Klasse `Alpha`
  - Analog zu *TimeSensor* in VRML, etwas vereinfacht

Konstruktor

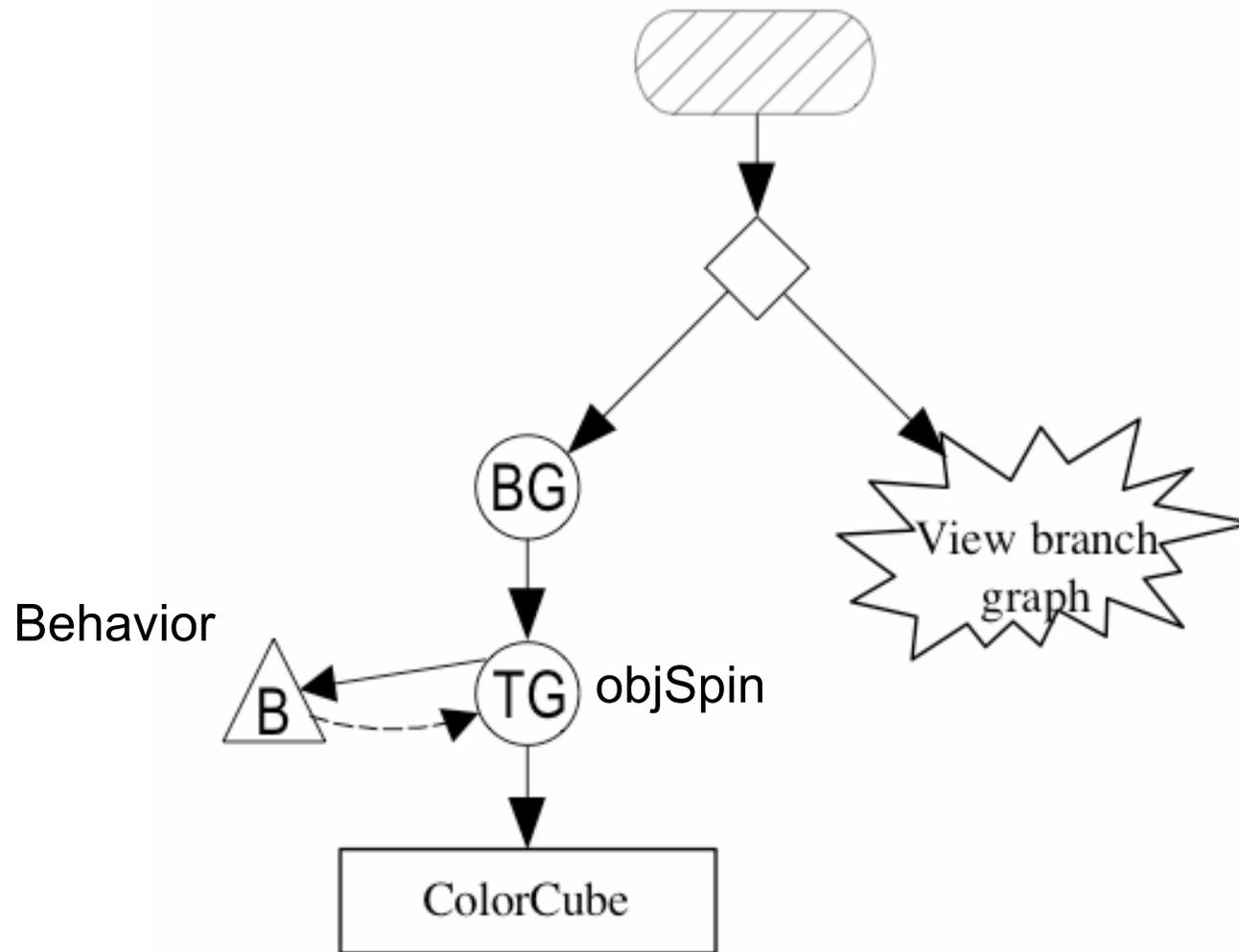
`Alpha(int loopCount, long increasingAlphaDuration)`

- *loopCount*: Anzahl der gewünschten Durchläufe, in denen der Wert linear von 0 auf 1 ansteigt
  - Wert `-1` bedeutet unendlich viele Wiederholungen
- *increasingAlphaDuration*: Dauer einer Periode in Millisekunden
  - Während dieser Zeitdauer steigt der Wert kontinuierlich von 0 auf 1

# Beispiel: HelloJava3Dc.java

```
public BranchGroup createSceneGraph() {  
  
    BranchGroup objRoot = new BranchGroup();  
  
    TransformGroup objSpin = new TransformGroup();  
    objSpin.setCapability  
        (TransformGroup.ALLOW_TRANSFORM_WRITE);  
    objRoot.addChild(objSpin);  
    objSpin.addChild(new ColorCube(0.4));  
  
    Alpha rotationAlpha = new Alpha(-1, 4000);  
    RotationInterpolator rotator =  
        new RotationInterpolator(rotationAlpha, objSpin);  
  
    BoundingSphere bounds = new BoundingSphere();  
    rotator.setSchedulingBounds(bounds);  
    objSpin.addChild(rotator);  
  
    return objRoot;  
}
```

# Szenengraph für animierte Szene



# B6. 3D-Computergrafik mit Java

B6.1 Grundlagen der 3D-Computergrafik

B6.2 Einführung in Java 3D

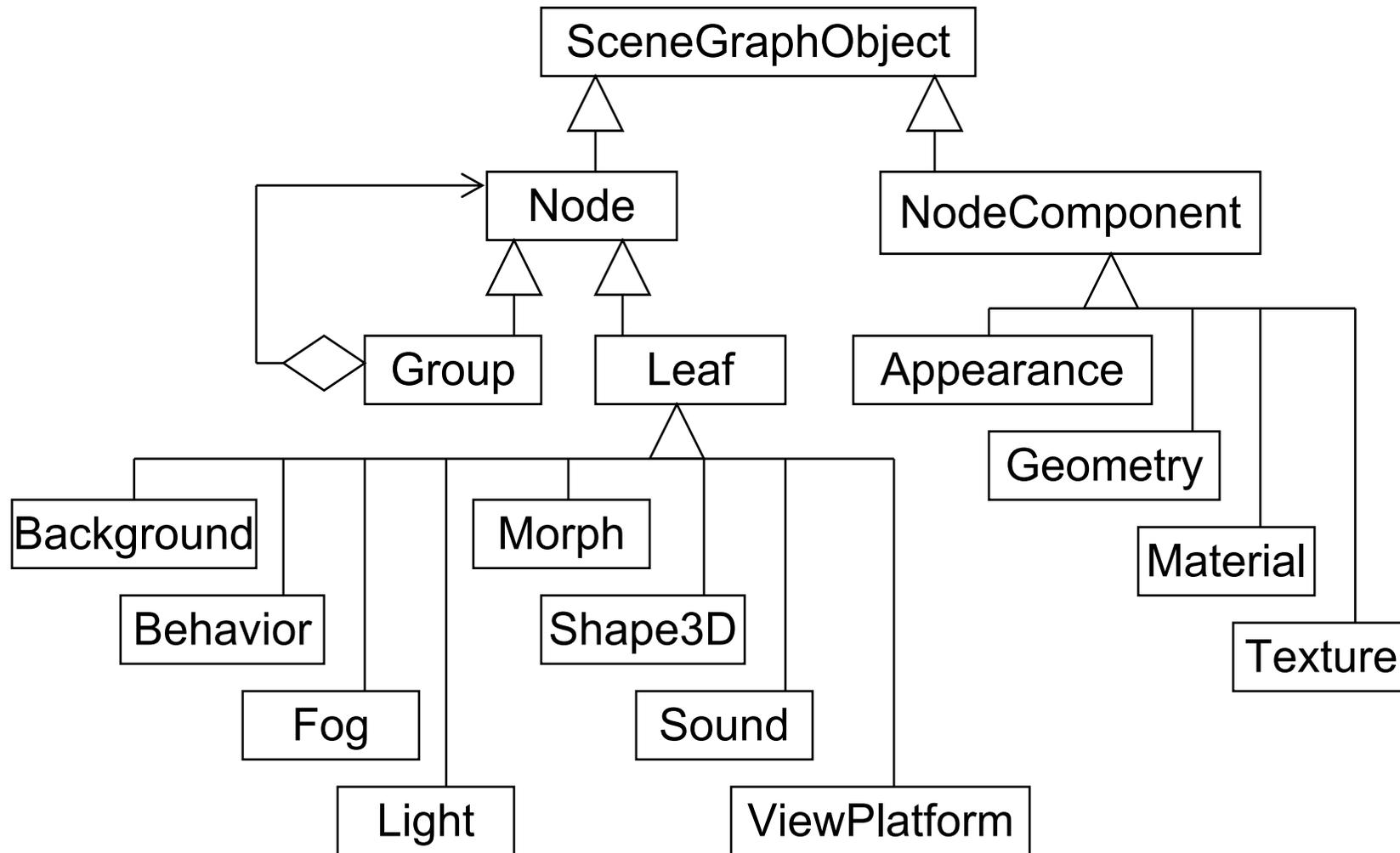
B6.3 Animation

B6.4 Geometrie, Material, Beleuchtung 

## Literatur:

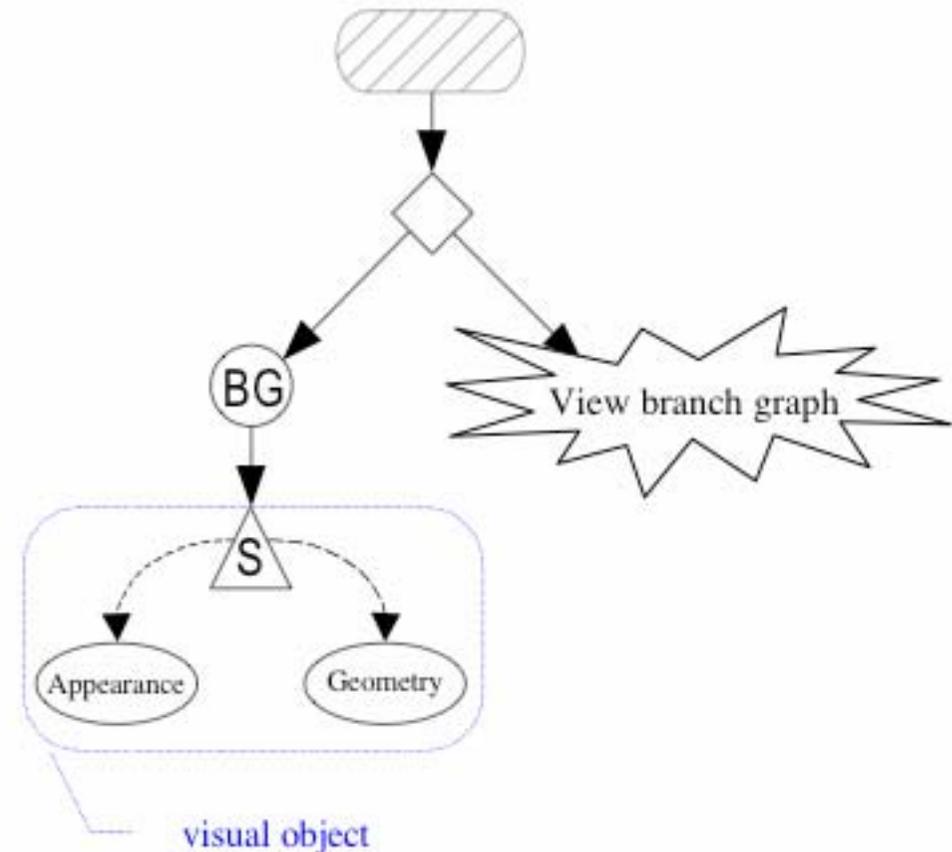
- D. Selman: Java 3D Programming, Manning 2002
- A. E. Walsh, D. Gehringer: Java 3D API Jump-Start, Prentice-Hall 2001
- <http://java.sun.com/products/java-media/3D/>
- <http://java.sun.com/developer/onlineTraining/java3d/>
- <http://java.sun.com/docs/books/java3d/>
- <http://www.j3d.org>, <http://www.java3d.org/>

# Verfeinerte Klassenstruktur



# Formen in Java 3D: Shape3D

- Knoten der Klasse **Shape3D** definieren Blattknoten für sichtbare Objekte
  - Referenz auf **Geometry**-Knotenkomponente
  - Referenz auf **Appearance**-Knotenkomponente (optional)
  - Festlegung der Referenzen im Konstruktor oder mit set-Methoden
- Capability Bits für **Shape3D**:
  - ALLOW\_APPEARANCE\_READ | ...WRITE
  - ALLOW\_GEOMETRY\_READ | ...WRITE
  - ALLOW\_COLLISION\_BOUNDS\_READ | ...WRITE



# Geometrische Primitive

- Java 3D unterstützt die gleichen primitiven geometrischen Objekte wie VRML
  - Box, Cone, Cylinder, Sphere
  - Klassen in `com.sun.j3d.utils.geometry`
  - Unterklassen von `Group`
- Konstruktoren spezifisch für die jeweiligen Formen
  - Länge, Breite, Radius etc.
- Allgemeine Funktionen für alle geometrischen Primitive (deklariert in `com.sun.j3d.utils.geometry.Primitive`)
  - `Shape3D getShape(int partid)`
    - Liefert das Unterelement mit gegebener Nummer
  - `void setAppearance(Appearance app)`
    - Einstellen einer gewünschten Präsentationsform
- **ColorCube**:
  - Spezialklasse (abgeleitet von `Shape3D`) vor allem zu Testzwecken
- Allgemein: Formen aus Polygonen zusammensetzbar

# Beispiel zu Geometrischen Primitiven

```
public BranchGroup createSceneGraph() {
    BranchGroup objRoot = new BranchGroup();
    ...
    TransformGroup objSpin = new TransformGroup();
    objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    ...
    objRoot.addChild(objRotate);
    objRotate.addChild(objSpin);

    Appearance app = new Appearance();
    app.setColoringAttributes(
        new ColoringAttributes
            (new Color3f(1.0f, 0.0f, 0.0f),
             ColoringAttributes.SHADE_FLAT));
    objSpin.addChild(new Box(0.4f, 0.4f, 0.4f, app));

    ...
    RotationInterpolator rotator = ...;
    ...
    objSpin.addChild(rotator);

    return objRoot;
}
```

# Appearance und Attribute

- Appearance-Knoten können eine Vielzahl von Attributen tragen:
  - PointAttributes
  - LineAttributes
  - PolygonAttributes
  - ColoringAttributes
  - TransparencyAttributes
  - RenderingAttributes
  - Material
  - TextureAttributes
  - Texture
  - TexCoordGeneration
- Z.B. Färbungen können auf verschiedenen Wegen erreicht werden:
  - einfach mit ColoringAttributes
  - komplex mit Material und Lighting

# Beispiel zu Material und Lighting

```
public BranchGroup createSceneGraph() {
    BranchGroup objRoot = new BranchGroup();
    ...
    TransformGroup objSpin = new TransformGroup();
    ...
    Appearance app = new Appearance();
    Material mat = new Material();
    mat.setDiffuseColor(new Color3f(1.0f, 0.0f, 0.0f));
    app.setMaterial(mat);
    objSpin.addChild(new Box(0.4f, 0.4f, 0.4f, app));
    ...
    DirectionalLight light1 = new DirectionalLight();
    light1.setDirection(new Vector3f(1.0f, -1.0f, -0.5f));
    light1.setInfluencingBounds(new BoundingSphere());
    objRoot.addChild(light1);

    SpotLight light2 = new SpotLight();
    light2.setDirection(new Vector3f(-1.0f, +1.0f, 0.f));
    light2.setPosition(2.0f, -1.0f, 0.0f);
    light2.setInfluencingBounds(new BoundingSphere());
    objRoot.addChild(light2);

    return objRoot;
}
```