Computergrafik - Übungen
W. Kurth, Th. Mangoldt

Java 3D
Einstiegs-Tutorial
Teil 2

Steuerung des Erscheinungsbildes (Appearance):

- wie in VRML können Materialeigenschaften (optische Eigenschaften) für die Wechselwirkung mit Beleuchtung festgelegt werden (siehe später bei Lichtquellen)
- Festlegung von "selbstleuchtenden" Farben mit **`ColoringAttributes`**
- ferner gibt es z.B. **`LineAttributes`** (Strichdicke, gestrichelt...) für Linien, **`PolygonAttributes`** (Normalen-Richtung, Einschalten von back face culling etc.), **`TransparencyAttributes`** ...

Konstruktor:
`Appearance()`
Default-Werte: Punkte und Linien mit Dicke 1 Pixel, ohne Antialiasing, Farbe Weiß, keine Transparenz

Methoden von **`Appearance`**:

```
void setPointAttributes(PointAttributes p)
void setLineAttributes(LineAttributes l)
void setPolygonAttributes(PolygonAttributes a)
void setColoringAttributes(ColoringAttributes c)
```

usw.

Mehrere Instanzen von **`Appearance`** können auf dieselben **`ColoringAttributes`**, **`PointAttributes`** etc. zugreifen.

Konstruktoren und Methoden der Attributklassen:

```
PointAttributes()
PointAttributes(float punktgroesse, boolean ant)
```

(Punktgröße in Pixeln, **`ant`**: Antialiasing eingeschaltet)

```
void setPointSize(float punktgroesse)
void setPointAntialiasingEnable(boolean ant)
```

```
PolygonAttributes()
PolygonAttributes(int pmode, int cull, float offset)
```

pmode ist **POLYGON_POINT, POLYGON_LINE** oder **POLYGON_FILL**,
cull ist **CULL_FRONT, CULL_BACK** oder **CULL_NONE**

PolygonAttributes-Methoden:

```
void setCullFace(int cull)
void setPolygonMode(int pmode)
void setPolygonOffset(float offset)
void setBackFaceNormalFlip(boolean flip)
```

flip bewirkt automatisches Umdrehen der Normalen von Rückseitenpolygonen (und damit deren Sichtbarmachen)

```
ColoringAttributes()
```

defaults: weiß, **SHADE_GOURAUD**

```
ColoringAttributes(Color3f color, int shademodel)
ColoringAttributes(float r, float g, float b, int
shademodel)
```

shademodel ist **SHADE_GOURAUD, SHADE_FLAT, FASTEST** oder **NICEST**

Methoden:

```
void setColor(Color3f color)
void setColor(float r, float g, float b)
void setShadeModel(int shademodel)
```

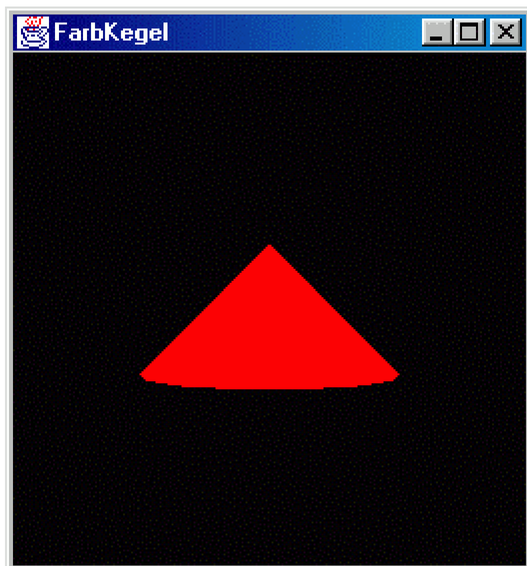
Beispiel:
Ein Primitivobjekt (Kegel) wird rot gefärbt.
Gleichzeitig zeigt das Beispiel, wie dieser rote Kegel eine eigene Klasse **MeinFarbKegel** definiert.
Es wird das komplette Java-Programm gezeigt.

```
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 FarbKegel extends Applet
   {

   // selbstgeschriebene VisualObject class:
   public class MeinFarbKegel extends Shape3D

      {
      private BranchGroup meineBG;

      public MeinFarbKegel()
         {
         meineBG = new BranchGroup();
         Cone kegel = new Cone(0.5f, 0.5f);
         kegel.setAppearance(macheFarbe());
         meineBG.addChild(kegel);
         meineBG.compile();
         }

      Appearance macheFarbe()
         {
         Appearance a = new Appearance();
         ColoringAttributes c_att = new
            ColoringAttributes();
         c_att.setColor(new
            Color3f(1.0f, 0.0f, 0.0f));
         a.setColoringAttributes(c_att);
         return a;
         }

      public BranchGroup getBG()
         {
         return meineBG;
         }
      } // end class MeinFarbKegel

   public BranchGroup macheSzenengraph()
      {
      BranchGroup objWurzel =
         new MeinFarbKegel().getBG();
      objWurzel.compile();
      return objWurzel;
      } // end macheSzenengraph
```

```
public FarbKegel()
    {
    setLayout(new BorderLayout());
    GraphicsConfiguration config =
      SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas = new Canvas3D(config);
    add("Center", canvas);
    BranchGroup szene = macheSzenengraph();
    szene.compile();
    SimpleUniverse u = new SimpleUniverse(canvas);
    u.getViewingPlatform(
       ).setNominalViewingTransform();
    u.addBranchGraph(szene);
    } // end Konstruktor

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

Ergebnis:



Laden von externen Szenen-Dateien

Eine Alternative zur Generierung eigener Szenen in Java 3D bietet der Import von Dateien, die mit anderen Mitteln erzeugt wurden (Liste der verfügbaren Loader siehe Teil 1 des Kurses).

Im Standard-Umfang des Java3D-Downloads ist die Zahl der Loader beschränkt; weitere können über eine Loader-Site beschafft werden.

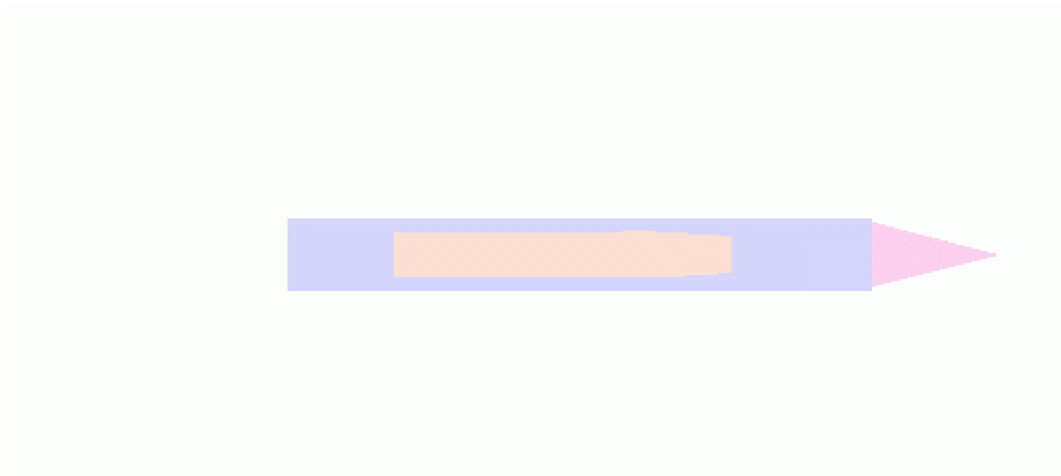
Hier ein Beispiel für den Import einer VRML-Datei (man beachte das Abfangen der Exceptions). Es wird das vollständige Programm gezeigt:

```
import com.sun.j3d.loaders.*;
import com.sun.j3d.loaders.vrml97.VrmlLoader;
import com.sun.j3d.loaders.ParsingErrorException;
import com.sun.j3d.loaders.IncorrectFormatException;
import com.sun.j3d.loaders.Scene;
import java.applet.Applet;
import java.awt.*;
//import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import java.io.*;

public class Ladevrml extends Applet
{

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

   Loader f = new VrmlLoader();
   Scene s = null;
```

```
    try
       { s = f.load("vogel1.wrl"); }



    catch (FileNotFoundException e)
       {
       System.err.println(e);
       System.exit(1);
       }
    catch (ParsingErrorException e)
       {
       System.err.println(e);
       System.exit(1);
       }
    catch (IncorrectFormatException e)
       {
       System.err.println(e);
       System.exit(1);
       }

    objWurzel.addChild(s.getSceneGroup());

    AmbientLight licht = new AmbientLight();
    licht.setInfluencingBounds(new BoundingSphere());
    objWurzel.addChild(licht);

    return objWurzel;
    }

 public Ladevrml()
    {
    setLayout(new BorderLayout());
    GraphicsConfiguration config =
       SimpleUniverse.getPreferredConfiguration();
    Canvas3D c = new Canvas3D(config);
    add("Center", c);
    BranchGroup szene = macheSzenengraph();
    SimpleUniverse u = new SimpleUniverse(c);
    u.getViewingPlatform(
       ).setNominalViewingTransform();
    u.addBranchGraph(szene);
    }
```

```
public static void main(String[] args)
   {
   Frame frame =
      new MainFrame(new Ladevrml(), 700, 700);
   }
}
```

Ergebnis (farblich invertiert):



Vektorielle Klassen

Für die Geometrie-Erzeugung mit *boundary representation* - Objekten wird auf Klassen aus d. Package **`javax.vecmath.*`** zurückgegriffen.

Systematik:

`Point*`	für Koordinaten
`Color*`	für Farbspezifikation (RGB oder RGBA)
`Vector*`	für Richtungsvektoren und Normalen
`TexCoord*`	für Texturkoordinaten

`*` besteht aus einer Kombination der Dimensionszahl (2, 3 oder 4) mit der Typspezifikation für die Komponenten (b, f oder d).

Die abstrakte Oberklasse ist Tuple*, mit Konstruktoren:

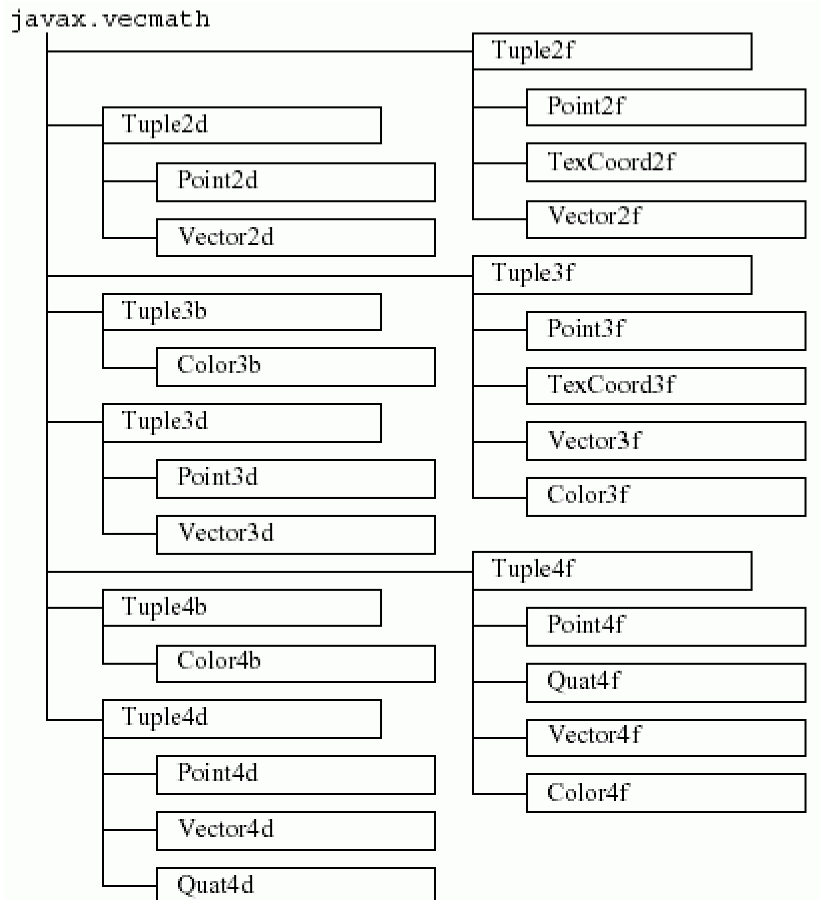
`Tuple2f()` default: (0; 0)
`Tuple2f(float x, float y)`
`Tuple2f(float[] t)`
`Tuple2f(Tuple2f t)`

`Tuple2f(Tuple2d t)`

Methoden:

```
void set(float x, float y)
void set(float[] t)
boolean equals(Tuple2f t1)
final void add(Tuple2f t1)
void add(Tuple2f t1, Tuple2f t2)
void sub(Tuple2f t1, Tuple2f t2)
void sub(Tuple2f t1)
void negate()
void negate(Tuple2f t1)
void absolute()
void absolute(Tuple2f t)
```

Die *public members* heißen **`x`** und **`y`** (für **`Tuple3*`**: x, y, z; für **`Tuple4*`**: x, y, z, w).
Die Konstruktoren und Methoden für Tuple2d usw. sind analog aufgebaut.
Klassenhierarchie:



Spezielle Methoden der **`Point`**-Klassen:

`float distance(Point3f p1)`
`float distanceSquared(Point3f p1)`
`float distanceL1 (Point3f p1)`   // Manhattan-Distanz

Color-Klassen:
mit float (zwischen 0 und 1) oder byte (zwischen 0 und 255)-Angaben für RGB oder RGBA (alpha-Kanal, Transparenz)

es kann nützlich sein, sich eigene Konstanten zu def., z.B.
`Color3f rot = new Color3f(1.0f, 0.0f, 0.0f)`

Spezielle Methoden der Vektor-Klassen:

```
float length()
float lengthSquared()
void cross(Vector3f v1, Vector3f v2)
                                        // Kreuzprodukt
float dot(Vector3f v1)                  // Skalarprodukt
void normalize()
void normalize(Vector3f v1)
float angle(Vector3f v1)
```

// Winkel in Bogenmaß zw. 0 und PI

Geometrie-Klassen

- Nicht-indexbasierte *boundary representations*
- Indexbasierte *boundary representations* (vgl. VRML)
- sonstige Geometrie-Klassen (**`Text3D`**, **`Raster`**...)

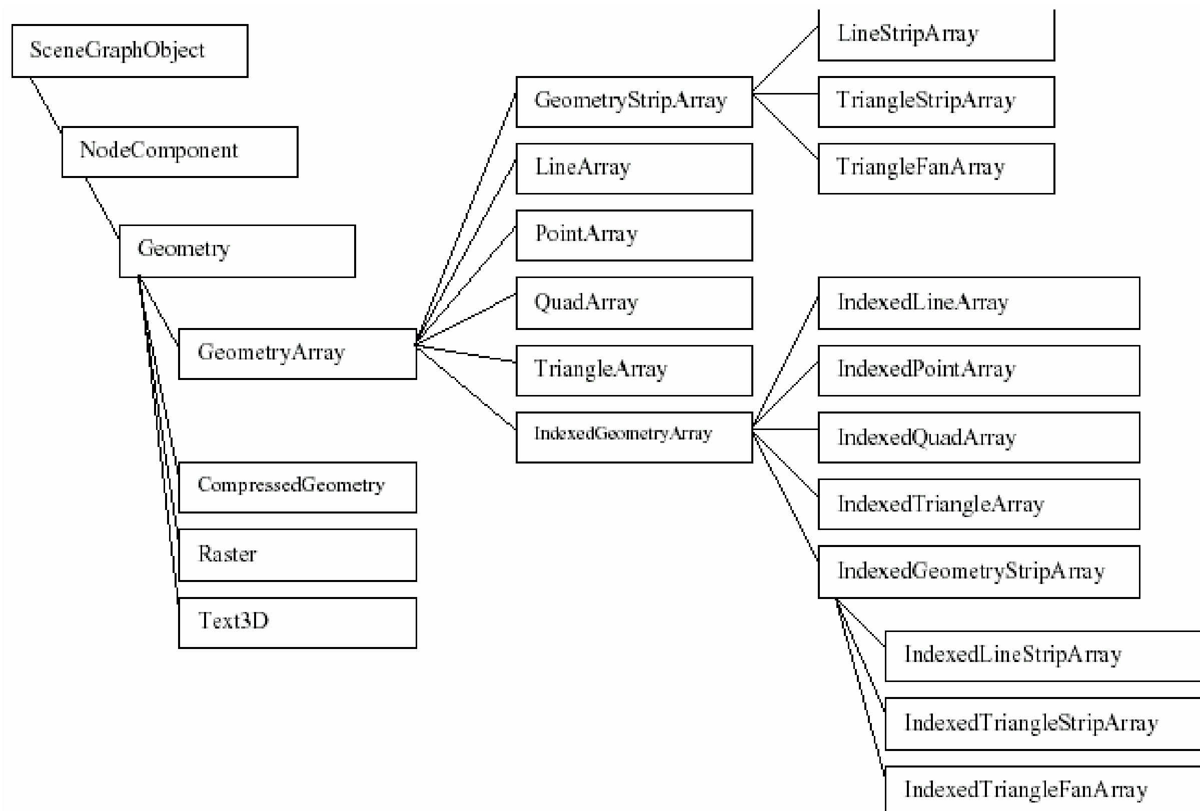
Abstrakte Oberklasse ist **`GeometryArray`**.

Beim Konstruktor-Aufruf werden gewöhnlich die Anzahl der verwendeten Ecken **`v`** und eine Kontrollzahl **`c`** für die interne Formatierung angegeben:
`GeometryArray(int v, int c)`

Die Kontrollzahl **`c`** besteht aus bitweise angeordneten Flags, die einzeln mit Schlüsselwörtern spezifiziert und mit bitweisem Oder (|) verknüpft werden können:

`COORDINATES` (dieses bit muss gesetzt sein)
`NORMALS` Erzeugung von Normalenvektoren
`COLOR_3` Farbangabe RGB für jede Ecke erwartet
`COLOR_4` Farbangabe RGBA für jede Ecke
`TEXTURE_COORDINATE_2`
`TEXTURE_COORDINATE_3`

Klassenhierarchie:

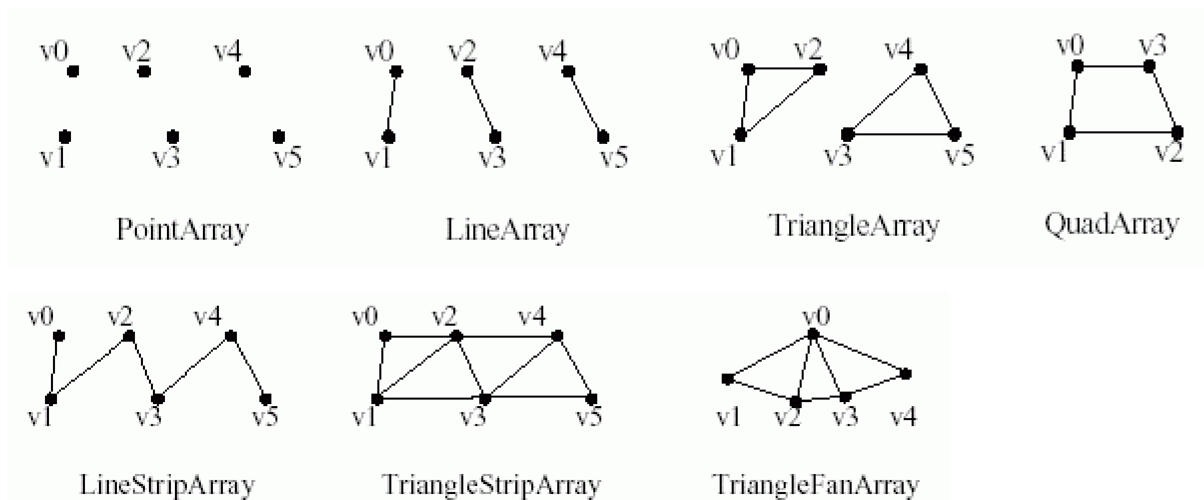


Strip = Streifen, *Fan* = Fächer.
Bei den Klassen ohne "Indexed" müssen für jedes Element (Linie, Dreieck, Viereck) alle Eckpunkte einzeln spezifiziert sein (also z.B. für einen Würfel jede Ecke insgesamt 3 mal) – Ausnahme: gemeinsame Nutzung von Nachbarecken bei strip und fan.
Bei den "Indexed"-Klassen wird diese Redundanz vermieden durch zusätzliche Angabe eines Index-Feldes.
Aber: "Indexed"-Klassen sind ineffizienter beim Rendering!

Konstruktoren für die *strip*- und *fan*-Klassen:

```
LineStripArray(int v, int c,
   int stripVertexCounts[])
```

(analog für TriangleStripArray, TriangleFanArray etc.)
Der zusätzliche Array-Parameter gibt an, wieviele Ecken jeweils zu 1 Streifen gehören.

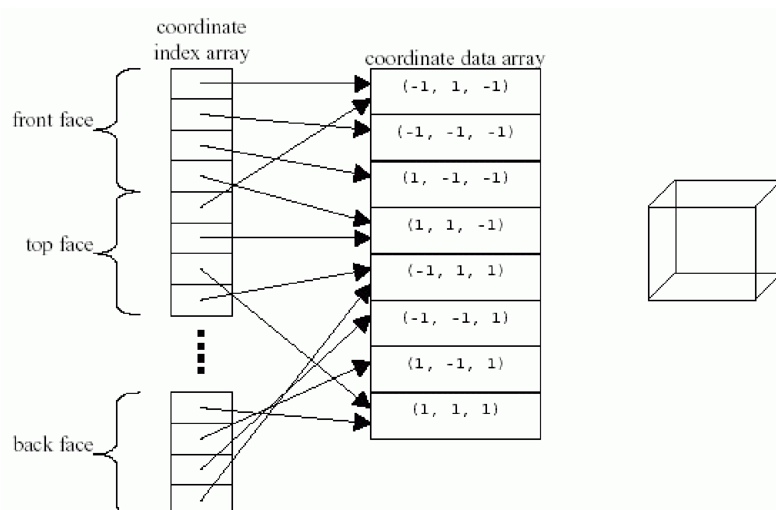


Beachte:

- beim QuadArray, LineStripArray, TriangleStripArray und TriangleFanArray können i.allg. auch mehrere der gezeigten Figuren in einem Objekt spezifiziert werden (deshalb Arrays!)
- die Dreiecke und Vierecke (Quads) werden defaultmäßig gefüllt dargestellt
- Java3D unterstützt keine gefüllten Primitive mit mehr als 4 Seiten. Komplexere Polygone müssen trianguliert oder in Vierecke zerlegt werden. Es gibt eine Triangulator-Utilityklasse, die komplexe Polygone trianguliert.

Bei den Indexed-Klassen gibt es bis zu 4 Index-Arrays: für die Positions- (Koordinaten-) Zuordnung, für die Farben-, Normalenvektoren- und Texturkoordinaten-Zuordnung.

Prinzip der Indizierung am Beispiel eines Würfels (vgl. auch Vorlesung, Kapitel über boundary representation, und VRML):



zugehörige Konstruktoren und Methoden:

IndexedGeometryArray and Subclasses Constructors

Constructs an empty object with the specified number of vertices, vertex format, and number of indices in this array.

```
IndexedGeometryArray(int vertexCount, int vertexFormat, int indexCount)
IndexedPointArray(int vertexCount, int vertexFormat, int indexCount)
IndexedLineArray(int vertexCount, int vertexFormat, int indexCount)
IndexedTriangleArray(int vertexCount, int vertexFormat, int indexCount)
IndexedQuadArray(int vertexCount, int vertexFormat, int indexCount)
```

IndexedGeometryStripArray and Subclasses Constructors

Constructs an empty object with the specified number of vertices, vertex format, number of indices in this array, and an array of vertex counts per strip.

```
IndexedGeometryStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedLineStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedTriangleStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedTriangleFanArray(int vc, int vf, int ic, int stripVertexCounts[]))
```

IndexedGeometryArray Methods (partial list)

`void setCoordinateIndex(int index, int coordinateIndex)`
Sets the coordinate index associated with the vertex at the specified index for this object.

`void setCoordinateIndices(int index, int[] coordinateIndices)`
Sets the coordinate indices associated with the vertices starting at the specified index for this object.

`void setColorIndex(int index, int colorIndex)`
Sets the color index associated with the vertex at the specified index for this object.

`void setColorIndices(int index, int[] colorIndices)`
Sets the color indices associated with the vertices starting at the specified index for this object.

`void setNormalIndex (int index, int normalIndex)`
Sets the normal index associated with the vertex at the specified index for this object.

`void setNormalIndices (int index, int[] normalIndices)`
Sets the normal indices associated with the vertices starting at the specified index for this object.

`void setTextureCoordinateIndex (int index, int texCoordIndex)`
Sets the texture coordinate index associated with the vertex at the specified index for this object.

`void setTextureCoordinateIndices (int index, int[] texCoordIndices)`
Sets the texture coordinate indices associated with the vertices starting at the specified index for this object.

(aus dem Java3D-Tutorial von Sun)

Beispiel:
Es wird ein Oktaeder konstruiert.

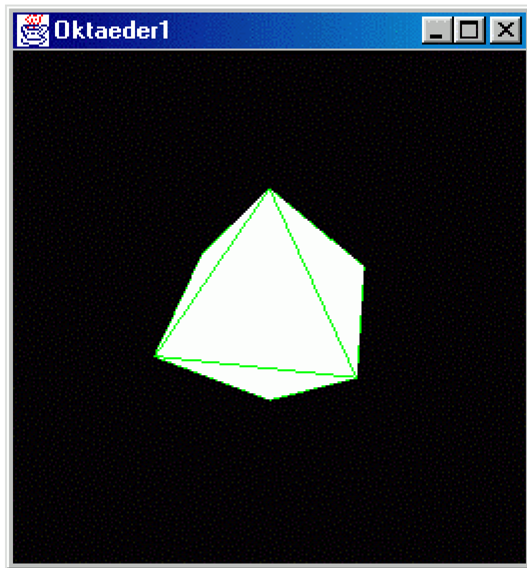
Um verschiedene Möglichkeiten im selben Programm zu zeigen, werden eine Mantelfläche (6 Dreiecke) als **`TriangleStripArray`** und "Boden" und "Deckel" (die 2 restlichen Dreiecke) als zusätzliches **`TriangleArray`** definiert (man wäre auch mit einer Klasse ausgekommen).
Darüberhinaus wird das System der Kanten als grün gefärbtes **`IndexedLineArray`** modelliert.

Es wird nur ein Fragment des Codes (die entscheidende Methode "macheSzenengraph") gezeigt, der Rest ist analog zum 1. Beispielprogramm (Farbwürfel).

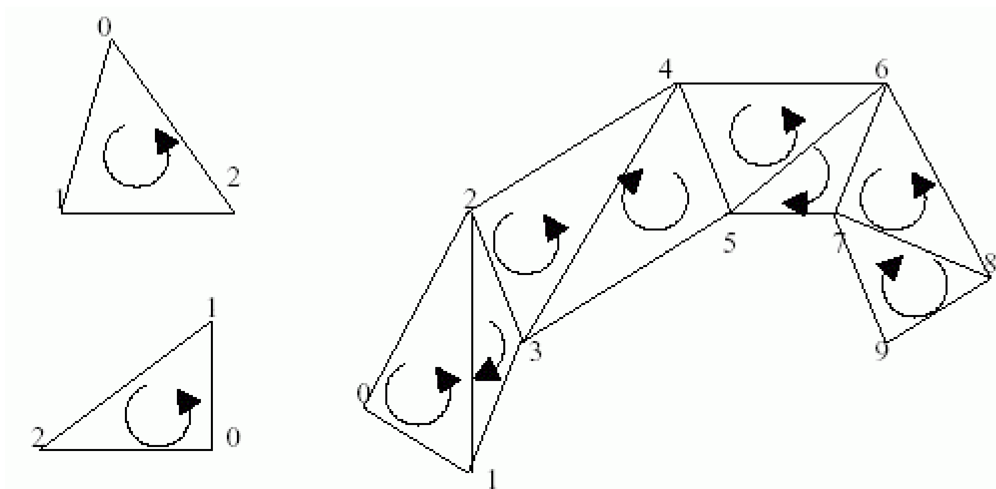
```
import com.sun.j3d.utils.geometry.*;
// ....
public BranchGroup macheSzenengraph()
   {
   BranchGroup objWurzel = new BranchGroup();
   BranchGroup oktaeder = new BranchGroup();

   TriangleStripArray mantel;
   TriangleArray deckel;
   IndexedLineArray kanten;

   Point3f ecke[] = new Point3f[8];
   int stripzahl[] = {8};
   ecke[0] = new Point3f(0.0f, 0.5f, 0.0f);
   ecke[1] = new Point3f(0.0f, 0.0f, 0.5f);
   ecke[2] = new Point3f(0.5f, 0.0f, 0.0f);
   ecke[3] = new Point3f(0.0f, -0.5f, 0.0f);
   ecke[4] = new Point3f(0.0f, 0.0f, -0.5f);
   ecke[5] = new Point3f(-0.5f, 0.0f, 0.0f);
   ecke[6] = ecke[0];
   ecke[7] = ecke[1];
   mantel = new TriangleStripArray(8,
      TriangleStripArray.COORDINATES, stripzahl);
   mantel.setCoordinates(0, ecke);
   deckel = new TriangleArray(6,
      TriangleArray.COORDINATES);
```

```
deckel.setCoordinate(0, ecke[1]);
deckel.setCoordinate(1, ecke[5]);
deckel.setCoordinate(2, ecke[3]);
deckel.setCoordinate(3, ecke[0]);
deckel.setCoordinate(4, ecke[2]);
deckel.setCoordinate(5, ecke[4]);

int[] indices = {
   0, 1,    0, 2,    0, 4,    0, 5,
   3, 1,    3, 2,    3, 4,    3, 5,
   1, 2,    2, 4,    4, 5,    5, 1 };
kanten = new IndexedLineArray(8,
   IndexedLineArray.COORDINATES, 24);
kanten.setCoordinates(0, ecke);
kanten.setCoordinateIndices(0, indices);

// Kanten rot faerben:
Appearance app = new Appearance();
ColoringAttributes c_att = new
   ColoringAttributes();
c_att.setColor(new Color3f(0.0f, 1.0f, 0.0f));
app.setColoringAttributes(c_att);

oktaeder.addChild(new Shape3D(mantel));
oktaeder.addChild(new Shape3D(deckel));
oktaeder.addChild(new Shape3D(kanten, app));

// Transformation, Komposition von 2 Rotationen:
Transform3D drehung = new Transform3D();
Transform3D drehung2 = new Transform3D();
drehung.rotX(Math.PI / 5.0d);
drehung2.rotY(Math.PI / 5.0d);
drehung.mul(drehung2);
TransformGroup objDreh = new
   TransformGroup(drehung);

objDreh.addChild(oktaeder);
objWurzel.addChild(objDreh);
return objWurzel;
} // end macheSzenengraph-Methode
```

Ergebnis:

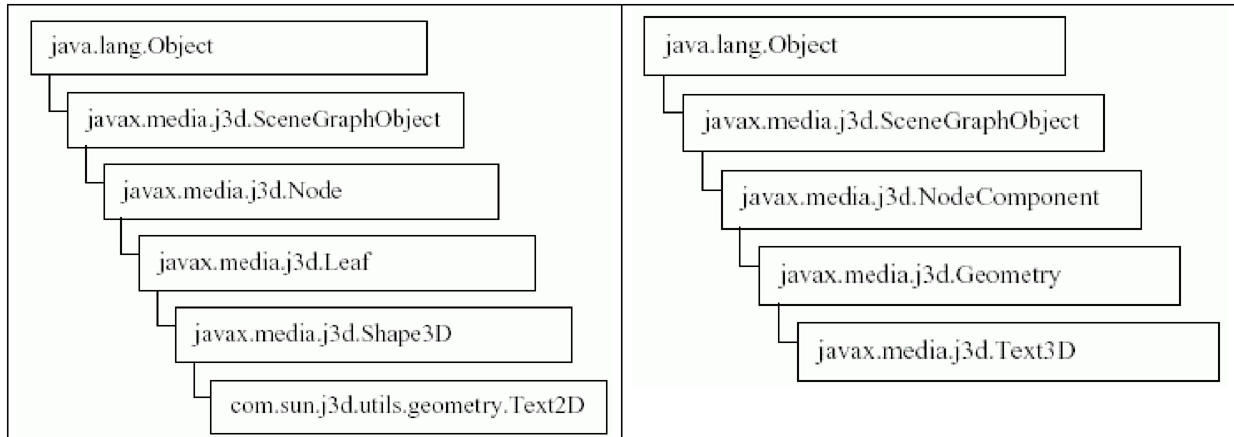


Beachte:
Bei TriangleStripArrays wird bei der Bestimmung der Vorderseiten der Dreiecke die übliche Regel (Orientierung gegen den Uhrzeigersinn) bei jedem zweiten Dreieck eines Streifens umgekehrt.



Text in Java 3D

Java 3D stellt 2 Möglichkeiten zur Verfügung, Text zu erzeugen: **`Text2D`** (planaren Text, polygonal) und **`Text3D`** (echte 3D-Buchstaben, d.h. durch Extrusion erzeugte, solide Körper). Beide Klassen haben wenig gemein und stehen an ganz unterschiedlichen Stellen der Klassenhierarchie:



2D-Text kann direkt als Knoten in den Szenengraphen eingefügt werden. Benötigt wird **`import java.awt.Font`**.

2D-Text ist defaultmäßig nur von einer Seite sichtbar. Durch Ändern der Polygon-Attribute kann dies geändert werden.

3D-Text fungiert als Geometrie-Knoten zu einem Shape3D-Objekt. Er kann mit Materialeigenschaften versehen, beleuchtet und gerendert werden. Defaultmäßig sind die "Seiten" der Buchstaben (Mantelflächen der Extrusion) nicht selbst-leuchtend, also ohne eingeschaltete Lichtquelle unsichtbar.

Konstruktor für 2D-Text:
```
Text2D(String text, Color3f c, String fontname, int
fontsize, int fontstyle)
```
Methode:
```
void setRectangleScaleFactor(float s)
```

Konstruktoren für 3D-Text:
```
Text3D()
Text3D(Font3D f)
Text3D(Font3D f, String text)
Text3D(Font3D f, String text, Point3f position)
Text3D(Font3D f, String text, Point3f position,
   int alignment, int path)
```
dabei gibt es für **`alignment`** und **`path`** die folgenden Möglichkeiten:

	ALIGN_FIRST (default)	ALIGN_CENTER	ALIGN_LAST
PATH_RIGHT (default)	Text3D	Text3D	Text3D
PATH_LEFT	D3txeT	D3txeT	D3txeT
PATH_DOWN	T e x t	T e x t	T e x t
PATH_UP	t x e T	t x e T	t x e T

Das **`Font3D`**-Objekt greift auf **`java.awt.Font`** zurück:

Konstruktor:
```
Font3D(Font f, FontExtrusion extrudePath)
```

Font-Konstruktor:
```
public Font(String name, int style, int size)
```
`style`: z.B. **`PLAIN, BOLD, ITALIC`**
`size`: Größe in pt

Extrusions-Konstruktoren:
`FontExtrusion()`
`FontExtrusion(java.awt.Shape extrusionShape)`

Default: 0.2 Einheiten Tiefe.
Spezifikation mit **`null`** erzeugt planaren Text.

Zu Methoden und Capabilites von **`Text3D`** und **`Font3D`** siehe Java 3D-Dokumentation bzw. Tutorial.

Beispiel (Code-Fragment):

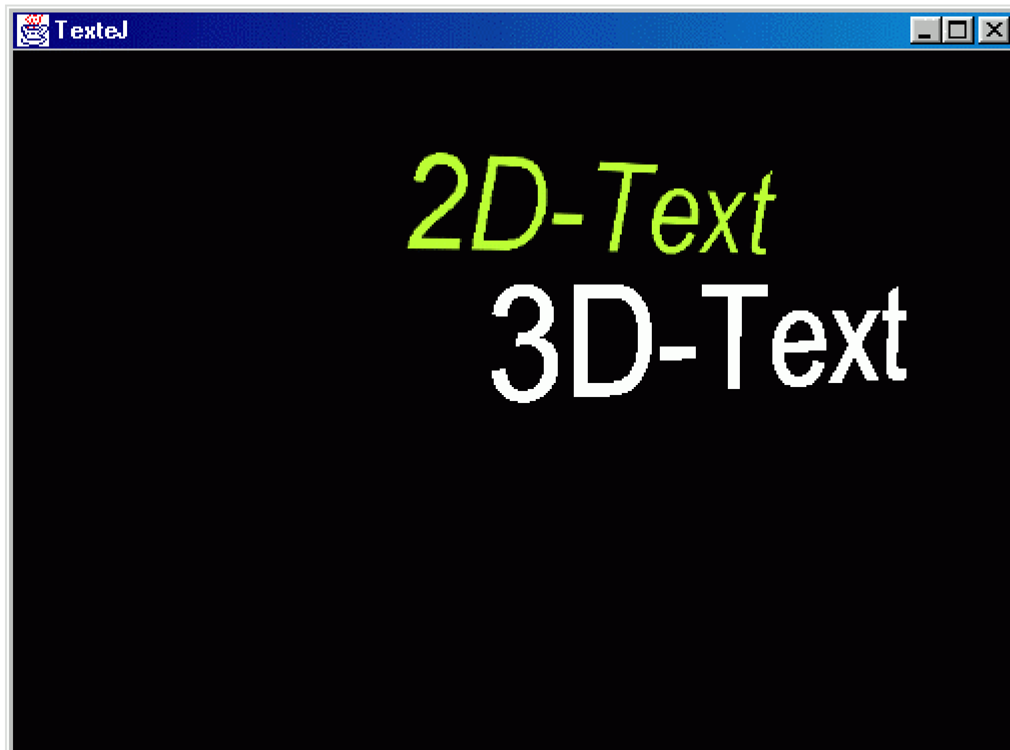
```
import java.awt.Font;
import com.sun.j3d.utils.geometry.*;
// ...
public BranchGroup
   macheSzenengraph(SimpleUniverse u)
   {
   BranchGroup objWurzel = new BranchGroup();

   Text2D text2d = new Text2D("2D-Text",
      new Color3f(0.7f, 1.0f, 0.2f),
      "Helvetica", 80, Font.ITALIC);
   // mache 2D-Text beidseitig lesbar:
   Appearance textapp = text2d.getAppearance();
   PolygonAttributes p_att = new PolygonAttributes();
   p_att.setCullFace(PolygonAttributes.CULL_NONE);
   p_att.setBackFaceNormalFlip(true);
   textapp.setPolygonAttributes(p_att);

   Font3D font3d = new Font3D(new
      Font("Times", Font.PLAIN, 10),
      new FontExtrusion());
   Text3D textGeom = new Text3D(font3d,
      new String("3D-Text"),
      new Point3f(-2.0f, 0.0f, 0.0f));
   Shape3D text3d = new Shape3D(textGeom);

   // verschiebe den 2D-Text:
   Transform3D verschieb = new Transform3D();
   verschieb.set(new Vector3f(-0.3f, 0.3f, 0.0f));
   TransformGroup vtext2d =
      new TransformGroup(verschieb);
   vtext2d.addChild(text2d);
```

```
   // skaliere den 3D-Text:
   Transform3D skal = new Transform3D();
   skal.setScale(0.04);
   TransformGroup stext3d = new TransformGroup(skal);
   stext3d.addChild(text3d);

   // Komposition von 2 Rotationen
   // auf beide Texte angewandt:
   Transform3D drehung = new Transform3D();
   Transform3D drehung2 = new Transform3D();
   drehung.rotX(0.1d);
   drehung2.rotY(0.5d);
   drehung.mul(drehung2);
   TransformGroup objDreh =
      new TransformGroup(drehung);

   objDreh.addChild(vtext2d);
   objDreh.addChild(stext3d);
   objWurzel.addChild(objDreh);

   // verschiebe ViewingPlatform:
   TransformGroup vpVer = null;
   Transform3D zurueck = new Transform3D();
   vpVer =
      u.getViewingPlatform(
        ).getViewPlatformTransform();
   zurueck.set(new Vector3f(0.0f, 0.0f, 3.0f));
   vpVer.setTransform(zurueck);

   return objWurzel;
   } // end macheSzenengraph-Methode

public TexteJ()
   {
   setLayout(new BorderLayout());
    GraphicsConfiguration config =
        SimpleUniverse.getPreferredConfiguration();
   Canvas3D canvas = new Canvas3D(config);
   add("Center", canvas);
   SimpleUniverse u = new SimpleUniverse(canvas);
   u.getViewingPlatform(
      ).setNominalViewingTransform();
   BranchGroup szene = macheSzenengraph(u);
   szene.compile();
   u.addBranchGraph(szene);
```

```
   } // end Konstruktor
```

Beachte: Es wird hier zugleich demonstriert, wie man in einem **`SimpleUniverse`** auf die **`ViewingPlatform`** (Benutzer-standpunkt) zugreift, um den Blickpunkt zu verschieben.

Ergebnis:



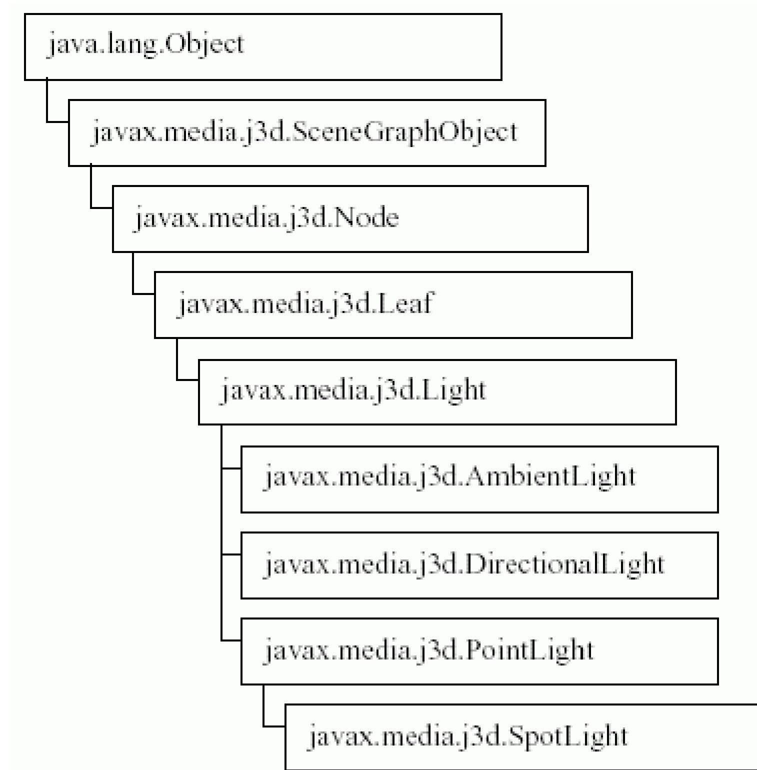
Weitere Java 3D-Klassen, die speziellen Knoten in VRML entsprechen, aber hier nicht näher besprochen werden:

- Background
- Fog
- LOD
- Billboard

Auch die Lichtquellen werden weitgehend analog zu VRML behandelt:

Licht

Klassenhierarchie der Lichtquellen:

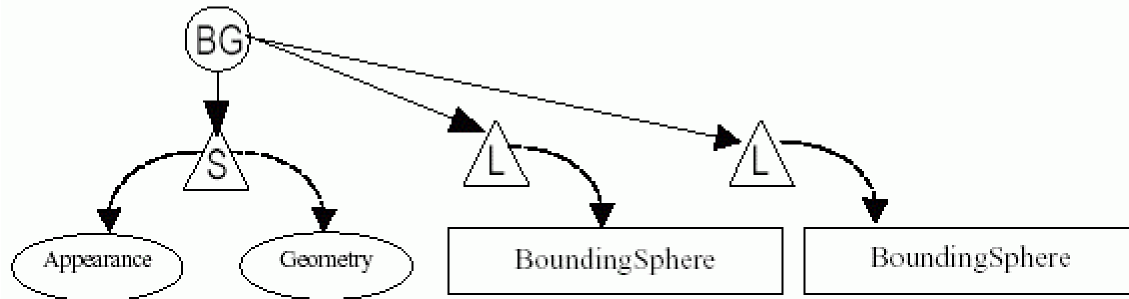


Die Lichtquellen werden in den Szenengraphen analog zu Shape-Knoten eingefügt (sind aber selbst nicht direkt sichtbar, nur durch ihre Wirkung auf Objekte).

Wichtige Unterschiede zu VRML:

- jede Lichtquelle hat *InfluencingBounds*, d.h. eine Wirksamkeitsregion (die defaultmäßig **`null`** ist) – man muss erst ein Bounds-Objekt kreieren, um das Licht wirksam werden zu lassen (die Wirksamkeitsregion bestimmt dann, *welche Objekte* von dem Licht beeinflusst werden – nicht, welche Teile von Objekten!)
- soll das Licht das Aussehen (shading) von Objekten beeinflussen, so muss festgelegt werden, dass diese Objekte mit Normalenvektoren versehen sind!

Szenengraph mit 2 Lichtquellen:



Methoden der (übergeordneten) **`Light`**-Klasse:

`void setColor(Color3f c)`
`void setEnable(boolean state)` // ein/aus
`void setInfluencingBounds(Bounds b)`

Capabilities von Light:

`ALLOW_INFLUENCING_BOUNDS_READ` bzw. **`_WRITE`**
`ALLOW_STATE_READ` bzw. **`_WRITE`**
`ALLOW_COLOR_READ` bzw. **`_WRITE`**

Konstruktoren:
`AmbientLight()` // default-Farbe weiß
`AmbientLight(Color3f c)`
`AmbientLight(boolean on, Color3f c)`
`DirectionalLight()` // default-Richtung (0; 0; –1)
`DirectionalLight(Color3f c, Vector3f dir)`
`DirectionalLight(boolean on, Color3f c, Vector3f dir)`
`PointLight()` // default-Position 0
`PointLight(Color3f c, Point3f pos, Point3f att)`
`att` sind die 3 Werte *a*, *b*, *c* in der Abschwächungs- (*attenuation*-) Formel $(a + bd + cd^2)^{-1}$, mit *d* = Distanz. Default: (1; 0; 0), also keine Abschwächung.

`PointLight(boolean on, etc....)`
`SpotLight()`
`SpotLight(boolean on, Color3f c, Point3f pos,`

```
  Point3f att, Vector3f dir, float spreadAngle,
  float concentration)
```
zu Methoden und Capabilities der Lichtquellen-Klassen siehe Dok./Tutorial.

Beispiel:
2 rote Kugeln werden von ambientem Licht und gerichtetem Licht beschienen. Die InfluencingBounds sind auf maximale Ausdehnung geschaltet. Man beachte das automatische Gouraud-Shading der Kugeln und den fehlenden Schatten. (Code-Fragment; der Rest ist analog zum FarbKegel-Beispiel:)

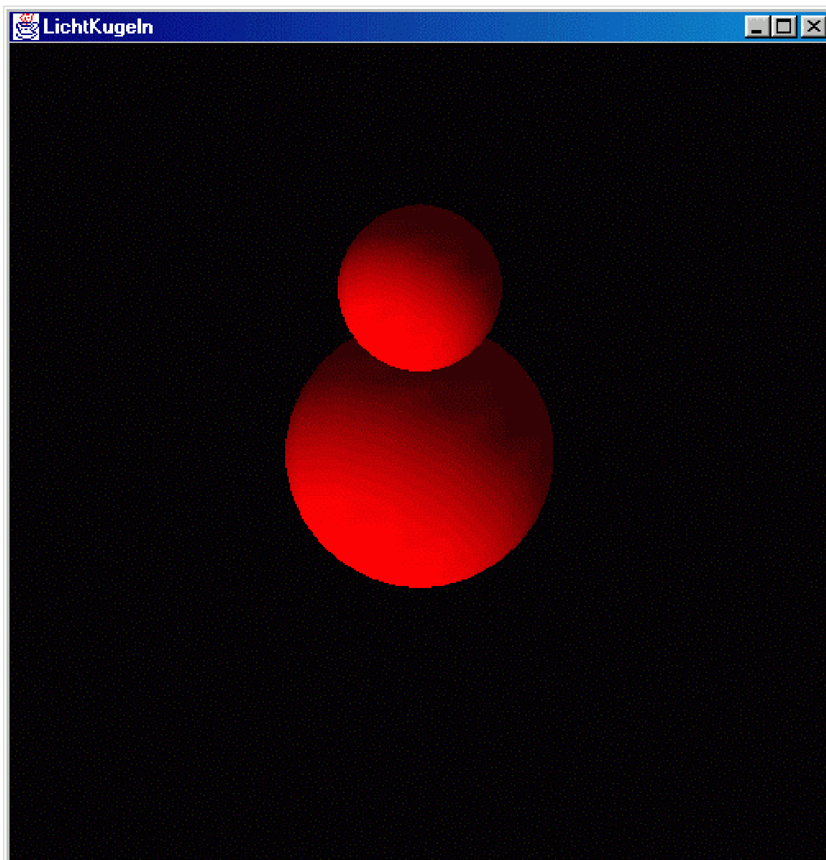
```
import com.sun.j3d.utils.geometry.Sphere;
// ....
// selbstgeschriebene Klasse:
public class MeineLichtKugeln extends Shape3D
   {
   private BranchGroup meineBG;

   public MeineLichtKugeln()
      {
      meineBG = new BranchGroup();

      // 1. Translation
      Transform3D trans1 = new Transform3D();
      trans1.set(new Vector3f(0.0f, 0.0f, -1.3f));
      TransformGroup tg1 =
         new TransformGroup(trans1);

      // 2. Translation
      Transform3D trans2 = new Transform3D();
      trans2.set(new Vector3f(0.0f, 0.4f, 0.0f));
      TransformGroup tg2 =
         new TransformGroup(trans2);

      // Kugel 1
      Sphere ku1 =
         new Sphere(0.5f, Sphere.GENERATE_NORMALS,
         macheMaterial());
      tg1.addChild(ku1);
      meineBG.addChild(tg1);

      // Kugel 2
```

```
   Sphere ku2 =
      new Sphere(0.2f, Sphere.GENERATE_NORMALS,
      macheMaterial());
   tg2.addChild(ku2);
   meineBG.addChild(tg2);

   // gerichtetes Licht
   DirectionalLight d_licht =
      new DirectionalLight();
   d_licht.setInfluencingBounds(new
      BoundingSphere(new
         Point3d(0.0d, 0.0d, 0.0d),
         Double.MAX_VALUE));
   d_licht.setColor(new
      Color3f(1.0f, 0.0f, 0.0f));
   Vector3f dir =
      new Vector3f(1.0f, 2.0f, -1.0f);
   dir.normalize();
   d_licht.setDirection(dir);
   meineBG.addChild(d_licht);

   // ambientes Licht
   AmbientLight a_licht = new AmbientLight();
   a_licht.setInfluencingBounds(new
      BoundingSphere(new
         Point3d(0.0d, 0.0d, 0.0d),
         Double.MAX_VALUE));
   a_licht.setColor(new
      Color3f(1.0f, 0.0f, 0.0f));
   meineBG.addChild(a_licht);

   meineBG.compile();
   }

Appearance macheMaterial()
   {
   Appearance a = new Appearance();
   Material mat = new Material();
   mat.setShininess(50.0f);
   mat.setDiffuseColor(new
      Color3f(1.0f, 0.0f, 0.0f));
   mat.setSpecularColor(new
      Color3f(0.0f, 0.0f, 0.0f));
   a.setMaterial(mat);
   return a;
```

```
        }

    public BranchGroup getBG()
        {
        return meineBG;
        }

    } // end class MeineLichtKugeln

public BranchGroup macheSzenengraph()
    {
    BranchGroup objWurzel = new
        MeineLichtKugeln().getBG();
    objWurzel.compile();
    return objWurzel;
    } // end macheSzenengraph
```

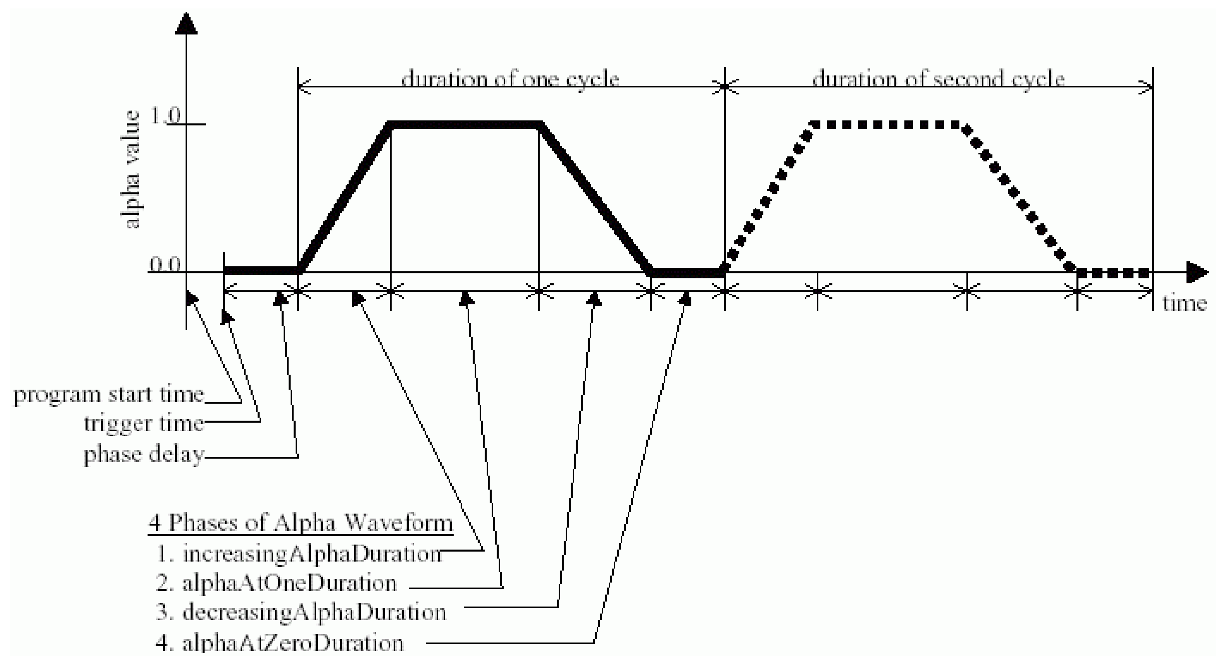
Man beachte die Verwendung des Kontrollparameters (des 2. Parameters) im `Sphere`-Konstruktor zur Erzwingung der Normalenvektoren-Erzeugung.

Ergebnis:

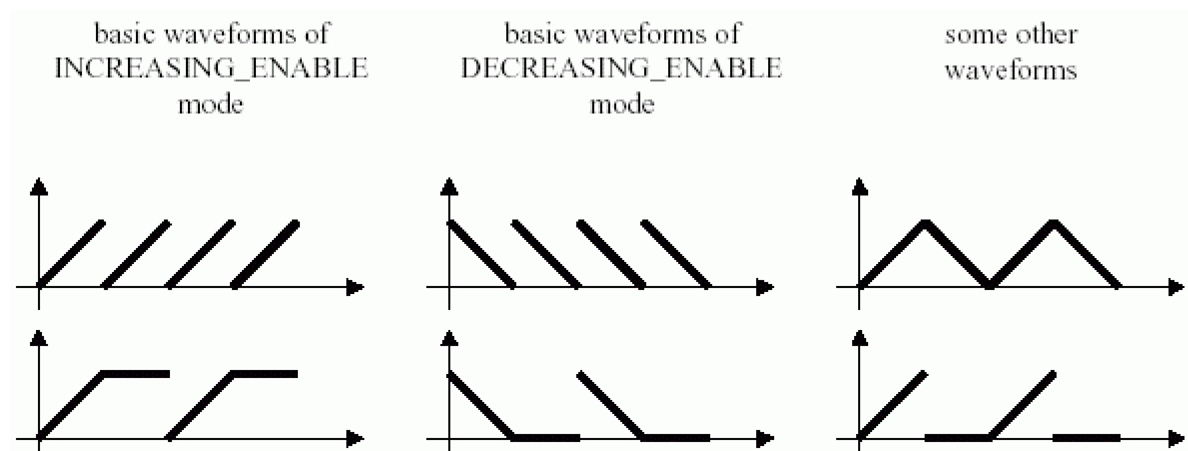


Animation

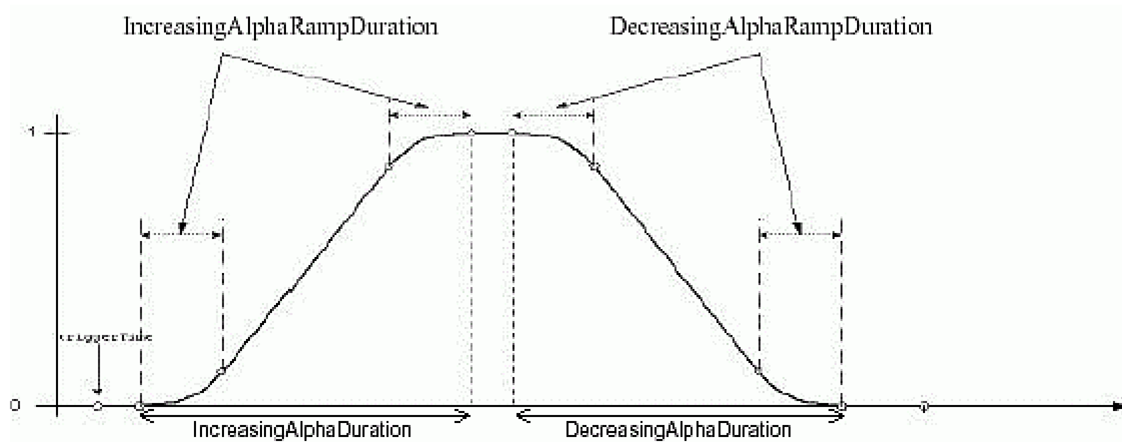
Animation wird wie in VRML durch Zeitgeber und Interpolatoren gesteuert.
Die Rolle des Zeitgebers (TimeSensor-Knoten in VRML) übernimmt in Java 3D die (mächtigere) Klasse **`Alpha`**.
Sie erzeugt stückweise lineare Signale der folgenden Form:



Insbesondere sind folgende Wellenformen möglich, von denen einige durch spezielle Kontroll-Flags einfacher als über die numerischen Parameter angesprochen werden können:



Die "Knicks" des Funktionsverlaufs können durch zusätzliche Parameter gerundet werden:



Wichtig ist der **`loopCount`**-Parameter:
Die Welle kann keinmal, einmal, *n* mal oder unendlich oft wiederholt werden. Ein **`loopCount`** von –1 bedeutet unendliche Wiederholung.

Alpha Constructor Summary

extends: `Object`

The alpha class converts a time value into an alpha value (a value in the range 0 to 1, inclusive). The Alpha object is effectively a function of time that generates values in the range [0,1]. A common use of the Alpha provides alpha values for Interpolator behaviors. The characteristics of the Alpha object are determined by user-definable parameters. Refer to Figure 5-2, Figure 5-6, and the text accompanying these figures for more information.

`Alpha()`
Constructs an Alpha object with mode = INCREASING_ENABLE, loopCount = -1, increasingAlphaDuration = 1000, all other parameters = 0, except StartTime. StartTime is set as the start time of the program.

`Alpha(int loopCount, long increasingAlphaDuration)`
This constructor takes only the loopCount and increasingAlphaDuration as parameters, sets the mode to INCREASING_ENABLE and assigns 0 to all of the other parameters (except StartTime).

```
Alpha(int loopCount, long triggerTime, long phaseDelayDuration,
      long increasingAlphaDuration, long increasingAlphaRampDuration,
      long alphaAtOneDuration)
```
Constructs a new Alpha object and sets the mode to INCREASING_ENABLE.

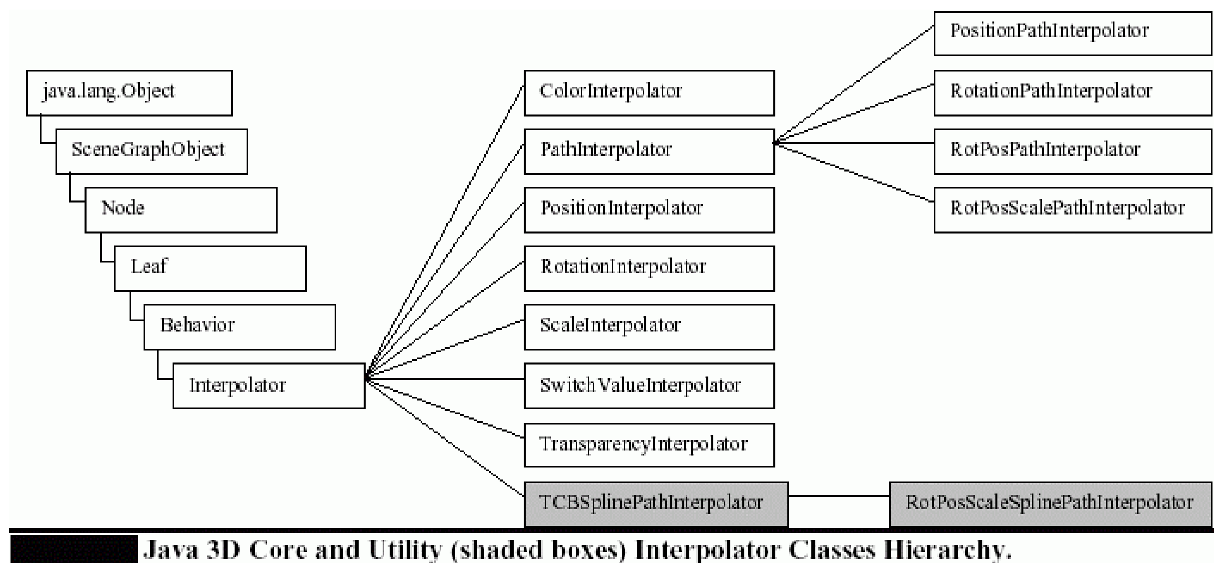
```
Alpha(int loopCount, int mode, long triggerTime, long phaseDelayDuration,
      long increasingAlphaDuration, long increasingAlphaRampDuration,
      long alphaAtOneDuration, long decreasingAlphaDuration,
      long decreasingAlphaRampDuration, long alphaAtZeroDuration)
```
This constructor takes all of the Alpha user-definable parameters.

Grundsätzliche Vorgehensweise bei Erstellung eines animierten Objekts:

1. erzeuge das Zielobjekt mit den entsprechenden Parametern und capabilities (...WRITE!)
2. erzeuge das Alpha-Objekt mit den gewünschten Parametern
3. erzeuge ein Interpolator-Objekt, welches das Alpha-Objekt und das Zielobjekt referenziert
4. versehe das Interpolator-Objekt mit SchedulingBounds
5. füge das Interpolator-Objekt in den Szenengraphen ein

Interpolatoren

Interpolatoren erben von einer abstrakten Oberklasse "`Behavior`". Wir finden die bekannten Möglichkeiten aus VRML wieder, nebst Erweiterungen:



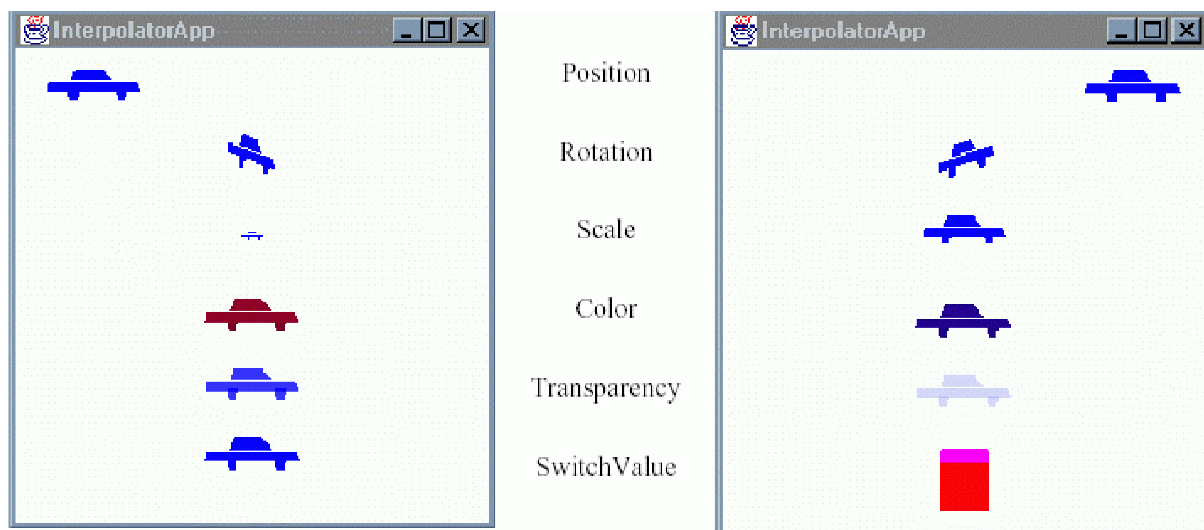
Java 3D Core and Utility (shaded boxes) Interpolator Classes Hierarchy.

Neu gegenüber VRML ist die Notwendigkeit, für alle Interpolatoren Wirkungsbereiche (SchedulingBounds) und für alle beeinflussten Objekte Capabilities festzulegen!

Tabelle der wichtigsten Interpolator-Klassen:

Interpolator class	**used to**	**target object type**
`ColorInterpolator`	change the diffuse color of an object(s)	Material
`PathInterpolator`[10]	*abstract class*	*TransformGroup*
`PositionInterpolator`	change the position of an object(s)	TransformGroup
`RotationInterpolator`	change the rotation (orientation) of an object(s)	TransformGroup
`ScaleInterpolator`	change the size of an object(s)	TransformGroup
`SwitchValueInterpolator`	choose one of (switch) among a collection of objects	Switch
`TransparencyInterpolator`	change the transparency of an object(s)	TransparencyAttributes

Ihre Wirkung, veranschaulicht:

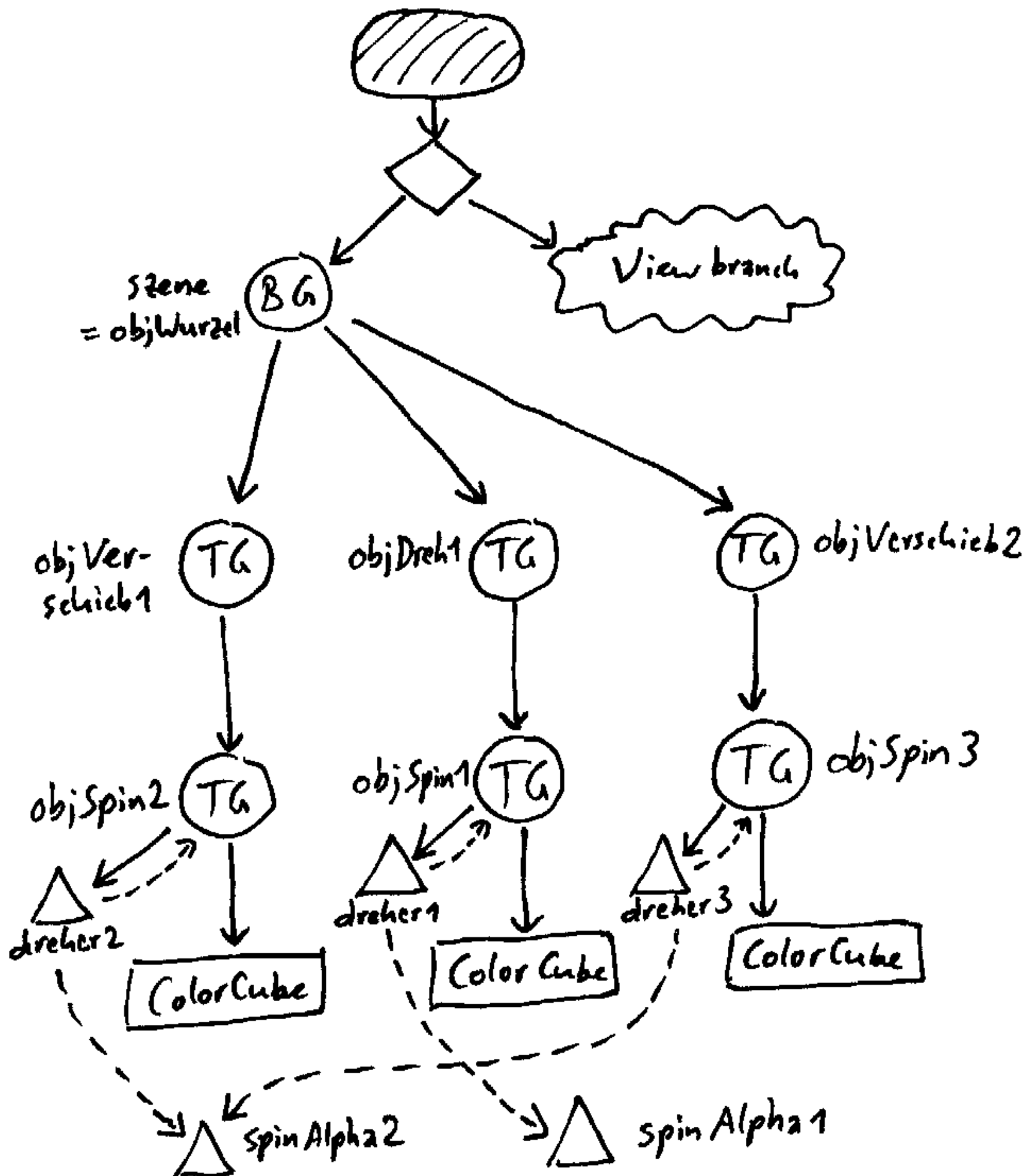


Für eine Auflistung der möglichen Konstruktoren und Methoden der einzelnen Interpolator-Klassen wird auf die Dokumentation bzw. auf das Tutorial verwiesen.

Wichtig: Die Interpolatoren werden in den Szenengraphen eingefügt, und zwar in der Regel als Kindknoten der (Transform-) Gruppenknoten, deren Kinder sie beeinflussen sollen. Sie referenzieren ihren Mutter-Knoten *und* das Alpha-Objekt.

Beispiel:
Drei Farbwürfel sollen sich kontinuierlich drehen: der mittlere schneller als die äußeren. Für beide äußeren Würfel wird dasselbe Alpha-Objekt verwendet. Vor dem Ingangsetzen der Rotation werden die Würfel in verschiedene Lagen gebracht.

Zugehöriger Szenengraph:

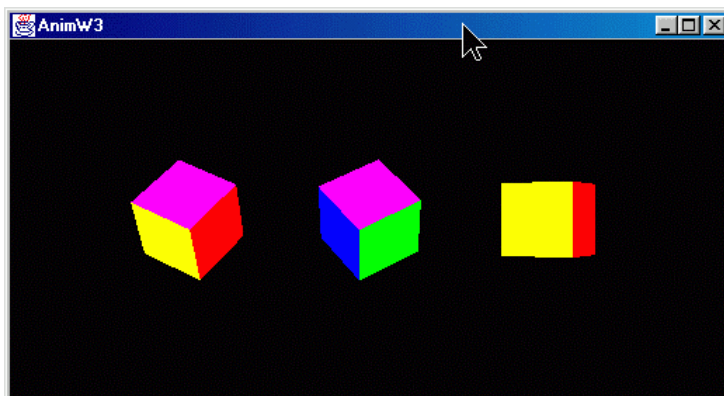


Codefragment (Rest wie im ersten Farbwürfel-Beispiel):

```
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 objDreh1 =
      new TransformGroup(drehung);

   Transform3D verschiebung1 = new Transform3D();
   // 2 Einheiten nach rechts
   Vector3f vektor1 = new Vector3f(0.5f, 0.0f, 0.0f);
   verschiebung1.set(vektor1);
   TransformGroup objVerschieb1 =
      new TransformGroup(verschiebung1);

   Transform3D drehschieb = new Transform3D();
   Transform3D verschiebung2 = new Transform3D();
   drehschieb.rotX(Math.PI / 4.0d);
   Vector3f vektor2 =
      new Vector3f(-0.5f, 0.0f, 0.0f);
   verschiebung2.set(vektor2);
   drehschieb.mul(verschiebung2);
   TransformGroup objVerschieb2 =
      new TransformGroup(drehschieb);

   // fuer Animation:
   TransformGroup objSpin1 = new TransformGroup();
   objSpin1.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);
   TransformGroup objSpin2 = new TransformGroup();
   objSpin2.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);
   TransformGroup objSpin3 = new TransformGroup();
   objSpin3.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);
```

```
objSpin1.addChild(new ColorCube(0.1));
objDreh1.addChild(objSpin1);
objWurzel.addChild(objDreh1);
objSpin2.addChild(new ColorCube(0.1));
objVerschieb1.addChild(objSpin2);
objWurzel.addChild(objVerschieb1);
objSpin3.addChild(new ColorCube(0.1));
objVerschieb2.addChild(objSpin3);
objWurzel.addChild(objVerschieb2);

Alpha spinAlpha1 = new Alpha(-1, 4000);
   // -1: Endlosschleife, 4000: 4 sec
RotationInterpolator dreher1 =
   new RotationInterpolator(spinAlpha1, objSpin1);

Alpha spinAlpha2 = new Alpha(-1, 12000);
RotationInterpolator dreher2 =
   new RotationInterpolator(spinAlpha2, objSpin2);

// mit selbem Alpha:
RotationInterpolator dreher3 =
   new RotationInterpolator(spinAlpha2, objSpin3);

// Aktivitaetsbereich:
BoundingSphere zone = new BoundingSphere();
dreher1.setSchedulingBounds(zone);
dreher2.setSchedulingBounds(zone);
dreher3.setSchedulingBounds(zone);
objSpin1.addChild(dreher1);
objSpin2.addChild(dreher2);
objSpin3.addChild(dreher3);

return objWurzel;
} // end macheSzenengraph-Methode
```

Ergebnis (Snapshot):



Bewegen von Objekten mit der Maus

Es gibt eine Reihe von Behavior-Subklassen, die mit der Maus assoziiert sind.
Ohne auf Details (Konstruktor-Varianten, Methoden...) einzugehen, erwähnen wir hier nur die wichtigsten:

Klasse	*ausgelöste Aktion*	*Maustaste*
MouseRotate	rotiert Objekt an seinem Platz	linke
MouseTranslate	verschiebt Objekt in Ebene parallel zur Bildebene	rechte
MouseZoom	verschiebt Objekt in Ebene senkrecht zur Bildebene	mittlere

Objekte der 3 Klassen können simultan für dasselbe Zielobjekt (dieselbe **`TransformGroup`**) aktiv sein.

Grundsätzliche Vorgehensweise:

1. Versehe die Ziel-**`TransformGroup`** mit READ- und WRITE-Capabilities
2. erzeuge ein **`MouseBehavior`**-Objekt (z.B. eines der obigen Klassen)
3. spezifiziere die Ziel-**`TransformGroup`**
4. versehe das **`MouseBehavior`**-Objekt mit SchedulingBounds
5. füge das **`MouseBehavior`**-Objekt in den Szenengraphen ein

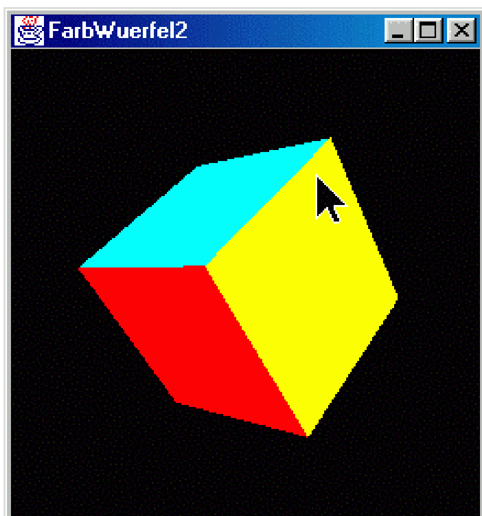
Für die Position im Szenengraphen gilt dasselbe wie oben für die Interpolator-Objekte.

Im folgenden Beispiel kann der Benutzer den Farbwürfel lediglich interaktiv drehen (nicht verschieben oder heranschieben). Verschieben und Zoomen geht analog.

Beispiel (Codefragment, Rest wie im ersten Farbwürfel-Beispiel):

```
import com.sun.j3d.utils.behaviors.mouse.*;
//...
public BranchGroup macheSzenengraph()
   {
   BranchGroup objWurzel = new BranchGroup();

   // Verwendung von MouseRotate:
   TransformGroup objDreh = new TransformGroup();
   objDreh.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);
   objDreh.setCapability(
      TransformGroup.ALLOW_TRANSFORM_READ);

   MouseRotate mausDreh = new MouseRotate();
   mausDreh.setTransformGroup(objDreh);
   mausDreh.setSchedulingBounds(new
      BoundingSphere());

   objWurzel.addChild(objDreh);
   objWurzel.addChild(mausDreh);
   objDreh.addChild(new ColorCube(0.4));

   objWurzel.compile();
   return objWurzel;
   } // end macheSzenengraph-Methode
```

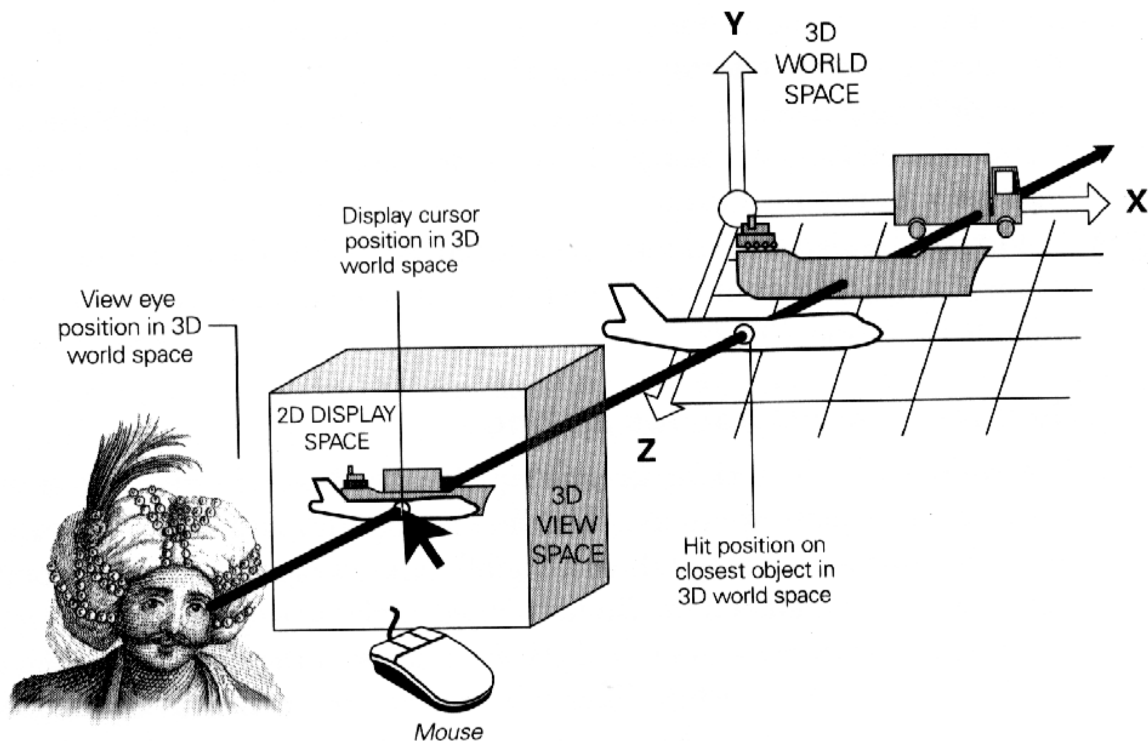
Das Ergebnis ist von den Aktionen des Benutzers abhängig:



Erweiterung: *Picking*

= Auswahl von sichtbaren Objekten mit der Maus und deren Beeinflussung

Abbildung von 2D in 3D,
ggf. nicht eindeutig



(aus J. Barrilleaux: 3D User Interfaces with Java 3D)

⇒ der Vorgang ist teuer ⇒ um ihn zuzulassen, muss eine spezielle Capability bei den in Frage kommenden Zielobjekten gesetzt werden:
`ENABLE_PICK_REPORTING`

Neben einer Reihe weiterer Picking-Klassen gibt es Klassen, die zu den oben genannten MouseBehaviors korrespondieren:

- **`PickRotateBehavior`**
- **`PickTranslateBehavior`**
- **`PickZoomBehavior`**

(zu Angaben über Konstruktoren und Methoden siehe Dok. bzw. Tutorial)

Beispiel:
Einer von zwei Farbwürfeln kann durch Anklicken ausgewählt und dann gedreht werden
(Codefragment, Rest wie beim ersten FarbWürfel-Beispiel)

```
import com.sun.j3d.utils.behaviors.picking.*;
// ...
public class Pick1 extends Applet
   {
   // macheSzenengraph benoetigt canvas:
   public BranchGroup
      macheSzenengraph(Canvas3D canvas)
      {
      BranchGroup objWurzel = new BranchGroup();

      // Initialisierungen:
      TransformGroup objDreh = null;
      PickRotateBehavior pickDreh = null;
      Transform3D verschieb =
         new Transform3D();
      BoundingSphere zone =
         new BoundingSphere();

      // Generierung der ersten Objekte:
      verschieb.setTranslation(
         new Vector3f(-0.6f, 0.0f, -0.6f));
      objDreh = new TransformGroup(verschieb);
      objDreh.setCapability(
         TransformGroup.ALLOW_TRANSFORM_WRITE);
      objDreh.setCapability(
         TransformGroup.ALLOW_TRANSFORM_READ);
      objDreh.setCapability(
         TransformGroup.ENABLE_PICK_REPORTING);

      objWurzel.addChild(objDreh);
      objDreh.addChild(new ColorCube(0.4));

      pickDreh = new
         PickRotateBehavior(objWurzel, canvas, zone);
      objWurzel.addChild(pickDreh);

      // Hinzufuegen eines zweiten ColorCube:

      verschieb.setTranslation(
```

```
      new Vector3f(0.6f, 0.0f, -0.6f));
   objDreh = new TransformGroup(verschieb);
   objDreh.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);
   objDreh.setCapability(
      TransformGroup.ALLOW_TRANSFORM_READ);
   objDreh.setCapability(
      TransformGroup.ENABLE_PICK_REPORTING);

   objWurzel.addChild(objDreh);
   objDreh.addChild(new ColorCube(0.4));

   objWurzel.compile();
   return objWurzel;
   } // end macheSzenengraph-Methode

public Pick1()
   {
   setLayout(new BorderLayout());
   GraphicsConfiguration config =
     SimpleUniverse.getPreferredConfiguration();
   Canvas3D canvas = new Canvas3D(config);
   add("Center", canvas);
   BranchGroup szene = macheSzenengraph(canvas);
   szene.compile();
   SimpleUniverse u =
      new SimpleUniverse(canvas);
   u.getViewingPlatform(
      ).setNominalViewingTransform();
   u.addBranchGraph(szene);
   } // end Konstruktor
// ...
}
```

Beachte: Hier wird in der Methode, die den Szenengraphen erstellt, zusätzlich zu den bisherigen Beispielen das **Canvas3D**-Objekt aus dem **SimpleUniverse** benötigt, da dieses im Konstruktor-Aufruf von **PickRotateBehavior** auftritt.