Computergrafik - Übungen
W. Kurth, T.Mangoldt

Java 3D
Einstiegs-Tutorial
Teil 3

Navigation mit der Maus

Java 3D eröffnet mehrere Möglichkeiten, sich mittels Maus-Eingaben im virtuellen Raum zu bewegen (vgl. Walsh & Gehringer 2002, S. 214 f.).

Am einfachsten ist es, die 3 MouseBehavior-Funktionen aus dem vorletzten Beispiel (Bewegen von Objekten mit der Maus) auf eine TransformGroup anzuwenden, die die Viewing-Platform, also den Betrachterstandpunkt, bewegt.
(Zum Zugriff auf die ViewingPlatform eines SimpleUniverse-Objekts siehe das Beispiel zu "Text in Java 3D".)

Beachte: Damit man die intuitiven Bewegungsrichtungen erhält, muss man jetzt im Konstruktor-Aufruf die Richtung mit dem Flag **`MouseBehavior.INVERT_INPUT`** umkehren.

Das folgende Beispielprogramm ermöglicht Bewegungen mit allen drei Maustasten (Drehung, seitliches Gehen, Heran-gehen).
Wenn die mittlere Maustaste fehlt oder nicht richtig funktioniert, kann sie durch "Alt-Taste + linke Maustaste" ersetzt werden. Am Programm muss nichts geändert werden.

Das Codefragment ist mit dem üblichen Rahmen aus dem FarbWuerfel-Beispiel zu ergänzen.
Statt des hier verwendeten Farbwürfels können natürlich beliebige andere Objekte in den Szenengraphen eingefügt werden:

```
public BranchGroup
   macheSzenengraph(SimpleUniverse su)
{
BranchGroup objWurzel = new BranchGroup();

// Initialisierungen; Zugriff auf ViewingPlatform:
TransformGroup sichtTG =
   su.getViewingPlatform(
      ).getViewPlatformTransform();
BoundingSphere mausZone =
   new BoundingSphere(new Point3d(),
      Float.MAX_VALUE);

// Generierung der sichtbaren Objekte:
objWurzel.addChild(new ColorCube(0.4));

// Instanziierungen der MouseBehavior-Objekte
MouseRotate meineDrehMaus =
   new MouseRotate(MouseBehavior.INVERT_INPUT);
meineDrehMaus.setTransformGroup(sichtTG);
meineDrehMaus.setSchedulingBounds(mausZone);
objWurzel.addChild(meineDrehMaus);

MouseTranslate meineGehMaus =
   new MouseTranslate(MouseBehavior.INVERT_INPUT);
meineGehMaus.setTransformGroup(sichtTG);
meineGehMaus.setSchedulingBounds(mausZone);
objWurzel.addChild(meineGehMaus);

MouseZoom meineRanMaus =
   new MouseZoom(MouseBehavior.INVERT_INPUT);
meineRanMaus.setTransformGroup(sichtTG);
meineRanMaus.setSchedulingBounds(mausZone);
objWurzel.addChild(meineRanMaus);

// Optimierungen am Szenengraphen und Abschluss:
objWurzel.compile();
return objWurzel;
} // end macheSzenengraph-Methode
```

Es gibt auch eine Klasse "**KeyNavigatorBehavior**" zur Navigation mit der Tastatur; siehe Java 3D API Tutorial, Chapter 4, p. 4-28.

Definition eigener Behavior-Objekte zur Event-Verarbeitung

Java 3D ermöglicht in Kombination mit den Event-Verarbeitungsmethoden des AWT-Packages die Reaktion auf verschiedenste sog. Wakeup-Bedingungen mit beliebigen Reaktionsweisen. Zur AWT wird auf allgemeine Java-Kurse bzw. -Literatur verwiesen.
Wir zeigen exemplarisch die Definition einer eigenen Behavior-Klasse, die einen Farbwürfel durch Drücken einer beliebigen Taste des Keyboards um einen kleinen Betrag rotieren lässt.

Dabei ist zu beachten:

- Die selbstdefinierte Behavior-Klasse muss die Java 3D-Klasse **`Behavior`** erweitern
- sie muss einen Konstruktor, eine **`initialize`**-Methode und eine **`processStimulus`**-Methode enthalten
- letztere enthält das, was bei Vorliegen des Stimulus (hier des Tastendrucks) passieren soll
- in **`initialize`** muss der Trigger gesetzt werden; dies geschieht über ein sog. Wakeup-Event, wovon Java 3D eine größere Zahl von Varianten bereitstellt (siehe Tabelle nach dem Beispiel).
- Wenn das Programm in der Lage sein soll, mehrmals zu reagieren, muss der Trigger am Schluss von **`processStimulus`** erneut gesetzt werden.
- Auch die eigene Behavior-Klasse muss nach dem Instanziieren mit SchedulingBounds versehen werden.
- Das Behavior-Objekt muss in den Szenengraphen eingehängt werden, sonst wird es nicht aktiviert.

Es werden das Package **`java.awt.event.*`** und die Klasse **`java.util.Enumeration`** benötigt.

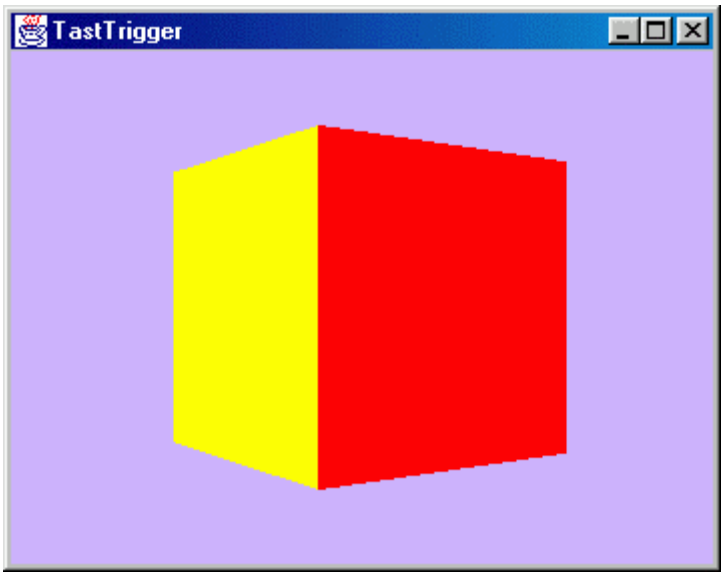
Wir zeigen hier das vollständige Programm, welches auf vielfältige Weise variiert werden kann.
Zur optischen Verbesserung wurde hier noch ein Hintergrund (Background-Objekt) eingefügt.

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import java.util.Enumeration;
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 TastTrigger extends Applet
   {
   public class MeinBehavior extends Behavior
      {
      private TransformGroup zielTG;
      private Transform3D drehung =
         new Transform3D();
      private double winkel = 0.0;
      // Konstruktor-Aufruf soll Ziel-TG enthalten:
      MeinBehavior(TransformGroup ztg)
         {
         this.zielTG = ztg;
         }

      // initialize-Methode (obligatorisch):
      public void initialize()
         {
         this.wakeupOn(new
            WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
         // Wakeup auf Tastendruck
         }

      // die folg. Methode wird ausgefuehrt, wenn
      // der Stimulus (Tastendruck) eintritt;
      // obligatorisch:
      public void processStimulus(Enumeration krit)
         {
         winkel += 0.1;
         drehung.rotY(winkel);
         zielTG.setTransform(drehung);
         this.wakeupOn(new
            WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
         }
      }  // end of class MeinBehavior
```

```
public BranchGroup macheSzenengraph()
   {
   BranchGroup objWurzel = new BranchGroup();
   BoundingSphere riesenkugel =
      new BoundingSphere(
         new Point3d(0.0d, 0.0d, 0.0d),
         Double.MAX_VALUE);

   // konkrete Ziel-TransformGroup:
   TransformGroup objDreh = new TransformGroup();
   objDreh.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);
   // sichtbares Objekt:
   objDreh.addChild(new ColorCube(0.4));
   objWurzel.addChild(objDreh);

   MeinBehavior meinDrehTrick =
      new MeinBehavior(objDreh);
      // Uebergabe der Ziel-TG
   meinDrehTrick.setSchedulingBounds(riesenkugel);
   objWurzel.addChild(meinDrehTrick);

   // Hintergrund:
   Background hg = new Background();
   hg.setColor(new Color3f(0.8f, 0.7f, 1.0f));
   hg.setApplicationBounds(riesenkugel);
   objWurzel.addChild(hg);

   // Optimierungen am Szenengraphen
   // und Abschluss:
   objWurzel.compile();
   return objWurzel;
   } // end macheSzenengraph-Methode

public TastTrigger()
   {
   setLayout(new BorderLayout());
   Canvas3D canvas = new Canvas3D(null);
   add("Center", canvas);
   SimpleUniverse u = new SimpleUniverse(canvas);
   u.getViewingPlatform(
      ).setNominalViewingTransform();
   BranchGroup szene = macheSzenengraph();
   szene.compile();
```

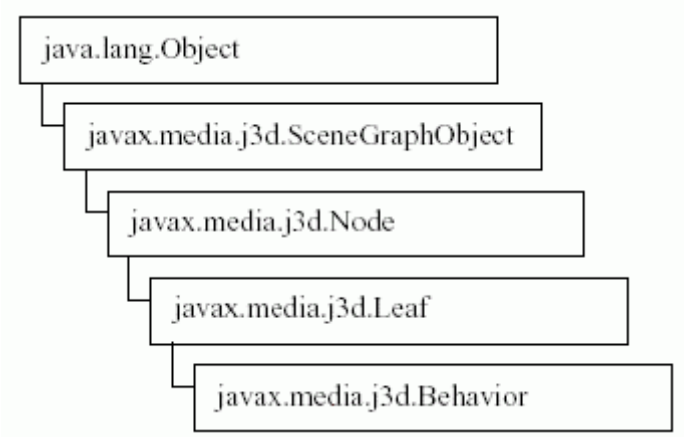
```
        u.addBranchGraph(szene);
        } // end Konstruktor

    public static void main(String[] args)
        {
        Frame frame =
            new MainFrame(new TastTrigger(), 350, 256);
        }
    } // end class TastTrigger
```

Ergebnis (abhängig von Tastatureingabe durch den Benutzer):

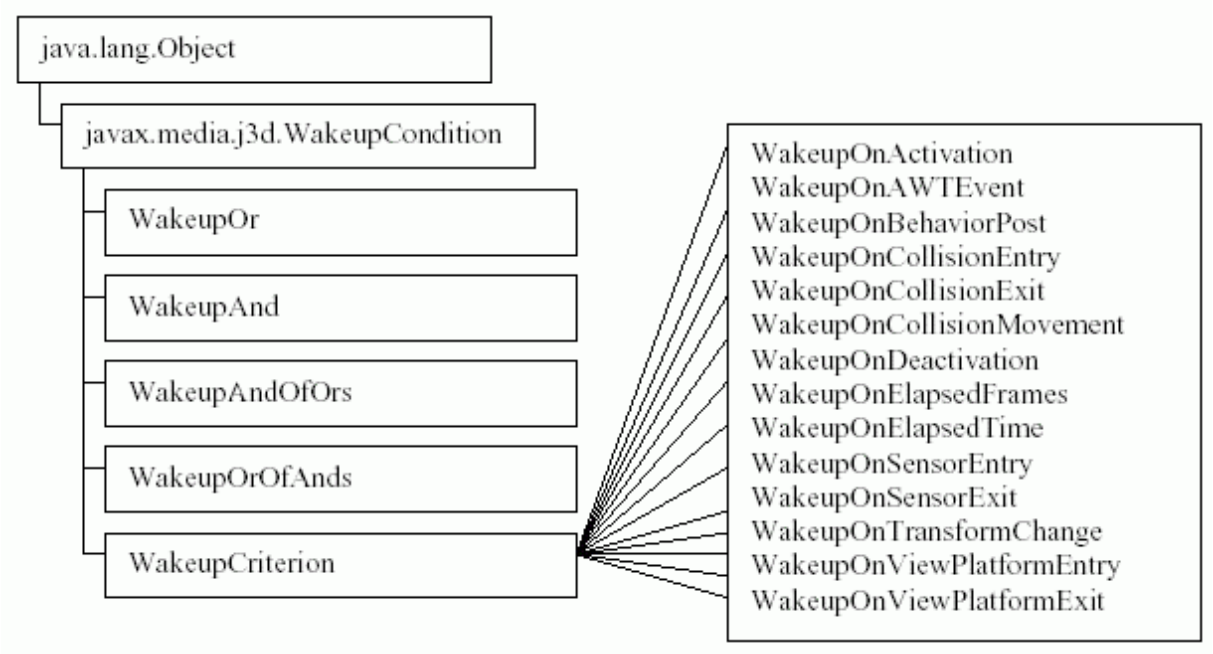


Wenn man den Trigger **`KeyEvent.KEY_PRESSED`** in **`initialize`** und **`processStimulus`** beispielsweise durch **`MouseEvent.MOUSE_PRESSED`** ersetzt, erfolgt die Rotation bei jedem Mausklick. Weitere Events, deren Verwendung sich oft anbietet, sind **`MOUSE_DRAGGED`** und **`MOUSE_MOVED`**.

Einordnung der **`Behavior`**-Klasse (und damit auch der selbstgeschriebenen Erweiterung) in die Klassenhierarchie:



Einordnung der Trigger-Klassen (Wakeup conditions):



Zu den zulässigen Konstruktor-Aufrufen, Methoden und vordefinierten Flags wird auf das Java 3D API Tutorial (Chapter 4) verwiesen.

Wir geben hier noch die Tabelle der 14 speziellen **`WakeupCriterion`**-Klassen wieder, von denen ja **`WakeupOnAWTEvent`** bereits im Beispiel benutzt wurde:

Wakeup Criterion	Trigger
WakeupOnActivation	on first detection of a ViewPlatform's activation volume intersecting with this object's scheduling region.
WakeupOnAWTEvent	when a specific AWT event occurs
WakeupOnBehaviorPost	when a specific behavior object posts a specific event
WakeupOnCollisionEntry	on the first detection of the specified object colliding with any other object in the scene graph
WakeupOnCollisionExit	when the specified object no longer collides with any other object in the scene graph
WakeupOnCollisionMovement	when the specified object moves while in collision with any other object in the scene graph
WakeupOnDeactivation	when a ViewPlatform's activation volume no longer intersects with this object's scheduling region
WakeupOnElapsedFrames	when a specific number of frames have elapsed
WakeupOnElapsedTime	when a specific number of milliseconds have elapsed
WakeupOnSensorEntry	on first detection of any sensor intersecting the specified boundary
WakeupOnSensorExit	when a sensor previously intersecting the specified boundary no longer intersects the specified boundary
WakeupOnTransformChange	when the transform within a specified TransformGroup changes
WakeupOnViewPlatformEntry	on first detection of a ViewPlatform activation volume intersecting with the specified boundary
WakeupOnViewPlatformExit	when a View activation volume no longer intersects the specified boundary

Interpolation von Geometrie: Das Morph-Objekt

Die bisher für die Animation behandelten Interpolatoren können nur einzelne Größen, Positionen etc. verändern. Zur Animation kompletter **`GeometryArray`**-Objekte (z.B. **`LineArray`**, **`TriangleStripArray`** etc.) gibt es die spezielle Klasse **`Morph`**.
Sie ermöglicht (in Kombination mit einem **`Alpha`**-Objekt) die lineare Interpolation zwischen "Frames" (Einzel-Geometrien), die als Array von **`GeometryArray`**-Objekten an **`Morph`** übergeben werden.
Das Beispiel, in dem ein einfaches LineStripSegment aus 4 Linien animiert wird (durch Interpolation zwischen 4 Standbildern), zeigt die nötigen Konstruktor-Aufrufe und

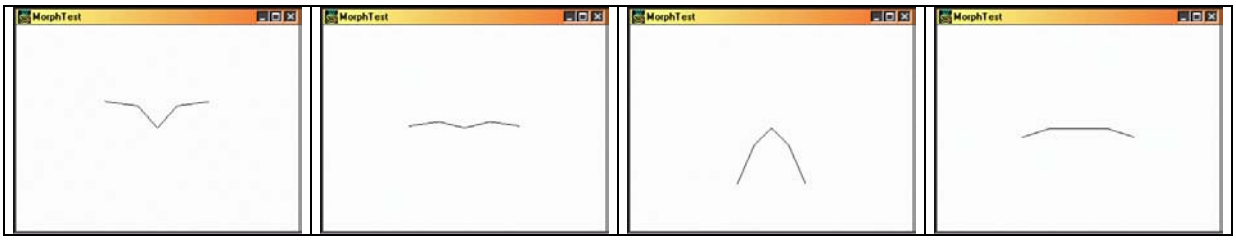
Initialisierungen. Es wird ein Code-Fragment gezeigt; der Rest des Programms entspricht wieder dem Farbwürfel-Beispiel:

```
import java.awt.Frame;
import java.awt.event.*;
import java.util.Enumeration;
import com.sun.j3d.utils.geometry.*;
// ...

public GeometryArray macheFrame0()
   {
   LineStripArray linien;
   Point3f ecke[] = new Point3f[5];
   int stripzahl[] = {5};
   ecke[0] = new Point3f(-0.4f, -0.06f, 0f);
   ecke[1] = new Point3f(-0.2f, 0f, 0f);
   ecke[2] = new Point3f(0f, 0f, 0f);
   ecke[3] = new Point3f(0.2f, 0f, 0f);
   ecke[4] = new Point3f(0.4f, -0.06f, 0f);
   linien = new LineStripArray(5,
      LineArray.COORDINATES, stripzahl);
   linien.setCoordinates(0, ecke);
   return linien;
   }
public GeometryArray macheFrame1()
   {
   LineStripArray linien;
   Point3f ecke[] = new Point3f[5];
   int stripzahl[] = {5};
   ecke[0] = new Point3f(-0.36f, 0.26f, 0f);
   ecke[1] = new Point3f(-0.12f, 0.2f, 0f);
   ecke[2] = new Point3f(0f, 0f, 0f);
   ecke[3] = new Point3f(0.12f, 0.2f, 0f);
   ecke[4] = new Point3f(0.36f, 0.26f, 0f);
   linien = new LineStripArray(5,
      LineArray.COORDINATES, stripzahl);
   linien.setCoordinates(0, ecke);
   return linien;
   }
public GeometryArray macheFrame3()
   {
   LineStripArray linien;
   Point3f ecke[] = new Point3f[5];
   int stripzahl[] = {5};
   ecke[0] = new Point3f(-0.24f, -0.4f, 0f);
```

```
    ecke[1] = new Point3f(-0.12f, -0.12f, 0f);
    ecke[2] = new Point3f(0f, 0f, 0f);
    ecke[3] = new Point3f(0.12f, -0.12f, 0f);
    ecke[4] = new Point3f(0.24f, -0.4f, 0f);
    linien = new LineStripArray(5,
       LineArray.COORDINATES, stripzahl);
    linien.setCoordinates(0, ecke);
    return linien;
    }

public class MeinMorphBehavior extends Behavior
    {
    private Morph zielMorph;
    private Alpha alpha;
    private double[] gewichte = {0, 0, 0, 0};
    private WakeupCondition trigger = new
       WakeupOnElapsedFrames(0);

    // Konstruktor-Aufruf soll Ziel-Morph und
    // Alpha enthalten:
    MeinMorphBehavior(Morph zm, Alpha al)
       {
       this.zielMorph = zm;
       this.alpha = al;
       }

    // initialize-Methode (obligatorisch):
    public void initialize()
       {
       this.wakeupOn(trigger);
       }

    // processStimulus, obligatorisch:
    public void processStimulus(Enumeration krit)
       {
       gewichte[0] = 0; gewichte[1] = 0;
       gewichte[2] = 0; gewichte[3] = 0;
       float t = 4f * alpha.value(); //hole Alpha-Wert
       // lineare Interpolation zwischen 2 Frames:
       int index = (int) t;      // welches Frame
       gewichte[index] =
          1.0 - ((double) t - (double) index);
       if (index < 3)             // welches 2. Frame
          gewichte[index+1] = 1.0 - gewichte[index];
       else
```

```
            gewichte[0] = 1.0 - gewichte[index];
         zielMorph.setWeights(gewichte);
         this.wakeupOn(trigger);
         }
      }  // end of class MeinMorphBehavior

public BranchGroup macheSzenengraph()
   {
   BranchGroup objWurzel = new BranchGroup();
   BoundingSphere riesenkugel =
      new BoundingSphere(
         new Point3d(0.0d, 0.0d, 0.0d),
         Double.MAX_VALUE);

   // GeometryArray[] erzeugen:
   GeometryArray[] geomArray = new GeometryArray[4];
   geomArray[0] = macheFrame0();
   geomArray[1] = macheFrame1();
   geomArray[2] = macheFrame0();
   geomArray[3] = macheFrame3();

   // Morph-Objekt erzeugen:
   Morph meinMorph = new Morph(geomArray);
   meinMorph.setCapability(
      Morph.ALLOW_WEIGHTS_WRITE);

   // Alpha-Objekt erzeugen:
   Alpha alpha = new Alpha(-1, 2000);
      // 2 Sek. Schleife
   alpha.setIncreasingAlphaRampDuration(100);

   // selbstgeschriebenes Behavior-Obj. erzeugen:
   MeinMorphBehavior mbh =
      new MeinMorphBehavior(meinMorph, alpha);
   mbh.setSchedulingBounds(riesenkugel);

   // Szenengraph zusammenbauen:
   objWurzel.addChild(meinMorph);
   objWurzel.addChild(mbh);

   // Optimierungen am Szenengraphen u. Abschluss:
   objWurzel.compile();
   return objWurzel;
   } // end macheSzenengraph-Methode
```

Ergebnis (invertierte Bilder):



Voraussetzungen für Verwendung von **`Morph`**:

- die verwendeten GeometryArrays müssen alle vom selben Typ sein,
- sie müssen alle dieselbe Eckenzahl haben.

Das **`Morph`**-Objekt benutzt ein Feld von Gewichtswerten, die für die lineare Interpolation zwischen den GeometryArrays verwendet werden. Die jeweilige aktuelle Verteilung der Gewichte wird zur Laufzeit durch die Methode **`processStimulus`** geleistet.

Texturen

Texturierung ist in Java 3D mit vielfältigen Optionen möglich. Das folgende Beispiel soll lediglich die einfachste Variante zeigen.

Texturen werden von **`Appearance`**-Knoten referenziert. Sie müssen zunächst geladen werden. Standardmäßig können Rasterdateien im JPG- oder GIF-Format geladen werden. Hierzu ist ein **`TextureLoader`**-Objekt notwendig.
Als Zeilen- und Spaltenzahl des Texturbildes werden von Java 3D intern Zweierpotenzen erwartet. Der Loader transformiert Dateien beliebiger Größe automatisch auf ein solches Format.

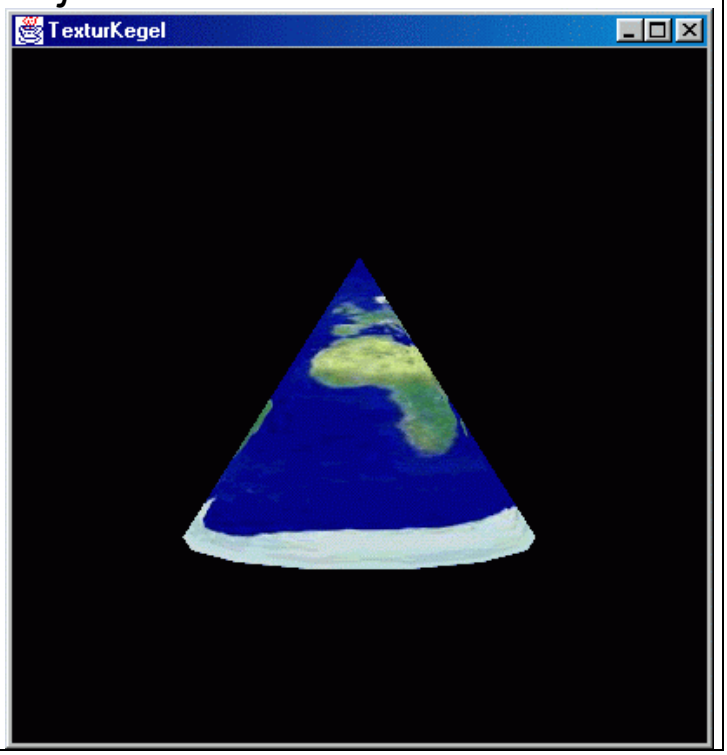
Bei den zu texturierenden geometrischen Objekten muss durch Setzen eines Flags sichergestellt werden, dass sie mit Texturkoordinaten versehen werden. Sonst kann keine Textur angebracht werden. Bei Primitiv-Objekten geschieht das Setzen des Flags durch Aufruf der Methode **`Primitive.GENERATE_TEXTURE_COORDS`** im Konstruktor (nach den Abmessungs-Angaben). Bei **`GeometryArray`**-Objekten wird dieselbe Kontroll-**`int`**-Zahl benutzt wie für **`COORDINATES`**, **`NORMALS`** etc. (siehe Erklärung des Konstruktor-Aufrufs vor dem Oktaeder-Beispiel); das vordefinierte Schlüsselwort heißt hier **`TEXTURE_COORDINATE_2`**.

Das folgende Codefragment zeigt die Szenengraph-Erzeugungs-Methode für einen Kegel, auf dessen Oberfläche eine Weltkarte aufgebracht wird. Der Rest des Programms entspricht dem FarbKegel-Beispiel.

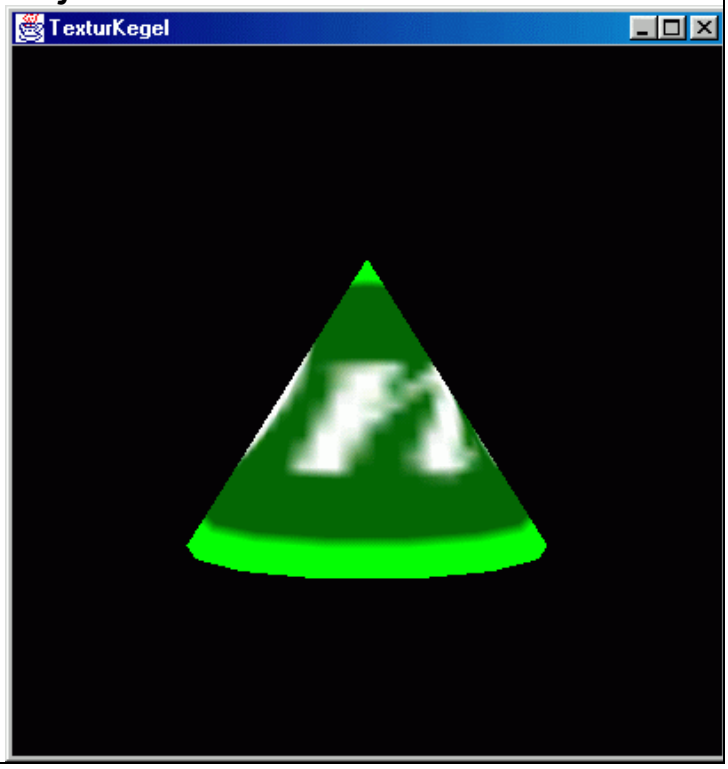
```
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.TextureLoader;
// ...
// selbstgeschriebene Visual object class:
public class MeinTexturKegel extends Shape3D
   {
   private BranchGroup meineBG;

   public MeinTexturKegel()
      {
      meineBG = new BranchGroup();
      Appearance app = macheTextur();
      Cone kegel = new Cone(0.5f, 0.8f,
         Cone.GENERATE_TEXTURE_COORDS, app);
      meineBG.addChild(kegel);
      meineBG.compile();
      }

   Appearance macheTextur()
      {
      Appearance a = new Appearance();
      TextureLoader loader = new
         TextureLoader("earth.jpg", null);
```

```
        Texture2D textur =
            (Texture2D) loader.getTexture();
        // ImageComponent2D bild = loader.getImage();
        // Texture2D textur = new Texture2D();
        // textur.setImage(0, bild);
        a.setTexture(textur);
        return a;
        }

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

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

Ergebnis:

Textur	*Objekt*
	

Variante:

Textur:	*Objekt:*
	

Zur Manipulation der Texturkoordinaten-Zuordnung, Anwendung von MipMapping und anderer fortgeschrittener Textur-Techniken wird auf die Dokumentation und das Tutorial verwiesen.