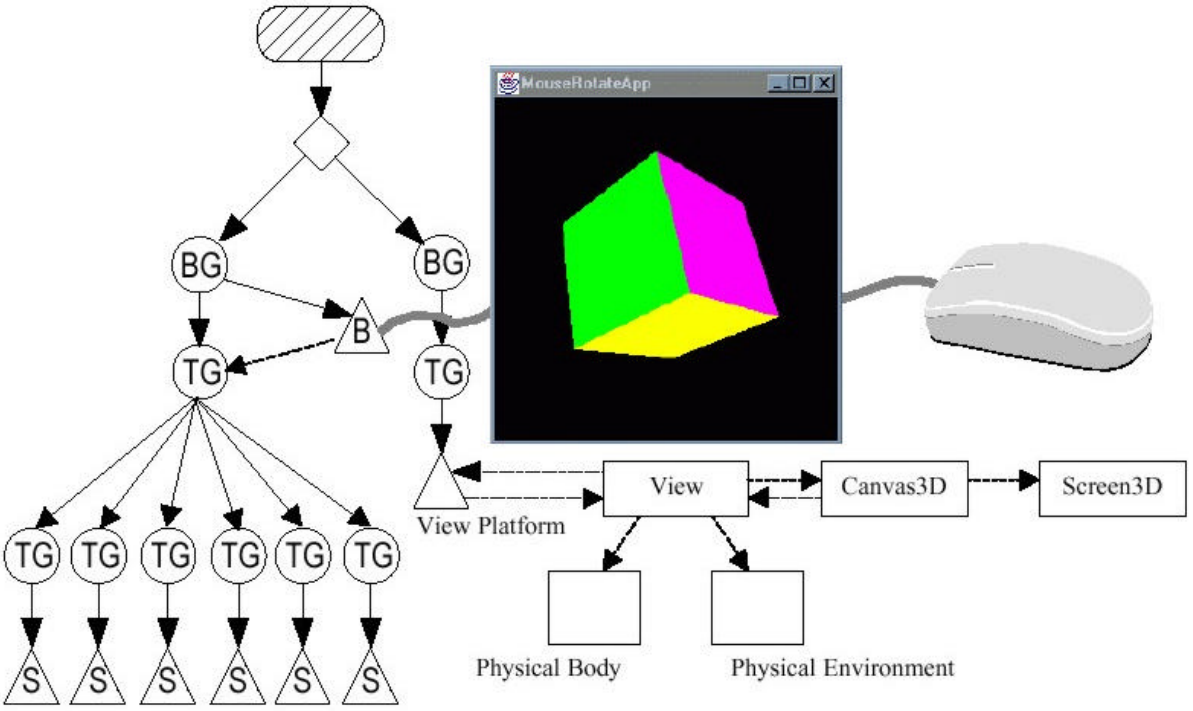
Java 3D
Interaktionen



Ausarbeitung zum Seminar „Java 3D“ im 7.Semester unter Leitung von Prof. Dr. W. Heinzel

zum Thema

Interaktionen

von
Stefan Bohl

Gliederung:

Die erzeugten, in diesem Tutorial verwendeten Beispiele finden Sie in Sourcecodeform auf der beigelegten CD.

1. Allgemeines zum Thema Interaktionen unter Java 3D

Für heutige Bedürfnisse in der Computeranwendung sind Interaktionen von Benutzer und Maschine nicht mehr wegzudenken. Heutzutage nicht mehr nur alleine für Spiele und zu Verschönerungszwecken, vielmehr auch für weitere Anwendungsgebiete. So treten immer mehr Programme in Erscheinung, bei denen ohne Interaktionen keine Funktionen ausgeschöpft werden könnten, so z.B. für Tomographien, die Entwicklung komplexer Lösungen im Automobilbau, für Aritektonische Neubauten, usw.

Aber was sind Interaktionen?
Wenn sich eine Szene in Reaktion auf Benutzereingaben ändert.

Hierbei gibt's es **sechs wesentliche Interaktionen eines Benutzers**.

- **Mausbewegung**
- **Mausklick**
- **Tastendruck**
- **Navigation**
- **Kollision**
- **Kombinationen**

Mausbewegung: Anwendungen hierbei sind z.B. Zeichenanwendungen, Richtungsänderungen, Verschieben
Mausklick: Beispiele sind hier Festhalten von Objekten, Betätigen von Flächen, Aktionen, Markieren
Tastendruck: Änderungen von Attributen, Richtungsänderungen, Farbenänderungen
Navigation: Wie der Name schon sagt, zur Navigation von Objekten oder Szenen
Kollision: Hierbei handelt es sich um Änderungen (z.B. Attributsänderungenm Richtungsänderungen), die durch zwei Objekte, die z.B. kollidierten, ausgelöst wurden
Kombinationen: Am häufigsten treten in der Anwendung Kombinationen von Einzelinteraktionen auf. Überall dort, wo man mit einer Interaktion das Programm

nicht vollständig beherrschen kann: Spiele, Bildbearbeitungen, 3D-Studios, CAD usw.

Im oben definierten Begriff Interaktionen findet sich der Begriff **„Änderungen von Szenen“** wieder.
Dies können sein:

- **Hinzufügen von Objekten**
- **Entfernen von Objekten**
- **Umordnung, Standortänderungen**
- **Verändern von Attributen**
- **Verändern von Transformationen von Objekten**

Diese sind stark verankert mit den Benutzerinteraktionen, wie oben beschrieben. So können mittels Kollision Attribute geändert oder Objekte entfernt werden. Ebenfalls sind mit Tastendruck, Navigation Umordnungen und Standortänderungen durchführbar. Mit Kombinationen aus Benutzereingaben wiederum viele verschiedene Kombinationen von Szenenänderungen möglich.

Szenenänderungen ohne Benutzereingaben sowie Benutzereingaben ohne Szenenänderungen sind zwar durchführbar, aber nicht sehr hilfreich für die Umsetzung von Problemen und sollten deshalb nicht verwendet werden. Sie sind lediglich sinnvoll wenn eine Animation realisiert werden soll, diese schließt aber eben wieder die Benutzereingaben aus.

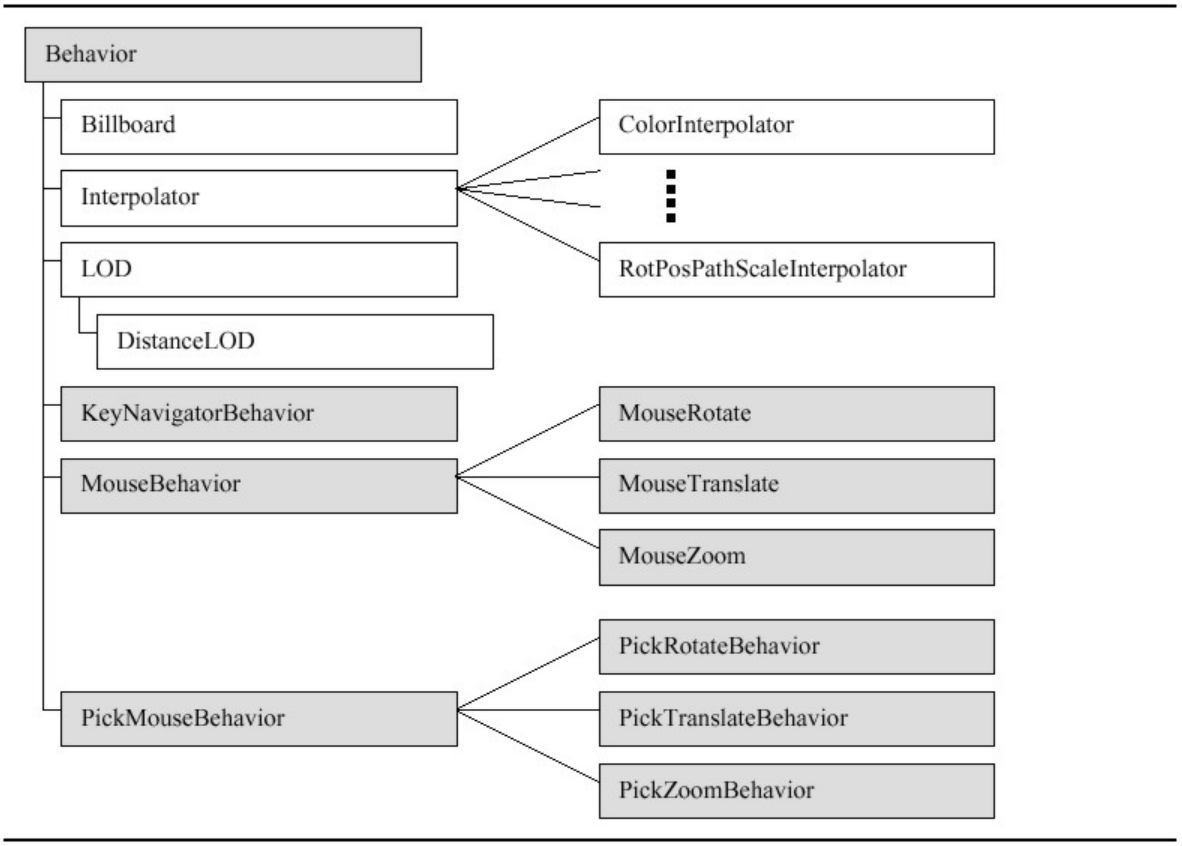
2. Die Behavior-Klassen (Verhaltensklassen)

Die Behavior-Klasse ist die Grundlage für Interaktionen, Animationen und Szenenänderungen. Sie ist eine abstrakte Klasse, die den Szenengraph ändern kann. Sie stellt dem Benutzer Code bereit um graphische Änderungen auf der virtuellen Umgebung durchführen zu können.
Das Hauptmerkmal dieser Klasse ist es, Szenenänderungen mittels Aktionen zu machen: **Eine Aktion kann eine Maus- oder Tastatureingabe, ein Zeitlimit oder ein Aufeinandertreffen (Kollision) zweier Objekte sein**. Eine Kombination mehrere Aktionen ist möglich. Dies geschieht durch oben beschriebene Szenenänderungen.

Folgende Tabelle zeigt die wesentlichsten Aktionen:

Aktion	**TransformGroup** Ändert Orientierung oder Standort	**Geometry** Ändert Farbe oder Aussehen	**SceneGraph** Fügt hinzu oder entfernt	**View** Ändert Blick
Benutzer	Interaktion	Applikations-abhängig	Applikations-Abhängig	Navigation
Kollision	Ändert Orientierung oder Standort	Ändert Erscheinen nach Kollision	Entfernt nach einer Kollision	Ändert Blickwinkel nach einer Kollision
Zeit	Animation	Animation	Animation	Animation

Ein Behavior unterstützt folgende Verhalten:



Billboard: Anschlagsverhalten, ist für die Orientierung von Objekten zuständig. Komplexe Objekte können auf Distanz gebracht und somit ganz angezeigt werden.
Interpolator: z.B. RotationInterpolator
LOD (Level of Detail): Ändert den Detailgrad von Objekten, so dass diese auch vollständig und gut dargestellt werden können. (verschieben das Objekt)
KeyNavigatorBehavior: Navigieren mit der Tastatur
MouseBehavior: Navigieren und Manipulieren mit der Maus
PickMouseBehavior: Manipulieren (picking) mit der Maus, sozusagen drag&drop

Wichtig:

Ein Verhalten (Behavior) wird aber nur dann ausgeführt, wenn der Aktivationsradius der Kamera den Schedulingbereich des Behaviors schneidet. Ein Schedulingbereich ist nicht standardmäßig, d.h. er muss explizit angegeben werden.
=> Verhalten wird sonst nie ausgeführt!

3. Schedulingbereiche, Behaviors und Szenenmodellierung

Ein Schedulingbereich eines Behaviors ist ein begrenzter Bereich, z.B. Container, Box etc. Für große Behaviors werden so genannte BoundingSphere verwendet.
Ist der Schedulingbereich relativ zum Koordinatensystem des Behaviors, so ist dieser im Ursprung des Koordinatensystems und der Schedulingbereich bewegt sich mit mit dem Behavior mit, sofern dieser bewegt wird. Eine andere Möglichkeit ist, dass der Schedulingbereich relativ zum Koordinatensystem eines BoundingLeaf ist. Dies wird durch folgende Punkte definiert:

1. Der begrenzte Bereich befindet sich im Ursprung der darüber liegenden TransformGroup
2. Nur wenn sich der BoundingLeaf bewegt, bewegt sich der Schedulingbereich mit, bei Bewegung des Behaviors, bleibt der Schedulingebereich stehen.

Zum ausführen von Interaktionen muss der Behavior an den sich ändernden Bereich angekoppelt sein, der wiederum nur innerhalb dieses gekoppelten Schedulingbereichs gültig ist. Da der Schedulingbereich an den Raum angebunden ist, und hier wiederum der Behavior an den Schedulingbereich, so bewegt sich der Behavior, wenn der Raum seine Postition ändert. **Hierbei ist aber noch zu beachten, dass der Behavior nur dann aktiv ist, wenn er im Sichtfeld, in der ViewPlatform, des Raumes sich befindet, d.h. wird nur ausgeführt, wenn die Änderungen auch sichtbar für den Betrachter sind.**

Zum verwenden von Behaviors in einem Java 3D Programm gibt es zwei Wege:

1. Man verwendet vordefinierte Klassen, wie kurz erwähnt BoundingSphere und BoundingLeaf. Auf eine nähere Betrachtung wird hier verzichtet.
2. Die zweite Möglichkeit, die wohl wichtigste, ist die Verwendung von eigenen, benutzerdefinierten Behaviorklassen. Auf diese Möglichkeit, wie man solche Klassen selber schreibt wird hier näher darauf eingegangen.

Zum schreiben von benutzerdefinierten Behaviors benötigt man **vier grundlegende Bausteine:**

1. Für Interaktionen und die daraus resultierenden Änderungen benötigt man zweifelsohne erst einmal ein **Objekt**, das verändert werden soll, z.B. Kugel, Würfel
2. Eine **initialize()**-Methode, die den TriggerEvent initialisiert und somit da Objekt über eine WakeupCondition aufwachen lässt. Haucht sozusagen Leben ein.
3. Die **processStimulus()**-Methode, die angibt, welche Änderungen am Objekt gemacht werden und auf welche Interaktion (Maus-, Tastatureingabe) geändert wird.
4. Einen so genannten **WakeupOnAWTEvent**. Dieser wartet auf ein definiertes Signal (Tastatur- oder Mauseingabe) und übergibt dieses. Danach wird die Änderung, falls erfolgreich, vorgenommen.

Das allgemeine Vorgehen beim Schreiben einer Behavior-Klasse

1. Man erzeuge eine Unterklasse der Klasse Behavior

   ```
   public class TestBehavior extends Behavior
   ```

2. Definieren eines Konstruktors, das eine Referenz auf das zu ändernde Objekt darstellt.

   ```
   TestBehavior(TransformGroup targetTG)        {
       this.targetTG = targetTG;
   }
   ```

3. Schreiben der initialize()-Methode, die den TriggerEvent initialisiert

   ```
   public void initialize()        {
       this.wakeupOn(new WakeupOnAWTEvent( /*worauf reagiert
       werden soll, Tastatur oder Maus, näheres in Kapitel 4 */ ));
   }
   ```

4. Schreiben der processStimulus()-Methode, welche den TrigerEvent umsetzt, das Objekt manipuliert und einen neuen TriggerEvent zur Verfügung stellt, darauf wieder wartet.

   ```
   public void processStimulus(Enumeration criteria)       {
       /* definiere die Objektänderungen */
       targetTG.setTransform(rotation);
   ```

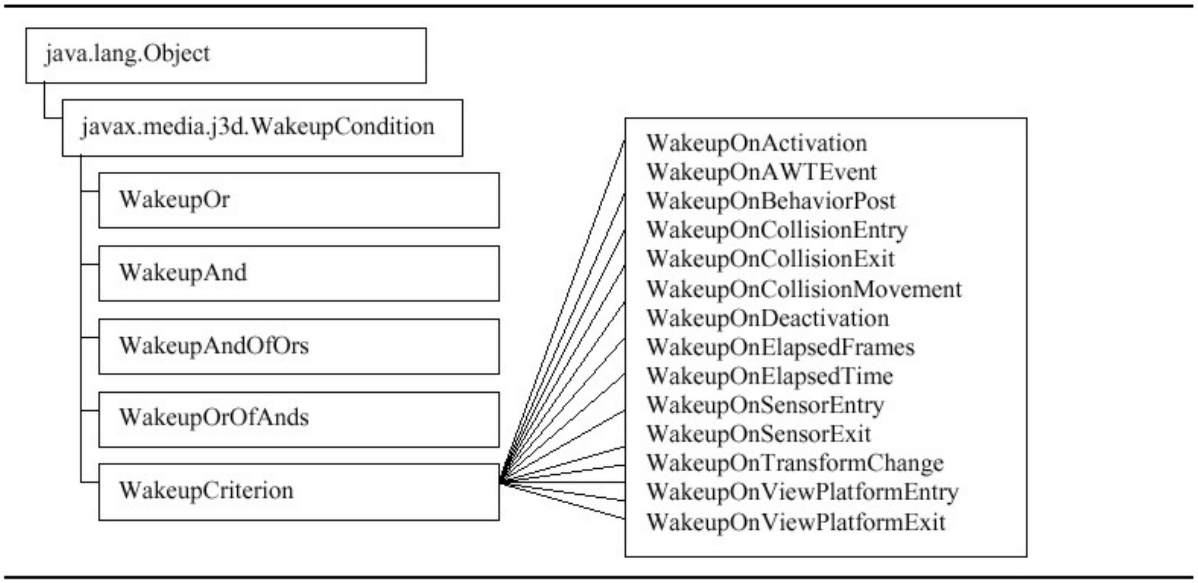
```
            this.wakeupOn(new WakeupOnAWTEvent( /*worauf reagiert
            werden soll, Tastatur oder Maus, näheres in Kapitel 4  */ ));
        }
```

Als weiterer Schritt, der nicht unter das Schreiben einer Behavior-Klasse fällt, muss noch eine Szene modelliert werden. Hier kann standardmäßig für einen Würfel folgender Code verwendet werden:

```
/***   Modellierung der Szene   ***/
  public BranchGroup createSceneGraph() {
        // Aufbau des Scenengraphen
        BranchGroup objRoot = new BranchGroup();

        // Fähigkeit zum Schreiben des manilpulierenden Objektes
        TransformGroup objRotate = new TransformGroup();
    objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

        // Aufbau des Scenengraphen
        objRoot.addChild(objRotate);
        // Als Kindobjekt den Würfel hinzufügen
        objRotate.addChild(new ColorCube(0.4));

        // Erzeugen einer Instanz von TestBehavior
        TestBehavior myRotationBehavior = new  TestBehavior(objRotate);
        // Referenz wird übergeben
        myRotationBehavior.setSchedulingBounds(new BoundingSphere());
        // Der Schedulingbereich wird benannt und in den Szenengraph hinzugefügt
        objRoot.addChild(myRotationBehavior);

        // Kompiliert (Optimiert) den Szenengraph
        objRoot.compile();
        return objRoot;
  }
```

4. WakeupConditions

Ein Behavior wird durch ein bestimmtes, festgelegtes Ergebnis ausgelöst. Diese Ereignisse werden durch WakeupCondition spezifiziert, welche die Urklasse in Java 3D darstellt. Fünf Klassen leiten sich aus dieser WakeupCondition ab, wobei eine die abstrakte WakeupCriterion spezifiziert, diejenige, welche die einzelnen Ereignisbehandlungen beinhaltetet. Die restlichen erlauben eine Zusammensetzung mehrerer „Aufwach“-Kriterien miteinander.



Die fünf Unterklassen von WakeupCondition

Die WakeupCondition unterstützt zwei Methoden:

- **allElements()**: gibt alle Wakeup-Kriterien dem WakeupCondition zurück.
- **triggeredElements()**: gibt zurück, welche Wakeup-Kriterien den Trigger ausgelöst haben.

WakeupCriterion:

Diese abstrakte Klasse beinhaltet 14 „Aufwach“-Kriterien, die in Klassen aufbereitet sind. Diese Klasse liefert nur eine Methode: **hasTriggered()** (hat ausgelöst). Dieses liefert nur „wahr“ oder „falsch“ Werte zurück, eben ob sie ausgelöst, aufgeweckt wurden oder nicht.

Die 14 Unterklassen von WakeupCriterion (14 „Aufwach“-Kriterien)

Wakeup Kriterium	Auslöser
WakeupOnActiviation	Wird ausgelöst, wenn die ViewPlatform in den Schedulingbereich eintritt, bzw. diesen schneidet und den Behavior aktiviert
WakeupOnAWTEvent	Durch einen spezifizierten AWT Event (Tastatur-Mausbelegung)
WakeupOnBehaviorPost	Wenn ein spezifiziertes Verhaltensobjekt einen definierten Trigger anschlägt
WakeupOnCollisionEntry	Bei einer Kollision mit einem anderen Objekt, hierbei gilt der Erstkontakt.
WakeupOnCollisionExit	Bei einer Kollision mit einem anderen Objekt, wobei hierbei der Event erst nach Beenden der Kollision ausgelöst wird.
WakeupOnCollisionMovement	Wenn sich ein Objekt bei der Kollision mit einem weitere Objekt bewegt
WakeupOnDeactivation	Wird gestartet, wenn die ViewPlatform in den Schedulingbereich austritt.
WakeupOnElapsedFrames	Wenn eine definierte Anzahl von Frames durchlaufen wurde
WakeupOnElapsedTime	Wenn eine definierte Zeitvorgabe abgelaufen ist
WakeupOnSensorEntry	Beim ersten Kontakt, wenn eine definierte Grenze überschritten wird
WakeupOnSensorExit	Wenn ein Bereich mit dessen Grenze, in der man sich befand, verlassen wird.
WakeupOnTransformChange	Wenn eine andere Transformation am Objekt durchgeführt wird.
WakeupOnPlatformEntry	Beim ersten Kontakt mit einer ViewPlatform, die eine Grenze überschreitet
WakeupOnPlatformExit	Wenn ein View nicht länger eine Grenze durchschneidet.

Kurze Erläuterung der Kriterien

Für die Klasse WakeupOnAWTEvent wiederum gibt es verschiedene Auslösevarianten, durch die das Objekt sein Verhalten, Standort ändern kann:

- **KeyEvent.KEY_TYPED**
- **KeyEvent.KEY_PRESSED**
- **KeyEvent.KEY_RELEASED**
- **KeyEvent.MOUSE_CLICKES**
- **KeyEvent.MOUSE_PRESSED**
- **KeyEvent.MOUSE_RELEASED**
- **KeyEvent.MOUSE_MOVED**
- **KeyEvent.MOUSE_DRAGGED**

Aber auch

- **MouseEvent.MOUSE_CLICKED**
- **MouseEvent.MOUSE_PRESSED**

So haben auch die restlichen 13 Unterklassen jeweils zugeordnete Variablen, auf die ich hierbei allerdings nicht eingehen möchte. Da dies für diese Ausarbeitung zu aufwendig wäre und den Rahmen dessen, was hier abgedeckt wird, sprengen würde. Hier empfehle ich einfach das Java 3D Tutorial von sun, Kapitel 4, für ausführlichere Instruktionen.

5. Interaktionsvarianten

5.1 Interaktionen per Tastatur und Maus

Hierbei wird ein einfaches Programm realisiert, welches den Zweck hat, einen Würfel zu zeichnen, der per Tastatur und/oder Maus gedreht werden kann.
Als erstes Beispiel werden wir den Würfel zeichnen und Ihn bei beliebigem Tastendruck um die Y-Achse drehen.
Bei zweiten werden wir den Event so umändern, dass sich der Würfel auf einen Masuklick ändert.
Als drittes Beispiel in diesem Unterkapitel werden wir das Programm so ändern, dass sich der Würfel in die X- und die Y-Achse drehen kann, allerdings nur bei vordefinierten Tastaturtasten.

Zu Beispiel 1:

Unser Programm soll einen simplen Würfel (ColorCube) rendert, der sich bei beliebiger Tasteneingabe um die Y-Achse bewegt.
Um ein Javaprogramm zu starten, müssen erst einmal die benötigten Klassen importiert werden.

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

Dies sollte für die Ausführung des Programms reichen.
Wir benennen unser Programm **RotateApplication** welches von einem Applet abgeleitet wird, weshalb wir auch die Klasse java.applet.Applet importieren.

```
public class RotateApplication extends Applet {
```

Als nächstes benötigen wir eine Behaviorklasse, so genannte Subklasse von Behavior, die wir in unserem Beispiel **SimpleBehavior** nennen und von einem Behavior ableiten. Hierin werden nun die Interktionsschritte beschrieben, angefangen mit der Definition der **TransformGroup** und von **Variablen** (rotation, abgeleitet von Transform3D und angle, die als Drehvariable benutzt wird).

```
public class SimpleBehavior extends Behavior        {

    // Eine TransformGroup wird angelegt
    private TransformGroup targetTG;
    private Transform3D rotation = new Transform3D();
    private double angle = 0.0;
```

Als nächste benötigen wir einen **einfachen Konstuktor**, welcher die Aufgabe hat, eine **Referenz des zu manipulierenden Objektes** (hier ColorCube) zu speichern.

```
    // Einfacher SimpleBehavior Konstruktor
    SimpleBehavior(TransformGroup targetTG)    {
       this.targetTG = targetTG;
    }
```

Im nächsten Schritt legen wir die **initialize()**-Methode an, die den initialen **Auslöser spezifiziert**.

```
    // Inizialisiere den Behavior, setze die "Aufwach"-Bedingungen, dann wenn der Behavior
    beginnt zu reagieren
    public void initialize() {
          // wacht auf, wenn eine Taste gedrückt wurde
     this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
    }
```

Zu guter letzt wird die Methode **processStimulus()** integriert. In dieser werden die Hauptschritte zum starten, durchführen und ändern des Events geschrieben. Hier erfolgt der eigentliche Teil der Interaktion. In der processStimulus()-Methode wird der neue Radius berechnet, die Rotation um die Y-Achse mit der Variable angle durchgeführt, sowie die neue Ausrichtung des Würfels der TransformGroup

zugeordnet und gesetzt. So kann nach Tastendruck der gedrehte Würfel in der ViewPlatform dem Benutzer angezeigt werden. Doch wie wird der ColorCube aufgeweckt? Dieser Szene wird ein neues WakeupOnAWTEvent zugeordnet, welches auf einen Tastendruck reagiert (siehe oben: WakeupConditions: KeyEvent.KEY_PRESSED)

```
        // manipuliert das Object nach Tastatureingabe
        public void processStimulus(Enumeration criteria)   {
        // bewegt das Object um die Y-Achse
                angle += 0.1;
                rotation.rotY(angle);
                targetTG.setTransform(rotation);
                // wecke den ColorCube bei Tastendruck auf
                this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
        }
}       // Ende des SimpleBehaviors
```

Anfolgend muss noch die Szene modelliert werden. Dies geschieht mittels einer BranchGroup. In ihr werden ein neuer BrachGroup, namens objRoot, sowie eine TransformGroup, namens objRotate, erzeugt. objRotate nimmt den ColorCube als Kindobjekt auf und objRoot wiederum nimmt dann objRotate als Kindobjekt auf. Nun wird noch der Schedulingbereich gesetzt, der als Kind dem objRoot zugefügt wird, damit dem Behavior ein Schedulingbereich zugewiesen wird. Somit schneiden sich Schedulingbereich und Kamerablick. Um die Erstellung zu optimieren wird der Szenengraph vorkompiliert und freigegeben.

```
/***   Modellierung der Szene   ***/
   public BranchGroup createSceneGraph() {
        BranchGroup objRoot = new BranchGroup();
        TransformGroup objRotate = new TransformGroup();
        objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

        objRoot.addChild(objRotate);
        objRotate.addChild(new ColorCube(0.4));

        SimpleBehavior myRotationBehavior = new SimpleBehavior(objRotate);
```

```
        myRotationBehavior.setSchedulingBounds(new BoundingSphere());
        objRoot.addChild(myRotationBehavior);

        // Kompiliert (Optimiert) den Szenengraph
        objRoot.compile();

        return objRoot;
    }
```

Anfolgend erscheinen noch ein Konstruktor, welcher die Java 3D Umgebung initialisiert, und die Hauptfunktion zum Starten des Programms.

```
// Der Konstruktor initialisiert die Java3D-Umgebung
public RotateApplication()    {
        // Layout, hier BorderLayout, festlegen
        setLayout(new BorderLayout());
        // Erzeugt eine Leinwand (=Canvas), auf der die Szene dargestellt wird
        Canvas3D canvas3D = new Canvas3D(null);
        // Java3D-Koordinatensystem wird zur Leinwand hinzugefügt
        add("Center", canvas3D);
        // Aufruf der Funktion zur Erstellung des Szenengraphen (bzw. der Szene)
        BranchGroup scene = createSceneGraph();
        // Erstellung des SimpleUniverse als Root-Objekt der gesamten Szene
        SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
        // Die Kameraposition wird entlang der Z-Achse so versetzt, dass die
        // Fenstergrenzen jeweils bei -1 und 1 des Koordinatensystems liegen
        simpleU.getViewingPlatform().setNominalViewingTransform();
        // Szenengraph in SimpleUniverse einfügen
        simpleU.addBranchGraph(scene);
    }

// Hauptfunktion zur Erzeugung des Applets
public static void main(String[] args) {
        Frame frame = new MainFrame(new RotateApplication(), 256, 256);
    }
}
```

Beispiel 2:

Um das obere Beispiel soweit zu ändern, dass hierbei, statt auf Tastatureingaben, auf Mausklicks reagiert wird, sind keine großartigen Anstrengungen im Code zu unternehmen. Dabei müssen nur in den Wakeup-Bedingungen in der initialize()- und der processStimulus()-Methode vom KeyEvent auf den MouseEvent geändert werden. So wie in Abschnitt 4 (WakeupCriterion) unter WakeupOnAWTEvent Beispiele für die Mausauslöser aufgezeigt wurden.

Die neue Bedingung und de neue Programmcode in der initialize()- und processStimulus()-Methode:

Alt:

```
this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
```

Neu:

```
this.wakeupOn(new WakeupOnAWTEvent(MouseEvent.MOUSE_PRESSED));
```

Und schon rotiert der Würfel beim drücken einer Maustaste. Ebenfalls einzusehen im Beispiel **RotateApplication**, wenn die Zeilen 51 und 66 auskommentiert und daraufhin die Zeilen 49 und 64 einkommentiert, oder Beispiel **RotateApplication3.**

Beispiel 3:

Um nun auf definierte Tastendrücke reagieren zu können, müssen mehrere Änderungen im Programmcode unternommen werden. **Siehe dazu auch das Beispiel RotateApplication2. In diesem Beispiel rotiert der ColorCube auf der Y-Achse bei der betätigten Taste „-„ und auf der X-Achse mit der Taste „+“.** Die Änderungen beziehen sich allerdings nur auf den Teil des SimpleBehaviors, also dort, wo der eigentliche Teil des Verhaltens definiert wird. Zuallererst benötigen wir ein neues Transform3D-Objekt, welches die verschiedenen Aktionen (Rotation um die Y-Achse, Rotation um die X-Achse) verbinden kann und zu einem Objekt erschließt, man kann sagen einen Speicher.

```
private Transform3D rotationTemp = new Transform3D();
```

Der einfache Konstruktor sowie die initialize()-Methode bleiben so erhalten wie in Beispiel 1. Dort werden keine Änderungen vorgenommen.
Die Methode processStimulus() muss völlig umgestaltet werden. Am Anfang wird ein Event definiert, der einem WakeupOnAWTEvent zugeordnet ist. Wird also eine Taste gedrückt, so wacht das Objekt auf und speichert die Tastenid in der Variablen namens event.

```
// Wakeupbedingung, d.h. tu was, wenn etwas geschieht
WakeupOnAWTEvent event =  (WakeupOnAWTEvent) criteria.nextElement();
```

Als nächster Schritt werden das neue Transform3D-Objekt rotationTemp und rotation auf den gleichen Stand gesetzt, so dass bei Änderung keine Abweichungen, falschen Ergebnisse erzielt und angezeigt werden. Es existieren ja mehrere Änderungsmöglichkeiten des Würfels.

```
// dem rotationTemp wird rotation zugeordnet, da mehrere Bewegungen vorherschen
rotationTemp.set(rotation);
```

Als nächstes wird ein neues KeyEvent, namens key, erzeugt. Dieses soll bei einem KeyEvent, also Tastaturaktion, die gedrückte Taste speichern um sie auswerten zu können.

```
// Hier wird der Event zugeordnet, hierbei handelt es sich um einen Tastaturevent
// ließt den Tastendruck aus
KeyEvent key = (KeyEvent) event.getAWTEvent()[0];
```

Nun muss der gespeicherte Tastaturwert in ein, für das Programm weiterverarbeitendes, Zeichen umgewandelt werden. Dieses Zeichen wird wiederum in einer Variablen gespeichert um es verwenden zu können. Dies geschieht mittels der Funktion:

```
// wandelt die gedrückte Taste in einen Zeichen um
char c  = key.getKeyChar();
```

Über eine einfache switch-case wird verglichen, ob die gedrückte Taste eine Aktion, eine Rotation, bewirkt oder nicht (Bei Tastendruck „-„ Rotation um die Y-Achse, Tastendruck „+“ ist Rotation um die X-Achse).
Analog zu Beispiel 1 wird bei erfolgreichem Durchlaufen der switch-Anweisung die rotation um die jeweilige Achse durchgeführt. Neu kommt hier hinzu, dass die Objekte rotation und rotationTemp miteinander multipiziert werden, damit sich der Würfel nur einmal bewegt und sich nicht wieder in die Ausgangslage zurückversetzt. Würde diese Anweisung vergessen, würde sich bei der Rotation um die X-Achse der Cube auf seine letzte X-Achsen-Position zurücksetzen, d.h. es wären für beide Rotationen „unterschiedliche“ Würfel in Bewegung.

```
// Bedingungen, was getan wird, bei welchem Tastendruck
switch(c) {
// Wenn + gedrückt...
        case '+': {
                angle += 0.1;
                // rotiere um die X-Achse mit 0.1-Schritten
                rotation.rotX(angle);
                // füge diese Rotation dem Gesamtbild zu
                rotation.mul(rotationTemp);
                targetTG.setTransform(rotation);
                break;
        }
        case '-': {
                angle += 0.1;
                rotation.rotY(angle);
                rotation.mul(rotationTemp);
                targetTG.setTransform(rotation);
                break;
        }
}
```

5.2 Tastaturnavigation

Bei der Tastaturgesteuerten Navigation wird nun, im Gegensatz zum vorherigen Kapitel, die **ViewPlatform dynamisch, d.h. hier wird nicht mehr das Objekt manipuliert, sondern die ViewPlatform**. Dies hat zur Folge, dass wir keinen Behavior mehr eigenhändig spezifizieren müssen, der das Objekt, den Würfel ändert. Dieser bleibt im folgendem statisch zugeordnet, es ändert sich sozusagen nur der „Blickwinkel“ auf dieses Objekt.
Zum schreiben eines Programms, in dem man per Tastatur navigieren kann, benötigt man eine vordefinierte Verhaltensklasse, die auf Tastaturbefehle reagiert, den so genannten **KeyNavigationBehavior**. **In dieses Klasse sind alle Methoden** einer benutzerdefinierten Verhaltensklasse enthalten (siehe 5.1), also die **initialize()** sowie die **processStimulus()-Methode**. Der in Ihr enthaltene AWTEvent überprüft die Eingaben, ob sie einem definierten Wert entspricht. Hierbei sind folgende Werte zugelassen:

Key	movement	**Alt-key movement**
←	rotate left	lateral translate left
→	rotate right	lateral translate right
↑	move forward	
↓	move backward	
PgUp	rotate up	translation up
PgDn	rotate down	translation down
+	restore back clip distance (and return to the origin)	
-	reduce back clip distance	
=	return to center of universe	

Nach den ermittelten Tastenanschlägen werden dann die entsprechenden Ergebnisse, Events, (siehe Spalte „movement“) angewandt und dargestellt.
Das Programm besteht also nur aus drei Teilen: einer Funktion zur Modellierung der Szene, dem **createSceneGraph()**, dem **Konstruktor** zum initialisieren der 3D-

Umgebung sowie die Hausptfunktion, **der main()**. Das Beispielprogramm hierzu lautet **KeyNavigatorApp()**.
Die Vorgehensweise:
Zu allererst müssen die benötigten Klassen importiert werden. Dazu verwendet man die Klassen aus Beispiel 5.1 sowie zusätzlich zwei neue, die die Methoden für die KeyNavigation enthalten. Dies sind:

```
import java.awt.AWTEvent;
import com.sun.j3d.utils.behaviors.keyboard.*;
```

Im nächsten Schritt muss die Szene mit dem benötigten **KeyNavigator modelliert** werden, sozusagen den Szenengraph kreieren. Nun werden die Variablen, Vektoren und Transformgruppen erzeugt und gesetzt. Der Vektor wird dazu verwendet, um den ersten Blickwinkel des Betrachtes, innerhalb eines XYZ-Koordinatensystems, auf das Ausgangsobjekt zu setzten.

```
// Modellierung der Szene mit einem KeyNavigator-Objekt
public BranchGroup createSceneGraph(SimpleUniverse su) {
    // Variablen, Vektoren und TransformGroup setzen
    TransformGroup vpTrans = null;
    BranchGroup objRoot = new BranchGroup();
    Vector3f translate = new Vector3f();
    Transform3D T3D = new Transform3D();
    TransformGroup TG = null;
```

Ist dies abgeschlossen, so kann das Objekt, hier ein Würfel, der Szene als Kindobjekt hinzugefügt werden.

```
    // Würfel der Szene als Kindobjekt hinzufügen
    objRoot.addChild(new ColorCube(0.2));
```

Der nächste Schritt enthält den wichtigsten Teil der Modellierungsmethode. **Nun wird die ViewPlatformTransformation mittels eines SimpleUniverse gespeichert**. Da das Objekt statisch bleibt, wird die ViewPlatform transformiert, d.h. verschoben,

geändert. **Der neue Blick, die neue Ausrichtung des ViewPlatforms wird in der zuvor angelegten TransformGroup übernommen**, gespeichert.

```
// erhält die ViewPlatformTransformation von einem SimpleUniverse Objekt zurück
vpTrans = su.getViewingPlatform().getViewPlatformTransform();
```

Nun wird, wie eingangs erwähnt, der erste Blickwinkel des Betrachters, der Ausgangsblick auf das Ausgangsobjekt, mittels des Vektors erzeugt bzw. angelegt.

```
// hier wird die Ausgangskameraposition des Betrachters gesetzt
translate.set( 0.0f, 0.3f, 0.0f);
T3D.setTranslation(translate);
vpTrans.setTransform(T3D);
```

Im nächsten Schritt wird noch die Grenze für das KeyNavigationBehavior-Objekt geliefert.

```
// setzt die Grenze für das KeyNavigatorBehavior-Objekt
keyNavBeh.setSchedulingBounds(new BoundingSphere(new Point3d(),1000.0));
```

Zum Schluss der Methode werden noch die üblichen Dinge geregelt, so wird **Behavior der Szene als Kindobjekt zugeordnet** sowie die Szene vorkompiliert und für einen schnelleren Ablauf optimiert.

```
        // Füge den KeyNavigatorBehavior der Szene zu
        objRoot.addChild(keyNavBeh);
        // Optimiert und kompiliert den Szenengraph vor
        objRoot.compile();
        return objRoot;
    }
```

Somit ist die createSceneGraph()-Methode abgeschlossen. Gehen wir also zum Konstuktor KeyNavigatorApp() über. Hierbei kann vieles aus dem Vorgänderprogramm übernommen werden. Wir **legen** wieder das **Layout fest** um danach **ein SimpleUniverse als Root-Objekt der Szene zu erzeugen**. Nun wird das **SimpleUniverse-Objekt der createSceneGraph-Objekt-Methode**

implementiert um es änderbar für Transformationen, ausgehend der ViewPlatform, **zu machen**.

```
// Konstruktor zum initialisieren der Umgebung
public KeyNavigatorApp() {
    // Layout festlegen, Zeichenfläche zuordnen
    setLayout(new BorderLayout());
    Canvas3D canvas3D = new Canvas3D(null);
    add("Center", canvas3D);

    // Erstellung des SimpleUniverse als Root-Objekt der gesamten Szene
    SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
    BranchGroup scene = createSceneGraph(simpleU);
```

Als vorletztes im Konstuktor setzten wir das Objekt, den Würfel noch so ein, so dass es im Blick des Betrachters ist. Nun kann der **Szenengraph dem SimpleUniverse zugeordnet** werden. Fertig.

```
    // setzt das Objekt so, das der Betrachter es sehen kann
    simpleU.getViewingPlatform().setNominalViewingTransform();
    // Szenengraph in SimpleUniverse einfügen
    simpleU.addBranchGraph(scene);
}
```

Noch die Hauptfunktion zum starten des Applets am Ende des Programms einfügen.

```
// Hauptfunktion zum starten des Programms
public static void main(String[] args) {
    Frame frame = new MainFrame(new KeyNavigatorApp(), 256, 256);
}
}
```

Mittels dieses Programms ist es möglich, per Pfeiltasten der Tastatur um dem Würfel vorbei zu navigieren.

5.3 Navigation mit der Maus

Die vordefinierten Maus-Verhaltens-Klassen beinhalten Verhaltensklassen, in denen die Maus als Eingabegerät für Interaktionen über ein Objekt verwendet wird. Die Hauptklassen hierbei sind **translating** (Bewegungen der Bildebene parallel zu einem Objekt), **zooming** (vorwärts und rückwärts bewegen) und **rotation** (Rotation von Objekten) zusammen mit den Bewegungen der Maus.
Die nachstehende Tabelle listet nochmals die der Verhaltenklassen und ihre Ergebnisse auf:

MouseBehavior class	Action in Response to Mouse Action	Mouse Action
MouseRotate	rotate visual object in place	left-button held with mouse movement
MouseTranslate	translate the visual object in a plane parallel to the image plate	right-button held with mouse movement
MouseZoom	translate the visual object in a plane orthogonal to the image plate	middle-button held with mouse movement

Bevor ich zum Beispiel übergehe, möchte ich noch die **allgemeine Vorgehensweise**, für das Erstellen von Mausverhaltensklassen, erläutern.

1. **Erzeugen eines MouseBehavior-Objektes, welches das Verhalten, das auf das Objekt angewendet werden soll, enthält. Es gibt: MouseRotate, MouseTranslate und MouseZoom.**
2. **Setzen der TransformGroup, die die verschiedenen Transformationen im SceneGraph enthält.**
3. **Spezifizieren einer Grenze für das MouseBehavior-Objekt.**
4. **Einhängen des MouseBehaviors in den Szenengraphen.**

Diese Schritte erkläre ich nun anhand eines Beispieles (**MouseNavigationApp.java**). **In diesem Beispiel kann der Benutzer einen Würfel** anhand der Maus navigieren, d.h. **translieren, rotieren und zoomen**. Dies geschieht mittels Drücken eines Buttons und umherfahren der Maus in die gewünschte Richtung. Z.B. für das Zoomen, Mittlere Taste (3 Buttons) gedrückt halten. Wenn mit der Maus nun nach vorne gefahren wird, wird der Würfel vergrößert, wird die Maus nach hinten gefahren, wird der Würfel verkleinert.

Zu allererst einmal muss die **Klasse für die Mausnavigation**, für das Verhalten, **eingebettet werden**. Dies geschieht analog zur Tastaturnavigation. Dies ist:

```
import com.sun.j3d.utils.behaviors.mouse.*;
```

Wir erstellen die Klasse MouseNavigationApp und modellieren darauf den Szenengraph (**createSceneGraph**) für die Mausnavigation. Für die Navigation benötigen wir ein **BranchGroup**, welches die Transformationen aufnimmt, die **TransformGroup**, die die Transformationen bereitstellt und ein **BoundingSphere**, die eine Grenze liefert.

```
public class MouseNavigatorApp extends Applet      {
        public BranchGroup createSceneGraph(SimpleUniverse su)          {
                // Erstelle die Wurzel des Zweiggraphen
                BranchGroup objRoot = new BranchGroup();
                TransformGroup vpTrans = null;
                BoundingSphere mouseBounds = null;
```

Der nächste Schritt enthält, analog zur Tastaturnavigation in Kapitel 5.2, den wichtigsten Teil der Modellierungsmethode. **Nun wird die ViewPlatformTransformation mittels eines SimpleUniverse gespeichert**. Da das Objekt statisch bleibt, wird die ViewPlatform transformiert, d.h. verschoben, geändert. **Der neue Blick, die neue Ausrichtung des ViewPlatforms wird in der zuvor angelegten TransformGroup übernommen**, gespeichert. Der Würfel wird erstellt und der Szene als Kindobjekt hinzugefügt.

```
                // erhält die ViewPlatformTransformation von einem SimpleUniverse Objekt zurück
                vpTrans = su.getViewingPlatform().getViewPlatformTransform();
                // Würfel der Szene als Kindobjekt hinzufügen
                objRoot.addChild(new ColorCube(0.2));
```

Die Grenze für das MouseNavigatorBehavior-Objekt muss gesetzt werden.

```
                // setzt die Grenze für das MouseNavigatorBehavior-Objekt
                mouseBounds = new BoundingSphere(new Point3d(), 1000.0);
```

Nun **wenden wir** einfach **die oben beschriebenen vier Schritte zur Implementierung an**. Da es sich um drei verschiedene Behaviors handelt, muss die für jede Art der Transformation ein Paket programmiert werden, um alle drei Verhalten (Rotate, Zoom, Translate) anwenden zu können. Ich erkläre hier nur ein Paket, da dies für die restlichen zwei übernommen werden kann.

Schritt 1: Erzeugen eines MouseBehavior-Objektes, hier MouseRotate (Rotationen)

```
MouseRotate myMouseRotate = new
MouseRotate(MouseBehavior.INVERT_INPUT);
```

Schritt 2: Setzen der TransformGroup

```
myMouseRotate.setTransformGroup(vpTrans);
```

Schritt 3: Spezifizieren der Grenze für das MouseBehavior-Objekt

```
myMouseRotate.setSchedulingBounds (mouseBounds);
```

Schritt 4: Einhängen des MouseBehaviors in den Szenengraphen.

```
objRoot.addChild(myMouseRotate);

MouseTranslate myMouseTranslate = new
MouseTranslate(MouseBehavior.INVERT_INPUT);
myMouseTranslate.setTransformGroup(vpTrans);
myMouseTranslate.setSchedulingBounds(mouseBounds);
objRoot.addChild(myMouseTranslate);

MouseZoom myMouseZoom = new
MouseZoom(MouseBehavior.INVERT_INPUT);
myMouseZoom.setTransformGroup(vpTrans);
myMouseZoom.setSche dulingBounds(mouseBounds);
objRoot.addChild(myMouseZoom);

// Optimiert und kompiliert den Szenengraph vor
objRoot.compile();
return objRoot;
}
```

Das war der ganze Code zur Erzeugung von MouseBehaviors.
Wir benötigen noch den **Konstruktor zur Erzeugung der Szene**, dieser bleibt so wie ich ihn in Kapitel 5.2 vorgestellt habe. Nun noch die Hauptfunktion zum starten

des Programms und man kann per Mauseingabe den ColorCube, drehen, verschieben und zoomen.

```
        // Konstruktor zum initialisieren der Umgebung
        public MouseNavigatorApp() {
                // Layout festlegen, Zeichenfläche zuordnen
                setLayout(new BorderLayout());
                setLayout(new BorderLayout());
                Canvas3D canvas3D = new Canvas3D(null);
                add("Center", canvas3D);

                // Erstellung des SimpleUniverse als Root-Objekt der gesamten Szene
                SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
                BranchGroup scene = createSceneGraph(simpleU);

                // setzt das Objekt so, das der Betrachter es sehen kann
                simpleU.getViewingPlatform().setNominalViewingTransform();

                // Szenengraph in SimpleUniverse einfügen
                simpleU.addBranchGraph(scene);
        }

        // Hauptfunktion zum starten des Programms
        public static void main(String[] args) {
                Frame frame = new MainFrame(new MouseNavigatorApp(), 256, 256);
        }
}
```

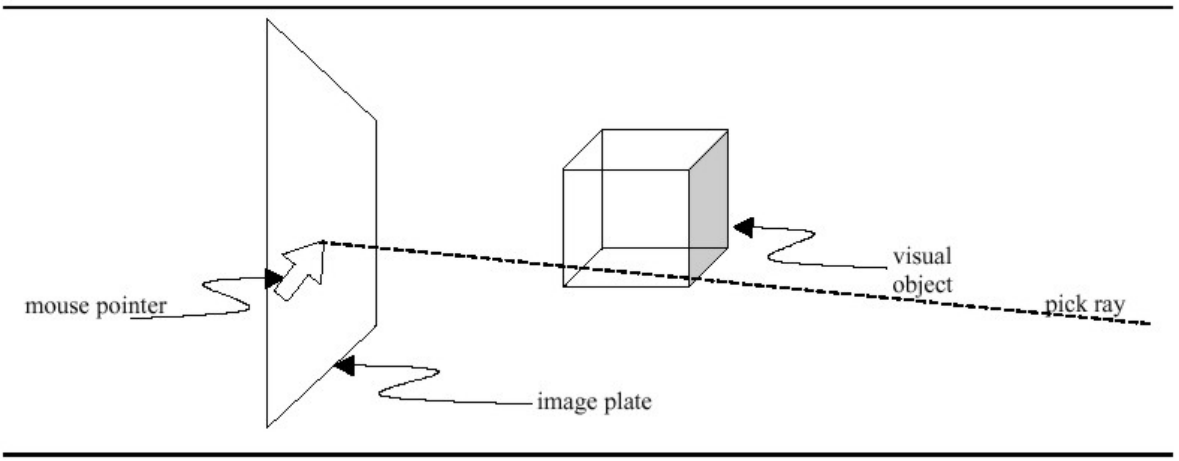
5.4 Picking

Was ist Picking?

Picking beschreibt den Wunsch des Benutzers, ein Objekt am Bildschirm anzuklicken und mir der Maus zu manipulieren. Dabei soll ein interaktives auswählen einer oder mehrer Objekte in der dargestellten Szene erfolgen können.

Ein Problem dabei ist, dass das die Auswahl eines Objektes zugeordneten Knotens nicht immer eindeutig ist. Deshalb wird eine so genannte Spur verwendet, die „**pick ray**“.

Im folgenden Bild ist der Ablauf dargestellt:



Der “**mouse pointer**” beschreibt den Zeiger der Maus, mit dem auf der **“image plate**”, d.h. Darstellfläche, das „**visual object**“, das Objekt, der Würfel, über die **„pick ray**“, Aufnehmspur aufgenommen und verschoben werden kann. Schneidet die **„pick ray**“ das Objekt, kann dies aufgenommen und manipuliert werden. Schneidet die „**pick ray**“ das Objekt nicht, so kann dieses nicht manipuliert werden.

Ein großes Problem bei Picking ist, dass es durch die vielen Manipulationen sehr rechenintensiv ist. Normalerweise müsste der Würfel aus sechs einzelnen Seiten, sechs einzelne **TransformGroups** gebildet werden, die wiederum einzeln angreifbar, **pickable** sein müssen. Daraus resultiert die Rechenintensivität. Java 3D stellt aber für diesen Fall, eine kleine Reihe von Möglichkeiten zur Verfügung, um

dies zu vereinfachen. Ein wichtiger Schritt hierbei sind die Fähigkeiten und Merkmale der Szenengraphknoten.
Durch das einbinden der Methode **setPickable()** in die Klasse sind die Szenengraphknoten greifbar. **Wird setPickable auf „false“ gesetzt, so sind keine Knoten, Unterknoten, Kindknoten greifbar. Der Nachteil hierbei ist allerdings, dass so keine Kindobjekte manipuliert werden können.**
Die zweite Möglichkeit ist die Fähigkeit des **ENABLE_PICKING_REPORTING**. Dieser Fall schließt einzig die Gruppenknoten mit ein. Wir dies also auf eine Gruppe angewandt, so kann diese Gruppe durch „picking“ manipuliert werden. **Gruppen die nicht zu der frei geschalteten Gruppe gehören bleiben außen vor**. Ebenso umgekehrt: **Kindobjekte werden hierbei nicht berücksichtigt**. Der Nachteil bei dieser Methode ist, es können schnell Fehler entstehen, wenn der Szenengraph nicht richtig eingestellt wurde.

Die Picking Utility Klasse beinhaltet drei Basisklassen zum picking.

1. **PickRotateBehavior** findet Verwendung bei der Rotation von Objekten mittels picking (**pick/rotate**)
2. **PickTranslateBehavior** wird angewandt wenn ein Objekt mittels direktem aufgreifen verschoben, transliert werden soll (**pick/translate**)
3. **PickZoomBehavior** zum vergrößern/-kleinern von Objekten mittels des picking (**pick/zoom**)

Bevor ein picking-Objekt im Szenengraph funktioniert, muss ein einziges das picking unterstützen. Dazu muss aber nur der folgende Code mit einbezogen werden.
Dieses Stück beinhaltet alles was notwendig ist, um picking in einem Java 3D Programm zu ermöglichen.

```
PickRotateBehavior pickRotate = new PickRotateBehavior(root, canvas, bounds);
root.addChild(pickRotate);
```

Das Objekt pickRotate überwacht jegliche Pickingaktionen auf dem Szenengraph. **Die root, Wurzel** liefert den **Bereich des Szenengraphes**, der überwacht werden soll. **Canvas ist der Bereich, in dem sich der Mauszeiger aufhält** und die **bounds, Grenze** ist die **Planungsgrenze des Picking-Behavior Objektes.**

Um nun ein Objekt per Picking zu manipulieren, zeige ich die **vier wesentlichen Vorgehenspunkte** bei der Programmierung einer solchen Klasse:

1. **Erstelle einen Szenengraph**
2. **Erstelle ein Picking-Verhaltensobjekt mit root, canvas und bounds merkmalen**
3. **Füge das Behavior-Objekt dem Szenengraph hinzu**
4. **Schalte die passenden Fähigkeiten für die Szenengraphobjekte, die Knoten ein**

Ich werde dies nun anhand eines Beispiels erkenntlich machen.

In meinem Beispiel kann ein Würfel anhand des Pickings aufgegriffen und verschiedenen Aktionen unterzogen werden, d.h. rotiert, transliert und vergrößert/-kleinert werden. Das Beispiel heißt: MousePickApp.java

Analog zu den vorhergehenden Beispielen müssen erst einmal die Basisklassen, die für das Pickingverhalten zuständig sind, importiert werden.

```
import com.sun.j3d.utils.behaviors.picking.*;
```

Wir erstellen die Klasse MousePickApp und modellieren darauf den Szenengraph (**createSceneGraph**) für das Picking. Für die Picking benötigen wir ein **BranchGroup**, welches die Transformationen aufnimmt und eine **TransformGroup**, die die Transformationen bereitstellt.

```
public class MousePickApp extends Applet        {
    // Modellierung der Szene mit einem KeyNavigator-Objekt
    public BranchGroup createSceneGraph(Canvas3D canvas)        {
        // Szenengraph erzeugen
        BranchGroup objRoot = new BranchGroup();
        TransformGroup objRotate = null;
        Transform3D transform = new Transform3D();
        BoundingSphere behaveBounds = new BoundingSphere();
```

Stelle nun die Transformationen bereit.

```
transform.setTranslation(new Vector3f(0.0f, 0.0f, 0.0f));
objRotate = new TransformGroup(transform);
```

An dieser Stelle werden jetzt die **Fähigkeiten der TransformGroup zugeordnet**. Es werden die Gruppenknoten mit eingeschlossen und das Picking erlaubt, Angeschalten.

```
// setze Fähigkeiten für die Knoten
objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
objRotate.setCapability(TransformGroup.ENABLE_PICK_REPORTING);
```

Noch den Würfel erzeugen und dem Szenengraph hinzufügen.

```
// Füge Würfel dem Szenengraph hinzu
objRoot.addChild(objRotate);
objRotate.addChild(new ColorCube(0.4));
```

Im nächsten Schritt **erzeugen** wird **die Pick-Behavior-Objekte**, also die Objekte, die die Pickingaktionen überwachen. Hierbei müssen drei **Objekte** angelegt werden, für jede Aktion eine **mit objRoot** (Wurzel des Szenengraphen), **canvas** (Zeichenfläche) und **behaveBounds** (Planungsgrenze).

```
// Erzeuge PickBehavior-Objekte, Mach das Objekt pickingfähig
// pick/rotate
PickRotateBehavior pickRotate = new PickRotateBehavior(objRoot, canvas,
behaveBounds);
// pick/zoom
PickZoomBehavior pickZoom = new PickZoomBehavior(objRoot, canvas,
behaveBounds);
// pick/translate
PickTranslateBehavior pickTranslate = new PickTranslateBehavior(objRoot,
canvas, behaveBounds);
```

Im letzen Schritt der createSceneGraph-Klasse müssen diese definierten Verhaltensklassen dem **Szenengraph hinzugefügt werden**.

```
        // Füge Behavior dem Szenegraph hinzu
        objRoot.addChild(pickRotate);
        objRoot.addChild(pickZoom);
        objRoot.addChild(pickTranslate);
```

Der Schluss wie gehabt.

```
        // Optimiert und kompiliert den Szenengraph vor
        objRoot.compile();
        return objRoot;
    }
```

Der Konstruktor zum initialisieren der Umgebung sowie die Hauptfunktion zum starten des Programms bleiben analog zu den vorhergehenden Beispielen und werden hier nicht mehr erläutert, aber zwecks der Vollständigkeit angeschrieben.

```
    // Konstruktor zum initialisieren der Umgebung
    public MousePickApp()        {
        // Layout festlegen, Zeichenfläche zuordnen
        setLayout(new BorderLayout());
        setLayout(new BorderLayout());
        Canvas3D canvas3D = new Canvas3D(null);
        add("Center", canvas3D);

        // Erstellung des SimpleUniverse als Root-Objekt der gesamten Szene
        SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
        BranchGroup scene = createSceneGraph(canvas3D);

        // setzt das Objekt so, das der Betrachter es sehen kann
        simpleU.getViewingPlatform().setNominalViewingTransform();
        // Szenengraph in SimpleUniverse einfügen
        simpleU.addBranchGraph(scene);
    }
```

```
        // Hauptfunktion zum starten des Programms
        public static void main(String[] args)          {
                Frame frame = new MainFrame(new MousePickApp(), 256, 256);
        }
}
```

So können Sie einfachst ein beliebiges Objekt per Picking verschieben, drehen, vergrößern und verkleinern.

Damit ist das Ende des Kapitels erreicht.

Ich hoffe Sie hatten Spaß und Freude an diesem praktischen Tutorial.

Ich wüsche Ihnen einen wunderschönen Tag.

Stefan Bohl

6. Quellenangaben

Java 3D Tutorial Kapitel 4 (Interaction) von Sun
http://java.sun.com

Fachhochschule Hagenberg, Österreich
http://webster.fhs-hagenberg.ac.at/staff/haller/mmp5_20012002/mmp5.htm