Die Virtual Reality Modeling Language (VRML, gesprochen Wörml) ist eine Skriptsprache, bzw. eine HTML-Erweiterung, die es erlaubt dreidimensionale Szenen darzustellen. Aktuell ist die Version 2.0 dieser Sprache.

Für die Darstellung der Szene benötigt man einen geeigneten Browser oder ein Plug-In für einen der Standardbrowser. Weit verbreitet auf diesem Sektor ist der Cortona-Player von Parallelgraphics (*http://www.parallelgraphics.com/products/cortona/*), der als Plug-In für Mozilla, Firefox bzw. den Internet-Explorer geeignet ist und für Linux auch FreeWrl.

Ein sehr einfaches Beispiel wäre das folgende:

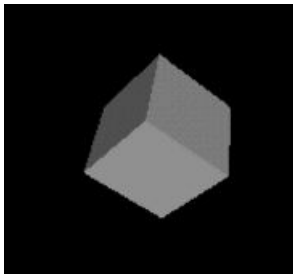
```
#VRML V2.0 utf8

Shape {
 appearance Appearance {
     material Material {}
 }
 geometry Box {}
}
```

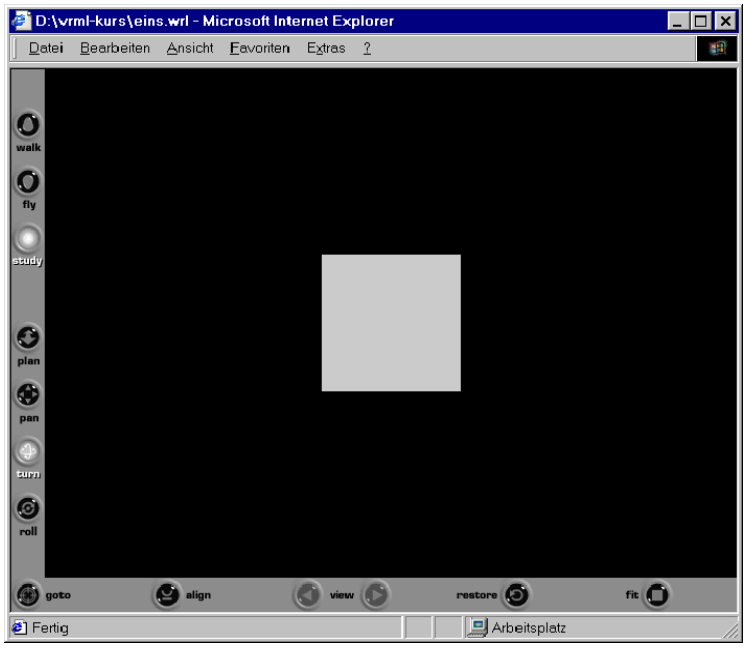
Die erste Zeile taucht in genau dieser Form in jedem VRML 2.0 Dokument auf. Es handelt sich um eine Kommentarzeile (‚#' am Zeilenbeginn), die angibt in welcher Weise der Rest des Dokumentes auszuwerten ist. Dieser Anfang ist von Scriptsprachen wie Perl bekannt. Auch wenn die Zeile als Kommentar gekennzeichnet ist, kommt es auf die exakte Schreibweise an!

Danach folgen dann als eigentlicher Inhalt die Objekte, die in VRML als *Knoten* bezeichnet werden. Hauptknoten ist hier *Shape*, der Knotentyp der für die Gestaltung von sichtbaren Objekten zuständig ist.
Vom Knotentyp Shape werden hier zwei Unterknoten mit angegeben, nämlich *appearance* und *geometry*. Für die Form des Objektes wird die *geometry Box* ausgewählt. Dabei handelt es sich um einen Würfel, mit der Kantenlänge mit der Kantenlänge 2.0, da wir keine anderen Angaben gemacht haben. Beim Erscheinungsbild wählen wir die vordefinierte Einstellung *Appearance* aus, ändern dort aber den *Subknoten material*, in dem wir die vordefinierte Einstellung *Material* angeben.

Im Browser sieht das folgendermaßen aus:

Genau in der Mitte des Bildschirmes ist der Würfel zu sehen. Am unteren und linken Rand des Bildes befinden sich die Navigationselemente des Players, die eine Bewegung sowohl des Objektes, als auch des Betrachters durch die Szene erlauben.





1. Grundlagen der VRML-Syntax

Die VRML-Syntax ist recht stark objektorientiert, obwohl dieser Begriff kaum irgendwo auftaucht. Bei VRML spricht man stattdessen viel von *Knoten*.
Es gibt nur wenige Datentypen und vordefinierte Bezeichner. Trotzdem werden Listings schnell sehr unübersichtlich, da die Zahl der Objekte (Knoten) in der Regel schnell groß wird. Man sollte daher von Anfang an auf eine übersichtliche Struktur und gute Kommentierung achten, sonst sind die Listings nicht wartbar.
Alle Zeichen, die nach „#“ folgen, werden als Kommentar betrachtet. Man kann also auch Kommentare an jede Codezeile anhängen.

Knoten
Zentraler Begriff ist der des Knotens. Bei den Knoten kann man drei Typen unterscheiden:
- Gruppenknoten
- Blattknoten
- Untergeordnete Knoten

1.1. Gruppenknoten

Ein Gruppenknoten kann andere Knoten enthalten, entweder wieder Gruppenknoten oder Blattknoten.
Zu den Gruppenknoten gehören:

Group
Der Gruppenknoten fasst mehrere Kindknoten zu einem Objekt zusammen. Das hilft bei der Strukturierung, hat aber auch Einfluss auf den Schwerpunkt des Objektes. Da man Objekte oft auch verschieben möchte, verwendet man meist den Transform-Knoten

Transform
Eine Erweiterung des Gruppenknotens, bei dem zusätzlich noch eine Verschiebung, eine Rotation bzw. eine Skalierung erfolgen kann.

```
#VRML V2.0 utf8
Background  {skyColor 1.0 1.0 1.0}

Transform {
  children [

    Shape {
      appearance Appearance {
         material Material {}
    }
    geometry Box {}
    }
  ]
  translation 2.0 0 0
}
```

Wichtigster untergeordneter Knoten ist hier *children*, eine Liste von Kindknoten. Falls die Liste, wie im vorliegenden Beispiel, nur aus einem Kindknoten besteht, können die eckigen Klammern entfallen.

Als weiterer Kindknoten ist hier *translation*, für die Verschiebung angegeben. Danach kommen die Werte, um die verschoben werden soll, in der Reihenfolge x-, y-, z-Verschiebung.

Von Bedeutung sind die untergeordneten Knoten:
- *children []*
- *translation 0.0 0.0 0.0* (x, y, z Verschiebung)
- *rotation 0.0 0.0 0.0 0.0* (x, y, z, Winkel Rotation)
- *scale 0.0 0.0 0.0* (x, y, z Skalierung)

Hinweis: die kursiv gesetzten Werteangaben stellen die Voreinstellungen dar. Wenn man mit der Voreinstellung zufrieden ist, dann kann der jeweilige untergeordnete Knoten weggelassen werden.

Inline
Mit diesem Knoten kann eine komplette VRML-Welt über die Angabe einer URL in die Szenerie integriert werden. Dies erlaubt einen stark modularen Aufbau, der die Übersichtlichkeit und Wartbarkeit der Listings erhöht.

```
Inline {
  url  "gyloh.wrl"
}
```

Man könnte hiermit auch Standardknoten wie den Hintergrund, die Beleuchtung und Höhenprofile aus den Listings heraushalten.

1.2. Blattknoten

Sie treten in einer Datei als eigenständige Knoten auf, oder sind Kindknoten in einem Gruppenknoten. Dazu gehören:

Shape
Ohne den Gestaltungsknoten Shape käme wohl kein Listing aus. Er verfügt über zwei untergeordnete Knoten, nämlich
- *appearance NULL*
- *geometry NULL*

die das Aussehen und die geometrische Form des Objektes beschreiben.
Beispiel siehe Seite 1.

DirectionalLight
Alle Lichtquellen sind eigenständige Blattknoten, sie sind nicht an irgendein geometrisches Objekt gebunden. Hier handelt es sich um paralleles gerichtetes Licht, das dem Sonnenlicht entspricht. Es stehen die untergeordneten Knoten
- *ambientIntensity 0.0* (0.0 bis 1.0, Licht von anderen Quellen)
- *color 1.0 1.0 1.0* (Farbe des Lichtes mit den drei Werten für r, g und b)
- *direction 0.0 0.0 -1.0* (Richtungsvektor des Lichtstrahles x, y, z)
- *intensity 1.0* (0.0 bis 1.0, Intensität)
- *on TRUE* (TRUE, FALSE, Zustand des Lichtschalters)

Gefunden habe ich für die resultierende Intensität, dass sie von der Farbe, der ambientIntensity und der intensity abhängt, aber mal als Summe, mal als Produkt.

PointLight
Licht, das sich punktförmig von einer Quelle aus in alle Richtungen ausbreitet. Es gibt die untergeordneten Knoten:
- *ambientIntensity 0.0* (0.0 bis 1.0, unklar, Licht von anderen Quellen)
- *attenuation 1.0 0.0 0.0* (Abschwächung des Lichtes)
- *color 1.0 1.0 1.0* (Farbe des Lichtes mit den drei Werten für r, g und b)
- *intensity 1.0* (0.0 bis 1.0, Intensität)
- *location 0.0 0.0 0.0* (Position der Lichtquelle x, y, z)
- *on TRUE* (TRUE, FALSE, Zustand des Lichtschalters)
- *radius 100* (Radius, in dem andere Knoten beleuchtet werden)

Für die Abschwächung des Lichtes gilt:

$$Intensität = \frac{1}{attenuation[0] + attenuation[1] \cdot radius + attenuation[2] \cdot radius}$$

Die Vorgabe ergibt eine Intensität von 1.0.

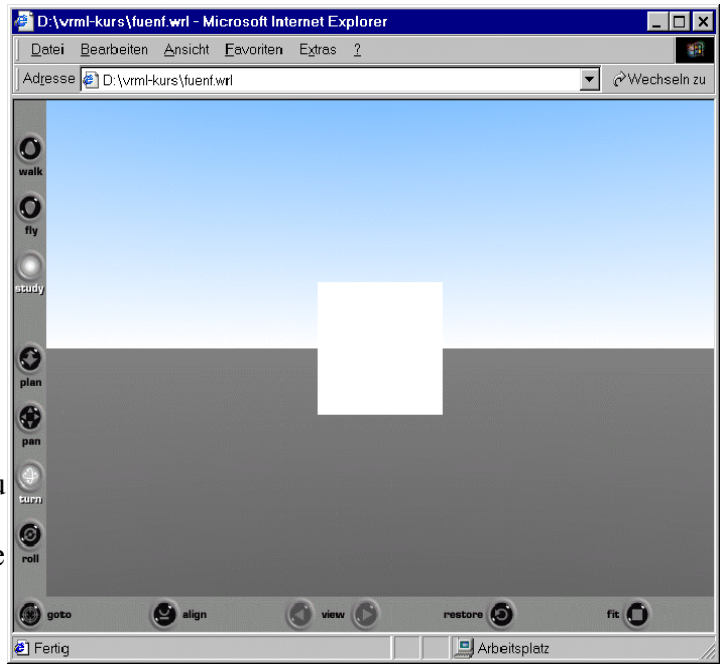
SpotLight
Kegelförmiger Lichtstrahl, der von einer punktförmigen Lichtquelle ausgeht.
- *ambientIntensity 0.0* (0.0 bis 1.0, unklar, Licht von anderen Quellen)
- *attenuation 1.0 0.0 0.0* (Abschwächung des Lichtes)
- *beamWidth 1.570796* (Öffnungswinkel des Lichtkegels mit konstanter Intensität 0.0 bis 1.570796=π/2=90°)
- *color 1.0 1.0 1.0* (Farbe des Lichtes mit den drei Werten für r, g und b)
- *cutOffAngle 0.785398* (äußerer Mantel des Lichtkegels 0.0 bis π/2)
- *direction 0.0 0.0 -1.0* (Richtungsvektor des Lichtstrahles x, y, z)
- *intensity 1.0* (0.0 bis 1.0, Intensität)
- *location 0.0 0.0 0.0* (Position der Lichtquelle x, y, z)
- *on TRUE* (TRUE, FALSE, Zustand des Lichtschalters)
- *radius 100* (Radius, in dem andere Knoten beleuchtet werden)

BackGround
Der normale voreingestellte Hintergrund ist schwarz und langweilig. Mit diesem Knoten kann man Farbverläufe für den Himmel und den Boden der virtuellen Welt angeben oder auch Grafiken für die Innenflächen der (würfelförmigen) Welt.
- *groundAngle []* (Liste von Winkelangaben)
- *groundColor []* (Liste von Farbwerten für den Boden)
- *skyAngle []* (Liste von Winkelangaben)
- *skyColor []* (Liste von Farbwerten für den Himmel)

Statt einer langwierigen Erklärung ein kleines Beispiel:

```
Background  {
  skyColor  [
     0.0 0.1 0.8,
     0.0 0.5 1.0,
     1.0 1.0 1.0
  ]
  skyAngle [0.785, 1.571]

  groundColor [
     0.0 0.0 0.0,
     0.3 0.3 0.3,
     0.5 0.5 0.5
  ]
  groundAngle [0.785, 1.571]
}
```

Für die Himmelsfarben werden hier definiert ein sehr dunkles Blau, ein helleres Blau und Weiß. Der leichte Grünanteil macht den Farbeindruck „natürlicher“. Von 0° (über dem Kopf) bis 45° verläuft die Farbe vom dunklen Blau bis zum hellen Blau. Von 45° bis 90° (Horizont) kommt dann der Verlauf vom hellen Blau bis zum Weiß.
Für die Grundfarben ist Schwarz, dunkles Grau und Grau definiert. Von 0° (unter den Füßen) bis 45° verläuft die Farbe von Schwarz zum dunklen Grau. Von 45° bis 90° (Horizont) verläuft dann die Farbe zum Grau.



Will man einfach nur einen weißen Hintergrund haben, statt des schwarzen, so kann man verkürzen zu:

```
Background  {skyColor 1.0 1.0 1.0}
```

Es gibt noch sechs weitere untergeordnete Knoten:

- *backUrl []*
- *bottomUrl []*
- *frontUrl []*
- *leftUrl []*
- *rightUrl []*
- *topUrl []*

Werte in diesen Listen sind jeweils Referenzen auf Grafikdateien, die als Panorama für die entsprechenden Innenflächen des Welt-Würfels dienen.

Viewpoint

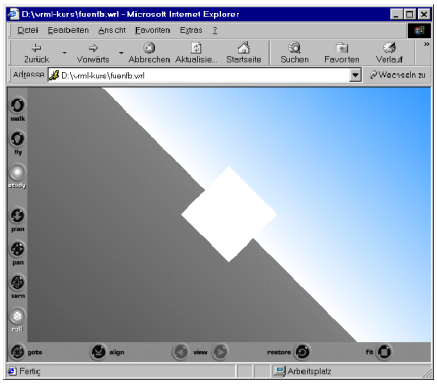
Beobachtungspunkte sind vordefinierte Positionen innerhalb der Welt, die über ein Auswahlmenü angesteuert werden können.

- *description “”* (Zeichenkette, die den Viewpoint beschreibt)
- *jump TRUE* (als aktuelle Benutzersicht einstellen)
- *orientation 0.0 0.0 1.0 0.0* (Drehachse 0 0 1 und Drehwinkel um die Achse)
- *position 0.0 0.0 10.0* (Betrachterposition innerhalb der Welt)

Wir erweitern das letzte Beispiel um:

```
Viewpoint {
 description "Meine Sicht"
 orientation 0 0 1 0.785
}
```

Es ergibt sich nebenstehendes Bild, eine Drehung der Welt um 45° um den Vektor in Blickrichtung.



1.3 Untergeordnete Knoten I (geometrische Knoten)

In dieser Rubrik finden sich u.a. auch die geometrischen Knoten, ohne die eine VRML-Welt langweilig wäre.

Box

Es handelt sich hierbei geometrisch betrachtet um einen Quader, dessen Zentrum sich im Ursprung des Koordinatensystems befindet. Texturen werden einzeln auf jede Seite des Quaders gemappt.

- *size 2.0 2.0 2.0* Die Abmessungen des Quaders

Cone

Ein Kegel, dessen Zentrum im Ursprung des Koordinatensystems liegt, die y-Achse ist seine zentrale Achse. Texturen werden unabhängig auf Mantel und Grundkreis gemappt

- *bottomRadius 1.0* Radius des Grundkreises
- *height 2.0* Höhe des Kegels
- *side TRUE* Wird der Kegelmantel dargestellt?
- *bottom TRUE* Wird der Grundkreis dargestellt?

Cylinder

Auch das Zentrum des Cylinders liegt im Ursprung und zentrale Achse ist wieder die y-Achse. Texturen werden getrennt auf die Teilflächen gemappt.

- *radius 1.0* Radius des Cylinders
- *height 2.0* Höhe des Cylinders
- *bottom TRUE* Wird die Grundfläche dargestellt?
- *top TRUE* Wird die Deckfläche dargestellt?
- *side TRUE* Wird der Mantel dargestellt?

Sphere

Die Kugel wird recht häufig in VRML Welten benötigt. Ich Zentrum befindet sich im Ursprung des Koordinatensystems und Texturen bedecken die gesamte Oberfläche.

- *radius 1.0* Der Radius

Text

Auch einfacher Text lässt sich in einer VRML-Welt als Objekt darstellen. Texturen werden über den gesamten Textbereich gemappt, aber nur auf den Buchstaben dargestellt.

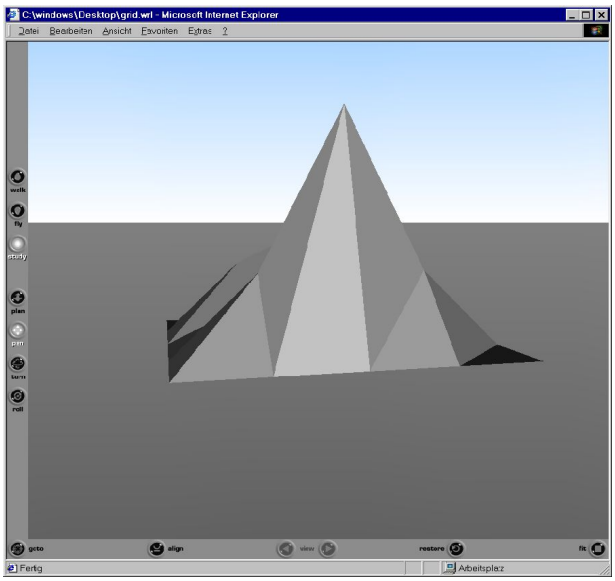
- *string []* Eine oder mehrere Zeichenketten in doppelten Anführungszeichen
- *fontStyle NULL* Hier kann ein FontStyle-Knoten angegeben werden, der Schriftart, -göße, ... festlegt
- *length 0.0* Länge der Zeichenkette. Bei Werten verschieden von 0.0 führt dies zu einer Skalierung
- *maxExtent 0.0* Obergrenze für die Ausdehnung der Zeichenketten des Knotens. Ein von 0.0 verschiedener Wert führt also gegebenenfalls zu einer Skalierung.

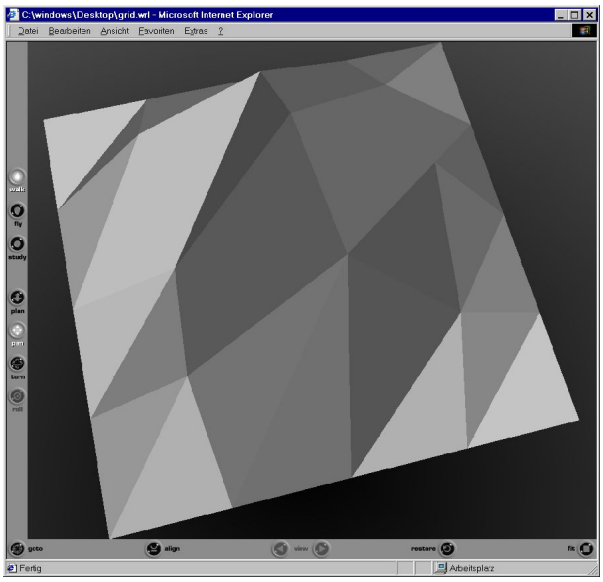
ElevationGrid
Hierbei handelt es sich um einen Knoten, mit dem man Geländeformationen darstellen kann. Benutzt wird dabei eine Art Schachbrett-Muster (muss nicht quadratisch sein) als Grundlage. Auf die einzelnen Felder können Quader mit frei wählbarer Höhe gestellt werden. Dadurch ergibt sich ein Höhenprofil, das noch zur Glättung interpoliert wird. Texturen bedecken die gesamte Fläche.

- *color NULL* Colorknoten mit Farbwerten pro Eckpunkt oder Flächenelement
- *height []* Zweidimensionale Liste mit den einzelnen Höhenwerten
- *xDimension 0* Anzahl der Felder in x-Richtung
- *xSpacing 0.0* Ausdehnung eines Feldes in x-Richtung
- *yDimension 0* Anzahl der Felder in y-Richtung
- *ySpacing 0.0* Ausdehnung eines Feldes in y-Richtung
- *zSpacing 0.0* Ausdehnung eines Feldes in z-Richtung

Statt einer langen Erklärung lieber gleich wieder ein Beispiel:

```
Shape {
 appearance Appearance {
   material Material {}
 }
 geometry ElevationGrid {
    xDimension 5
    zDimension 5
    xSpacing 1.0
    zSpacing 1.0
    height [
     0.0 0.0 0.0 0.0 0.0,
     0.0 1.0 1.5 1.0 0.0,
     0.0 0.5 2.5 1.5 0.0,
     0.0 1.0 3.0 1.0 0.0,
     0.0 0.0 0.0 0.0 0.0
    ]
 }
}
```





Bei den folgenden Knoten handelt es sich um solche, die man für kompliziertere geometrische Objekte benötigt, die sich nicht einfach aus den bisher genannten Grundobjekten aufbauen lassen

PointSet
Eine Liste bzw. Wolke von Punkten. Diese Listen werden selten unabhängig benötigt, sondern meistens für die Definition von Linien bzw. Flächen.

- *color NULL* Kann einen Color-Knoten enthalten, mit einer Liste von Farbwerten für jeden Punkt
- *coord NULL* Kann einen Coordinate-Knoten enthalten, mit einer Liste von Punkten.

IndexedLineSet
Eine Liste von Linien, die sich durch das systematische Verbinden von Punkten ergibt.-

- color NULL Kann einen Color-Knoten enthalten, mit einer Liste von Farbwerten für jede Linie
- *colorIndex []* Will man nicht für jeden Punkt eine andere Farbe benutzen, so kann man im Color-Knoten die Farben definieren und hier nur die Farbnummer den Linien zuordnen.
- *coord NULL* Kann einen Coordinate-Knoten enthalten, mit einer Liste von Punkten.
- *coordIndex []* Liste von Linienzügen, die über die Nummern der Punkte beschreiben werden. Ein Linienzug endet mit -1

Das folgende Listing beschreibt das Gittermodell eines Würfels:

```
Shape {
 appearance Appearance {
   material Material {}
 }
 geometry IndexedLineSet {
   coord Coordinate {
    point [
      -1.0  1.0  1.0,   #0 links oben vorn
      -1.0 -1.0  1.0,         #1 links unten vorn
       1.0 -1.0  1.0,         #2 rechts unten vorn
       1.0  1.0  1.0,         #3 rechts oben vorn
      -1.0  1.0 -1.0,         #4 links oben hinten
      -1.0 -1.0 -1.0,         #5 links unten hinten
       1.0 -1.0 -1.0,   #6 rechts unten hinten
       1.0  1.0 -1.0          #7 rechts oben hinten
    ]
   }
   coordIndex [
     0, 1, 2, 3, 0, -1,       # vorderes Quadrat
     4, 5, 6, 7, 4, -1,       # hinteres Quadrat
     0, 3, 7, 4, 0, -1,       # oberes Quadrat
     1, 2, 6, 5, 1, -1,       # unteres Quadrat
     3, 2, 6, 7, 3, -1,       # rechtes Quadrat
     0, 1, 5, 4, 0            # linkes Quadrat
   ]
 }
}
```

IndexedFaceSet
Hiermit kann man (nahezu) beliebige geometrische Objekte aus einzelnen Flächenstücken definieren. Die Verwendung von Texturen ist möglich, aber aufwendig steuerbar.

- *color NULL* Kann einen Color-Knoten enthalten, mit einer Liste von Farbwerten für jede Linie
- *colorIndex []* Will man nicht für jeden Punkt eine andere Farbe benutzen, so kann man im Color-Knoten die Farben definieren und hier nur die Farbnummer den Linien zuordnen.
- *coord NULL* Kann einen Coordinate-Knoten enthalten, mit einer Liste von Punkten.
- *coordIndex []* Liste von Linienzügen, die über die Nummern der Punkte beschreiben werden. Ein Linienzug endet mit -1
- *solid TRUE* Die Polygonstuktur wird bei TRUE als Festkörper behandelt, verdeckte Flächen werden nicht dargestellt. Will man das Objekt vernünftig drehen können, so muss der Wert auf FALSE gesetzt werden.

```
Shape {
 appearance Appearance {
   material Material {}
 }
 geometry IndexedFaceSet {

    solid FALSE

    coord Coordinate {
     point [
       -1.0  1.0  1.0,   #0 links oben vorn
       -1.0 -1.0  1.0,         #1 links unten vorn
        1.0 -1.0  1.0,         #2 rechts unten vorn
        1.0  1.0  1.0,         #3 rechts oben vorn
       -1.0  1.0 -1.0,         #4 links oben hinten
       -1.0 -1.0 -1.0,         #5 links unten hinten
        1.0 -1.0 -1.0,   #6 rechts unten hinten
        1.0  1.0 -1.0    #7 rechts oben hinten
     ]
    }
    coordIndex [
      0, 1, 2, 3, 0, -1,        # vorderes Quadrat
      4, 5, 6, 7, 4, -1,        # hinteres Quadrat
      0, 3, 7, 4, 0, -1,        # oberes Quadrat
      1, 2, 6, 5, 1, -1,        # unteres Quadrat
      3, 2, 6, 7, 3, -1,        # rechtes Quadrat
      0, 1, 5, 4, 0             # linkes Quadrat
    ]
 }
}
```

Ein so beschriebener Würfel ist von einem mit der geometry Box optisch nicht zu unterscheiden. Unterschiede treten erst dann auf, wenn es um das Anbringen von Texturen geht.

Extrusion
Geht man von einem Flächenstück aus und verschiebt dieses entlang einer Linie im Raum, so lässt sich damit ein Körper beschreiben. Der Begriff ist der Fertigungstechnik entlehnt, bei der Körper entstehen, indem der Kunststoff durch eine flache Form gepresst wird.
Man kann sich aber auch einen Spritzbeutel vorstellen, aus dem Sahne herausgedrückt wird. Als Flächenstück kommt ein beliebiger, nicht unbedingt geschlossener, Linienzug zum Einsatz (die Form der Düse). Die Linie für die Verschiebung (der Weg des Spritzbeutels) kann aus mehreren Zwischenschritten bestehen. Bei jedem Zwischenschritt kann das Flächenstück zusätzlich skaliert und gedreht werden.
Texturen werden wie beim Zylinder aufgebracht.

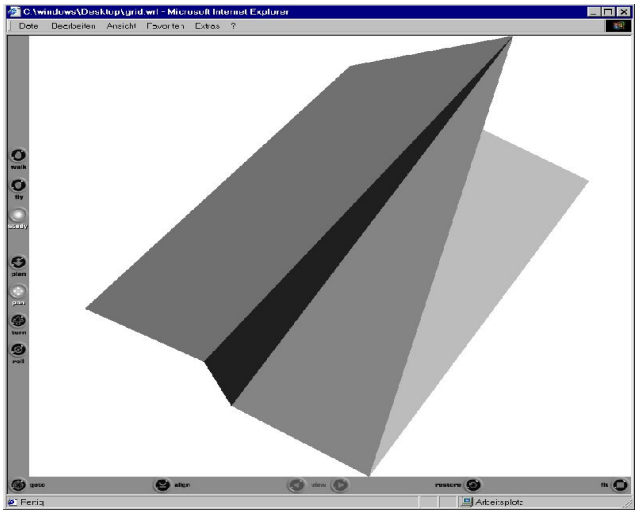
- *CrossSection [1 1, 1 -1, -1 -1, -1 1, 1 1]*
  Der Linienzug, hier ein Quadrat, der als Querschnitt für das Objekt dienen soll.
- *spine [0 0 0, 0 1 0]* Linienzug, der die Verschiebung beschreibt. Es können hier beliebig viele Punkte benutzt werden.
- *beginCap TRUE* Bei TRUE wird der Anfangsdeckel des Objektes mit dargestellt. Bei FALSE nicht.
- *endCap TRUE* Bei TRUE wird der Enddeckel des Objektes mit dargestellt. Bei FALSE nicht.
- *scale [1 1]* Legt für jeden Verschiebungsschritt die Skalierungsfaktoren fest.
- *convex TRUE* Bei TRUE ist der Körper nach außen gewölbt.
- *creaseAngle 0.5* Ist der Winkel zwischen zwei Flächenstücken kleiner als dieser Wert, so wird weich schattiert, was den Knick „wegbügelt“. Ansonsten wird hart schattiert.
- *solid TRUE* Bei TRUE gilt der Körper als massiv.
- *orientation [0 0 1 0]* Beschreibt eine Drehung für jeden der Verschiebungsschritte. Wird nur ein Wert angegeben, so gilt er für alle Schritte.

Ein paar kleine Bespiele:

```
Shape{
 appearance Appearance {
   material Material {}
 }
 geometry Extrusion {
   crossSection [ 1 2,
                  1 -2,
                 -1 -2,
                 -1 2,
                  1 2 ]
   spine [        0 0 0,
                  0 3 0 ]
 }
}
```

Beschreibt einen Quader mit den Abmessungen 2x4x3.
Bei dem folgenden Objekt wird die Fläche bei der Verschiebung (Extrusion) auch noch um 180° gedreht.

```
Shape{
 appearance Appearance {
   material Material {}
 }
 geometry Extrusion{
   crossSection [  1  2,
                   1 -2,
                  -1 -2,
                  -1  2,
                   1  2 ]
   spine  [        0 -1 0,
                   0 1 0   ]
   orientation[    0 1 0 0,
                   0 1 0 3.14]
 }
}
```



1.4. Untergeordnete Knoten II (Hilfsknoten)

Diese Knoten beschreiben Eigenschaften von geometrischen Objekten.

Color

Kann eine Liste von Farbangaben im RGB-Format beinhalten. Zu beachten ist, dass Texturen Vorrang besitzen gegenüber Farben.

- *color []* Farbwert im RGB-Format, drei Zahlen zwischen 0.0 und 1.0

Coordinate

Koordinaten von Punkten, wie man sie für PointSet, IndexedLineSet IndexedFaceSet benötigt.

- *point []* Liste von Punkten, die jeweils mit drei Gleitkommazahlen beschrieben werden.

Normal

Fast jeder Oberstufenschüler hat schon einmal mit Normalenvektoren zu tun gehabt. Mit Hilfe dieser Vektoren werden vom Browser z.B. die Brechungen und Schattierungen berechnet. Man kann die Normalenvektoren aber auch konkret angeben.

- *vector []* Der Normalenvektor sollte ein Einheitsvektor, also skaliert sein.

TextureCoordinate

Hiermit wird festgelegt, wie Texturen auf die Oberfläche von Objekten gemappt werden. Angegeben wird ein virtuelle, zweidimensionales Koordinatensystem, das mit dem s,t-Koordinatensystem des Objektes zur Deckung gebracht wird.

- *point []* Texturkoordinaten, bestehend aus zwei Gleitkommazahlen.

1.5. Untergeordnete Knoten III (Aussehen von geometrischen Objekten)

Entscheidend für das Aussehen von geometrischen Objekten ist der Appearance-Knoten, der über eine Zahl von spezifischen Unterknoten verfügt.

Appearance

Der Knoten der das Aussehen von geometrischen Objekten beschreibt.

- *material NULL* — Hier kann ein Material-Knoten eingebunden werden, der Eigenschaften wie Reflektion, Transparenz oder Leuchtkraft beschreibt.
- *texture NULL* — Legt Muster fest, die auf die Oberfläche des Objektes gemappt werden. Möglich sind Knoten vom Typ ImageTexture, MovieTexture und PixelTexture.
- *textureTransform NULL* — Beschreibt Operationen, die mit der Textur vor dem Mappen vorgenommen werden sollen.

Material

Der Material-Knoten beschreibt Objekt-Eigenschaften wie Reflektion, Transparenz oder Leuchtkraft.

- *ambientIntensity 0.2* — Beschreibt, wie stark der Körper diffuses Licht reflektiert.
- *diffuseColor .8 .8 .8* — Farbe des vom Objektreflektierten Lichtes.
- *emissiveColor 0 0 0* — Farbe des vom Objekt abgestrahlten Lichtes.
- *shininess 0.2* — Glanz des Objektes.
- *specularColor 0 0 0* — Reflektionswert für jede der drei Grundfarben
- *transparency 0* — Transparenz des Objektes. Für normale Fensterscheiben sind Werte ab 0.5 geeignet.

ImageTexture

Eine Grafikdatei im gif-, jpg- oder png-Format kann auf die Oberfläche eines Objektes gelegt werden. Transparente gif-Dateien werden richtig berücksichtigt.

- *url []* — Liste von Abbildungen, die als Textur benutzt werden sollen. Die erste gefundene Datei wird benutzt.
- *repeatS TRUE* — Soll die Textur in s-Richtung wiederholt werden?
- *repeatT TRUE* — Soll die Textur in t-Richtung wiederholt werden?

MovieTexture

Analog zur Grafikdatei kann auch eine Videodatei auf ein Objekt abgebildet werden. Dabei ist bisher nur das MPEG1-Format zulässig.

- *url []* — Liste von Filmen, die als Textur benutzt werden sollen. Die erste gefundene Datei wird benutzt.-
- *repeatS TRUE* — Soll die Textur in s-Richtung wiederholt werden?
- *repeatT TRUE* — Soll die Textur in t-Richtung wiederholt werden?
- *loop FALSE* — Bei TRUE wird das Abspielen des Videos unendlich wiederholt.
- *speed 1* — Wiedergabegeschwindigkeit für das Video.

PixelTexture
Einfache Punktmuster muss man nicht als gif- oder jpg-Datei angeben, sonder kann sie direkt in das Listing integrieren.

- *image 0 0 0* Folge von Zahlenwerten, die das Bild beschreibt. Die ersten beiden Zahlen beschreiben Breite und Höhe des Bildes. Die dritte Zahl beschreibt die Farbtiefe bzw. den Farbmodus. Danach kommt dann pro Bildpunkt 1 bis 4 Bytes je nach Farbtiefe. Das Muster wird normalerweise auf die Größe des Objektes gestreckt.
- *repeatS TRUE* Soll die Textur in s-Richtung wiederholt werden?
- *repeatT TRUE* Soll die Textur in t-Richtung wiederholt werden?

TextureTransform
Mit diesem Knoten kann man die Abbildung der Textur auf die Oberfläche des Objektes steuern. Die Textur kann skaliert, rotiert und verschoben werden.

- *center 0 0* Ursprungspunkt für Skalierung und Rotation
- *rotation 0* Winkel, der die Drehung der Textur beschreibt
- *scale 1 1* Skalierungsfaktoren in s- bzw. t-Richtung. Es wird aber nicht die Textur skaliert, sondern das virtuelle Koordinatensystem. Ein Wert von 2 halbiert also die Größe der Textur. Beim Skalieren sind die Repeat-Felder der Texturen wichtig dafür, ob Gekachelt wird oder nicht.
- *translation 0 0* Verschiebung der Textur.

2. Mehrfachverwendung von Knoten

Für größere Projekte ist es hilfreich, wenn man Objekte an einer Stelle definieren und dann innerhalb des Projektes mehrfach verwenden kann. Idealerweise trennt man sogar zwischen Definition und Verwendung in verschiedene Dateien.

2.1. DEF

Jeder Knoten innerhalb einer VRML-Definition kann benannt werden. Dazu setzt man das Schlüsselwort DEF gefolgt von dem Namen vor die Knotendefinition.

```
DEF kasten Shape {
 geometry Box {
   size 1 2 3
}
}
```

definiert ein Objekt „kasten“, das also ein Quader mit den Abmessungen 1x2x3 darstellt. Will ich dieses Objekt später noch einmal nutzen, so wird es mit

```
USE kasten
```

erneut instanziiert.

2.2. Prototypen

Effektiver als einfache Definitionen sind richtige Prototypen. Hier können sich die einzelnen Instanzen dann sogar in Werten unterscheiden, die als Parameter übergeben werden.
Zuerst ein kleines Beispiel:

```
PROTO  kasten [
 field        SFVec3f  size 1.0 1.0 1.0
 exposedField   SFColor  color  .8  .8  .8
]

{
  Shape {
    appearance Appearance {
       material Material {
         diffuseColor IS color
       }
    }
    geometry Box {
     size IS size
    }

 }
} # Ende PROTO Kasten
```

Definiert einen Prototyp „kasten“. Es wird eine Default-Größe und eine Default-Farbe festgelegt.
Instanziiert man „kasten“ folgendermaßen:

```
kasten {
 size 1 2 3
}
```

so wird ein grauer Kasten mit den Abmessungen 1x2x3 erzeugt. Die Prototyp-Definition allein erzeugt noch kein sichtbares Objekt.

Eine Prototypdefinition ist folgendermaßen aufgebaut:

```
PROTO <prototypname>
[
             <Schnittstelle>
]
{
       <Implementierung>
}
```

Im Interface-Teil werden angegeben die Schlüsselfelder

- exposedField
- field

gefolgt von einem Feldtyp, einem Feldnamen und dem Default-Wert. Der Unterschied zwischen field und exposedField besteht darin, dass auf ein exposedField auch Ereignisse einwirken können. Field kann immer benutzt werden, exposedField nur dann, wenn die Zuweisung des Wertes zu einer Größe erfolgt, die entsprechend definiert ist. Im obigen Beispiel ist die size-Feld einer Box nicht exposed, das color-Field von Material aber.

Die Default-Werte werden immer dann angenommen, wenn bei der Instanziierung keine Werte angegeben werden.

Im Implementierungsteil werden die Bezeichner aus dem Interfaceteil mittels *<feld> IS <parameter> eingesetzt.*

Ein sehr allgemeiner Prototyp zum Erzeugen von geometrischen Objekten könnte folgendermaßen aussehen:

```
PROTO  objekt [
 exposedField   SFVec3f  trans 0 0 0
 exposedField   SFColor  color  .8  .8  .8
 exposedField   MFString tex [ ]
 exposedField   SFNode geom Sphere {}
]

{
 Transform {
   children [
     Shape {
       appearance Appearance {
         material Material {
           diffuseColor IS color
         }
         texture ImageTexture {
           url IS tex
         }
       }
       geometry IS geom
     }
   ]
   translation IS trans
 }
} # Ende PROTO objekt
```

Eine Kugel kann dann einfach mittels:

```
objekt {
}
```

erzeugt werden und ein Würfel mittels:

```
objekt {
 geom Box {}
}
```

2.3. Externe Prototypen

Wie bereits beschrieben kann es sinnvoll sein Prototypen in einer eigenen Datei zu sammeln. Für diese Datei ist nichts besonders zu beachten. Will man aber einen Prototyp in eine andere Datei einbinden, so benötigt. man dazu zuerst eine EXTERNPROTO-Zeile, bevor man den Prototyp wie gewohnt nutzen kann.

```
EXTERNPROTO kasten [
 field        SFVec3f  size
 exposedField   SFColor  color ]
     "prototypen.wrl#kasten"
```

Hier muss im Prinzip der vollständige Interface-Teil wiedergegeben werden, bis auf die Angabe der Default-Werte.

3. Animationen

Mit VRML sind wir nicht nur in der Lage komplexe Objekte im Raum zu erstellen, sondern sie auch noch zu animieren.
Bevor wir die Animationen systematisch angehen ein einfaches Beispiel:

```
#VRML V2.0 utf8
# Animation

Background { skyColor 1 1 1 }

DEF Wuerfel Transform {
 children [
  Shape {
   appearance Appearance {
    material Material {}
   }
   geometry Box {}
  }
 ]
}

DEF Uhr TimeSensor {
 cycleInterval 5.0
 loop TRUE
}

DEF Interpolator PositionInterpolator {
 key [ 0.0 0.75 1.0 ]
 keyValue [
   0.0 0.0 0.0,
   0.0 2.0 0.0,
   0.0 0.0 0.0
 ]
}

ROUTE Uhr.fraction_changed TO Interpolator.set_fraction

ROUTE Interpolator.value_changed TO Wuerfel.set_translation
```

Es erscheint ein einfacher Würfel, der sich regelmäßig auf dem Bildschirm nach Oben und wieder in die Ausgangsposition bewegt.
Die Animation ersteht durch das Zusammenspiel von den drei Objekten

- Wuerfel
- Uhr
- Interpolator.

Der „Wuerfel“ ist das Objekt, das bewegt werden soll. Das Objekt „Uhr“ liefert den Zeittakt für die Animation, der komplette Durchlauf beträgt 5 Sekunden und wird unendlich oft wiederholt. Der Interpolator errechnet in Abhängigkeit von Eingabewerten Punkte. Bei der Eingabe 0.0 liefert er 0.0 0.0 0.0 zurück, bei 0.75 liefert er 0.0 2.0 0.0 zurück und bei der Eingabe 1.0 wieder die Ausgabe 0.0 0.0 0.0. Die Zwischenwerte errechnet er selbstständig.

Gekoppelt werden die drei Objekte über die zwei Routen. Die Uhr liefert über das Feld fraction_changed einen Wert zwischen 0.0 und 1.0 zurück. Dieser Wert wird an das Feld set_fraction des Interpolators übergeben, der daraufhin einen neuen Ausgabewert errechnet und über sein Feld value_changed an das Feld set_translation des Objektes „Wuerfel“ übergibt.

Die ungleichförmige Bewegung, abwärts geht es schneller, als aufwärts, rührt daher, dass im Feld „key“ ungleiche Abstände angegeben werden

3.1 Bestandteile einer Animation

Eine Animation besteht meist aus folgenden Komponenten:

- (geometrisches) Objekt bzw. zugehöriger Transform-Knoten
- Sensor (reagiert auf Ereignisse und löst Ereignisse aus)
- Interpolator (berechnet Zwischenwerte zu den Key-Frames)
- Routen zwischen den Objekten

Routen kann man nun zwischen benannten Objekten definieren, insofern ist es wichtig, dass alle Objekte mittels „DEF“ benannt werden.
Ziel bei geometrischen Objekten werden normalerweise die Felder

- translation
- rotation
- scale

sein. Es gibt noch einige Möglichkeiten mehr, man kann Kindknoten hinzufügen bzw. entfernen, darauf soll hier aber nicht eingegangen werden.

3.2 Routen

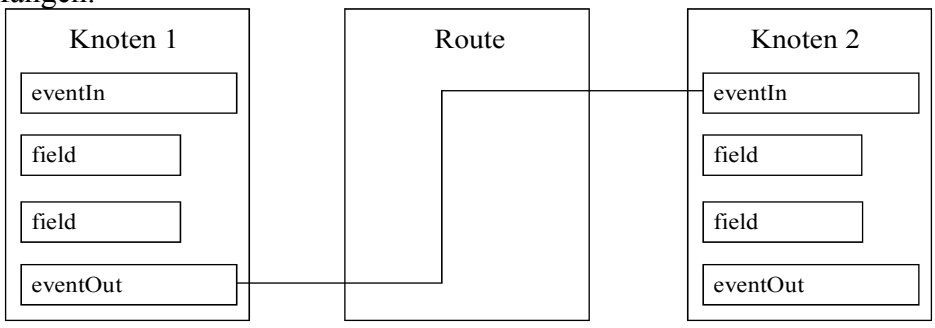
In VRML gehört zu einem Feld eines Objektes eine Zugriffsart. Mögliche Zugriffsarten sind:

- field
- eventIn
- eventOut
- exposedField

Bei der Zugriffsart „field“ gehören keinerlei Ereignisse zu dem Feld. Bei der Art „exposedField“ können sowohl Ereignisse empfangen, als auch ausgelöst werden. Bei den beiden anderen Arten ist jeweils nur eine Richtung möglich.

Kann das Feld „name“ Ereignisse empfangen, so wird der aktuelle Wert über „set_name“ geändert.
Kann das Feld „name“ Ereignisse auslösen, so steht das Ereignis in „name_changed“ zur Verfügung.

Eine Route verknüpft Felder die Ereignisse auslösen mit Feldern, die Ereignisse empfangen.



In der Route steckt implizit auch eine Bedingung. Nur wenn sich beim Ausgabefeld etwas ändert, dann erfolgt die Eingabe beim Eingabefeld.

Von einem Feld können mehrere Routen ausgehen, genauso können bei einem Feld mehrere Routen ankommen. Über einen einzigen Timesensor könnte man also mehrere Bewegungen auslösen.

3.3. Interpolatoren

Alle Interpolatoren haben den gleichen Aufbau:

- *key []* Liste mit Stützwerten (z.B. Zeitwerten) auf der Eingabeseite. Die Liste muss aufsteigend sortiert sein
- *keyValue []* Liste mit Stützwerten auf der Ausgabeseite. Die Art der Werte hängt vom Interpolator ab. Die Anzahl der Werte muss mit der im key-Feld übereinstimmen.
- *set_fraction* Eingabefeld
- *value_changed* Ausgabefeld

Die einzelnen Interpolatoren unterscheiden sich nur in der Art der Werte in der Liste mit den Stützwerten auf der Ausgabeseite.

PositionInterpolator

Die Werte im Feld keyValue sind vom Typ MFVec3F, also Positionsangaben.

```
keyValue [
      0.0 0.0 0.0,      # Grundposition
      0.0 0.2 0.0,      # nach Oben
      0.0 0.0 0.0       # zurück
]
```

OrientationInterpolator

Die Werte im Feld keyValue sind vom Typ MFRotation, beschreiben also Drehungen um eine Achse.

```
keyValue [
      0.0 0.0 1.0 0.00, # 0° Grundposition
      0.0 0.0 1.0 3.14, # 180°
      0.0 0.0 1.0 6.28  # 360° Grundposition (warum dürfte hier nicht
                                              wieder 0° stehen?)
]
```

ColorInterpolator

Die Werte im Feld keyValue sind vom Typ MFColor, also Farbangaben.

```
keyValue [
      0.0 1.0 0.0,      # Grün
      1.0 0.0 0.0,      # Rot
      0.0 1.0 0.0       # wieder Grün
]
```

Scalarinterpolator

Die Werte im Feld keyValue sind vom Typ MFFloat, also Gleitkommazahlen. Damit könnte man z.B. die Transparenz eines Objektes beeinflussen.

```
keyValue [
      0.0,  # undurchsichtig
      1.0,  # durchsichtig
      0.0   # undurchsichtig
]
```

Analog zum *PositionInterpolator* gibt es noch *CoordinationInterpolator* und *NormalInterpolator*.

3.4. Sensoren

Einen Sensor haben wir bereits kennen gelernt, den TimeSensor. Daneben gibt es eine Zahl von weiteren Sensoren:

- TimeSensor Zeitgeber
- TouchSensor Reagiert auf Berührungen mit dem Mauszeiger
- VisibilitySensor Registriert die Sichtbarkeit eines Objektes durch den Betrachter
- ProximitySensor Reagiert auf Annäherung des Betrachters
- PlaneSensor dient derVerschiebung eines Objektes in der xy-Ebene durch Ziehen mit der Maus
- SphereSensor Rollen eines Objektes um den Ursprung des Koordinatensystems durch Ziehen mit der Maus
- CylinderSensor Rotation um die y-Achse durch Ziehen mit der Maus

TimeSensor

Ein eingebauter Timer, der auf der Systemuhr basiert.

Felder

- *cycleInterval 1* Zeitintervall für einen vollständigen Durchlauf
- *enabled TRUE* Ist der Timer aktiv?
- *loop FALSE* Bei TRUE wird der Timer nach jedem Durchlauf neu gestartet
- *startTime 0* Startzeit für die Aktivität des Sensors
- *stopTime 0* Zeitpunkt zu dem der Timer seine Aktivität beendet

Ereignisse

- *cycleTime* Wird mit jedem neuen Durchlauf erzeugt
- *fraction_changed* Hier werden kontinuierlich Zeitwerte zwischen 0 und 1 augegeben

TouchSensor

Reagiert auf Berührungen des Objektes durch die Maus

Felder

- *enabled TRUE* Bei TRUE reagiert der Sensor auf Berührungen

Ereignisse

- *isOver* Wird TRUE, wenn der Zeiger das Objekt berührt, ansonsten FALSE
- *isActive* Wird TRUE, wenn die Linke Maustaste während einer Berührung gedrückt ist, ansonsten FALSE.

Das folgende Listing, eine Erweiterung des Beispiels aus 3., koppelt beide Sensoren miteinander:

```
#VRML V2.0 utf8
# Animation 2

Background { skyColor 1 1 1 }
DEF Wuerfel Transform {
 children [
   Shape {
     appearance Appearance {
        material Material {}
     }
     geometry Box {}
   }
 ]
}
```

```
DEF Interpolator PositionInterpolator {
 key [ 0.0 0.75 1.0]
 keyValue [
   0.0 0.0 0.0,
   0.0 2.0 0.0,
   0.0 0.0 0.0
 ]
}

DEF Uhr TimeSensor {
 enabled FALSE
 cycleInterval 5.0
 loop TRUE
}

DEF Schalter TouchSensor {}

ROUTE Schalter.isActive TO Uhr.set_enabled

ROUTE Uhr.fraction_changed TO Interpolator.set_fraction
ROUTE Interpolator.value_changed TO Wuerfel.set_translation
```

Die Erweiterungen sind fett hervorgehoben. So wird die Animation erst aktiv, wenn man mit der Maus auf den Würfel zeigt und die linke Maustaste drückt.
Dass die Uhr schon vorher lief kann man daran erkennen, das der Würfel meist einen Sprung auf eine Zwischenposition macht. Der Timer lief zwar schon vorher, hat aber keine Ereignisse geliefert.

VisibilitySensor
Ermittelt, ob ein quaderförmiger Bereich für den Betrachter ganz oder teilweise sichtbar ist. Der Abstand spielt keine Rolle. Hiermit kann man Animationen abschalten, wenn der Betrachter sie nicht wahrnehmen kann.
Felder
- *center 0 0 0* Mittelpunkt des quaderförmigen Bereiches
- *enabled TRUE* Nur bei TRUE werden Ereignisse geliefert
- *size 0 0 0* Ausdehnungen des Quaders

Ereignisse
- *isActive* Liefert TRUE beim Sichtbarwerden des Bereiches, sonst FALSE.

ProximitySensor
Ermittelt, ob der Betrachter in einen quaderförmigen Bereich eingedrungen ist.
Felder
- *center 0 0 0* Mittelpunkt des quaderförmigen Bereiches
- *enabled TRUE* Nur bei TRUE werden Ereignisse geliefert
- *size 0 0 0* Ausdehnungen des Quaders

Ereignisse
- *isActive* Liefert TRUE beim Eindringen des Betrachters in den Bereich, FALSE beim Verlassen.
- *position_changed* Liefert Positionsveränderungen des Betrachters im Bereich des Sensors (Typ SFVec3f)
- *orientation_changed* Liefert Änderungen der Betrachterorientierung im Bereich des Sensors (Typ SFRotation).

Die letzten drei Sensoren dienen dazu mit der Maus Objekte im Raum zu bewegen. Dazu werden unterschiedliche Freiheitsgrade für das jeweilige Objekt zur Verfügung gestellt.

PlaneSensor
Mit der Maus wird das Objekt in der xy-Ebene verschoben. In einer VRML-Welt könnte man hiermit Schiebetüren realisieren, die vom Benutzer geöffnet bzw. geschlossen werden können.
Felder

- *autoOffset TRUE* Bei der Einstellung TRUE beginnt eine neue Verschiebung an der aktuellen Position, bei FALSE an der Ursprungsposition. Der Wert wird in offset gespeichert.
- *offset 0 0 0* Wenn autoOffset auf TRUE gestellt ist, wird hier die relative Verschiebung zum Ursprung gespeichert.
- *enabled TRUE* Nur bei TRUE werden Ereignisse geliefert
- *maxPosition -1 -1* Rechte obere Ecke des gedachten Rechtecks, das den Bewegungsbereich beschränkt. Sind die Werte kleiner als die von minPosition, so ist die Translation nicht begrenzt.
- *minPosition 0 0* Linke untere Ecke des Rechtecks, das den Bewegungs bereich beschränkt.

Ereignisse

- *isActive* Liefert TRUE beim Drücken der Maustaste über dem Objekt, FALSE beim Loslassen.
- *trackPoint_changed* Liefert Positionsveränderungen des Mauszeigers (Typ SFVec3f). Die Beschränkungen durch das Rechteck spielen hier keine Rolle.
- *translation_changed* Liefert Positionsveränderungen des Objektes (Typ SFVec3f). Die Beschränkungen durch das Rechteck spielen hier eine Rolle.

Unser bisheriges Beispiel hierauf angepasst sieht folgendermassen aus:

```
#VRML V2.0 utf8
# Animation 2

Background { skyColor 1 1 1 }

DEF Wuerfel Transform {
 children [
   Shape {
    appearance Appearance {
      material Material {}
    }
    geometry Box {}
   }
 ]
}

DEF Verschiebe PlaneSensor {
 maxPosition 1 1
 minPosition -1 -1
}

ROUTE Verschiebe.translation_changed TO Wuerfel.set_translation
```

CylinderSensor
Dieser Sensor erlaubt nur Rotationen um die y-Achse. Hiermit lassen sich „normale“ Türen realisieren, die vom Benutzer geöffnet und geschlossen werden können.
Felder
- *autoOffset TRUE* Bei der Einstellung TRUE beginnt eine neue Drehung an der aktuellen Position, bei FALSE an der Ursprungsposition. Der Wert wird in offset gespeichert.
- *offset 0* Wenn autoOffset auf TRUE gestellt ist, wird hier die relative Drehung zum Ausgangswert gespeichert.
- *enabled TRUE* Nur bei TRUE werden Ereignisse geliefert
- *maxAngle -1* Größter erlaubter Drehwinkel.. Ist der Wert kleiner als der von minAngle, so ist die Rotation nicht begrenzt.
- *minPosition 0 0* Minimaler Wert für den Drehwinkel.

Ereignisse
- *isActive* Liefert TRUE beim Drücken der Maustaste über dem Objekt, FALSE beim Loslassen.
- *trackPoint_changed* Liefert Positionsveränderungen des Mauszeigers (Typ SFVec3f). Die Beschränkungen spielen hier keine Rolle.
- *rotation_changed* Liefert Positionsveränderungen des Objektes (Typ SFRotation). Die Beschränkungen durch das Rechteck spielen hier eine Rolle.

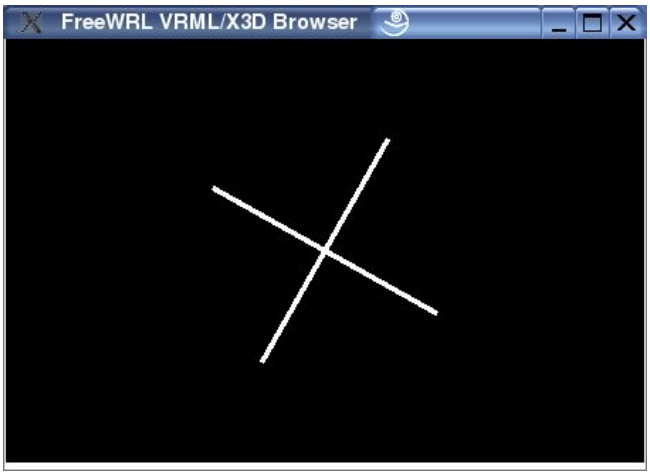
SphereSensor
Erlaubt die freie Drehung eines Objektes um den Ursprung des lokalen Koordinatensystems.
Felder
- autoOffset TRUE Bei der Einstellung TRUE beginnt eine neue Drehung an der aktuellen Position, bei FALSE an der Ursprungsposition. Der Wert wird in offset gespeichert.
- *offset 0 1 0 0* Wenn autoOffset auf TRUE gestellt ist, wird hier die relative Drehung zum Ausgangswert gespeichert.
- *enabled TRUE* Nur bei TRUE werden Ereignisse geliefert

Ereignisse
- *isActive* Liefert TRUE beim Drücken der Maustaste über dem Objekt, FALSE beim Loslassen.
- *trackPoint_changed* Liefert Positionsveränderungen des Mauszeigers (Typ SFVec3f). Die Beschränkungen spielen hier keine Rolle.
- *rotation_changed* Liefert Positionsveränderungen des Objektes (Typ SFRotation). Die Beschränkungen durch das Rechteck spielen hier eine Rolle.

3.5. Beispiele

Zum Abschluss ein paar Beispiele für grundlegende Animationseffekte.

Ein einfacher Propeller:

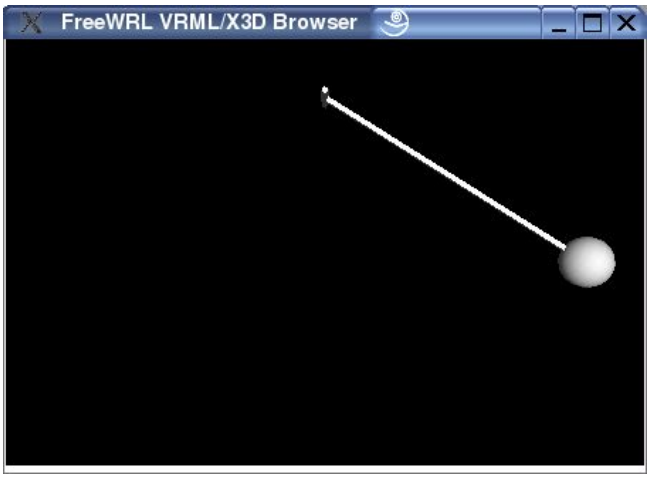



```
#VRML V2.0 utf8
DEF Propeller Transform {
  children [
    Shape {
      appearance Appearance {
        material Material {}
      }
      geometry Box {size 5 0.1 0.1 }
    }
    Shape {
      appearance Appearance {
        material Material {}
      }
      geometry Box {size 0.1 5 0.1 }
    }
    Shape {
      appearance Appearance {
        material Material {}
      }
      geometry Box { size 0.1 0.1 1 }
    }
  ]
}
DEF Uhr TimeSensor {
  cycleInterval 2.0
  loop TRUE
}
DEF Interpolator OrientationInterpolator {
  key [0.0 0.5 1.0]
  keyValue [
    0 0 1 0.0,
    0 0 1 1.57
    0 0 1 3.14
  ]
}
ROUTE Uhr.fraction_changed TO Interpolator.set_fraction
ROUTE Interpolator.value_changed TO Propeller.set_rotation
```

Ein blinkendes Blaulicht

```
#VRML V2.0 utf8

Transform {
  children [
    Shape {
      appearance Appearance {
        material DEF Meinmat Material {
          diffuseColor 0.8 0.8 0.8
        }
      }
      geometry Cylinder {
        radius 2
       height 1
      }
    }
  ]
  translation 0 0 0
}
DEF Uhr TimeSensor {
  cycleInterval 1.0
  loop TRUE
}
DEF Interpolator ColorInterpolator {
  key [ 0 0.5 1 ]
  keyValue [ 0 0 0,
             0 0 1,
             0 0 0]
}
ROUTE Uhr.fraction_changed TO Interpolator.set_fraction
ROUTE Interpolator.value_changed TO Meinmat.set_diffuseColor
ROUTE Interpolator.value_changed TO Meinmat.set_emissiveColor
```

Ein Pendel




```
#VRML V2.0 utf8

DEF Propeller Transform {
  children [
    Shape {
      appearance Appearance {
        material Material {}
      }
      geometry Box {
        size 0.1 0.1 1
      }
    }
    Transform {
      children [
        Shape {
          appearance Appearance {
            material Material {}
          }
          geometry Box {
            size 0.1 6 0.1
          }
        }
      ]
      translation 0 -3 0
    }
    Transform {
      children [
        Shape {
          appearance Appearance {
            material Material {}
          }
          geometry Sphere {
            radius 0.5
          }
        }
      ]
      translation 0 -6 0
    }
  ]
  translation 0 3 0
}

DEF Uhr TimeSensor {
  cycleInterval 5.0
  loop TRUE
}

DEF Interpolator OrientationInterpolator {
  key [0.0 0.25 0.5 0.75 1.0]
  keyValue [
    0 0 1 0.0,
    0 0 1 1.2,
    0 0 1 0.0,
    0 0 1 -1.2,
    0 0 1 0
  ]
}

ROUTE Uhr.fraction_changed TO Interpolator.set_fraction
ROUTE Interpolator.value_changed TO Propeller.set_rotation
```