13. Grundzüge von VRML

VRML:

Virtual Reality Markup Language
→ Virtual Reality Modelling Language

- Beschreibungssprache für VR-Szenen
- (kein Softwaresystem)
- Umsetzung durch *VRML-Browser*
- VRML-Szenen in ASCII-Dateien abgelegt
- (erweiterte) XML-formatierte Version: **X3D**
- Standardisierung: Dieselbe Szene durch verschiedene Browser darstellbar

VRML- (und X3D-) Umfang:

3D-Grafik
+ Dynamik (Animation)
+ Interaktion
+ Klang
+ Internetfähigkeit (VRML-Browser als Plugin von Webbrowsern)

Versionen: VRML 1.0, 2.0, 97

Sprachspezifikation:
http://www.web3d.org/x3d/specifications/vrml

Features:

Geometriebeschreibungen	Grundprimitive (Quader, Zylinder, Kegel, Kugel) polygonale Objekte elevation grids (Terrain-Gitter) extrudierte Objekte
Materialbeschreibungen	Farbe Schattierungen
Beleuchtungsbeschreibungen	
Texturbeschreibungen	
Textbeschreibungen	
Transformationen	Translationen Rotationen Skalierungen allgemeine Transformationen
Animationen	Zeit Interpolation
Hintergrund und Nebel	
Interaktionsbeschreibungen	
Multimediabeschreibungen	Video 3D-Klang
Strukturbeschreibungen	Gruppierung Prototypen
Detaillierungsbeschreibungen	LOD = level of detail
Kamerabeschreibungen	Viewpoints
Hypermediabeschreibungen	Anchors
Beschreibung von Programmierlogik	Java Javascript VRMLscript

- VRML kann mit immersiven Technologien umgesetzt werden
- Einbindung von Programmen beliebiger Komplexität
- Programme können in VRML-Szenen eingreifen (Anwendungen z.B.: Mehrbenutzersysteme, Datenbank-Anbindung, Simulationen)

Geschichte

- Erste Ideen zu virtuellen Welten im WWW von Mark Pesce und Tony Parisi, 1994
- im Rahmen der ersten internationalen Konferenz über das WWW (Mai 1994 bei CERN in Genf) wird eine Sitzung zu einer geplanten "Virtual Reality Markup Language" abgehalten (wichtige Rolle: Tim Berners-Lee, der Entwickler von HTML); das Akronym "VRML" wird geprägt
- Oktober 1994: VRML 1.0 wird von Tony Parisi und Gavin Bell präsentiert, basiert nach Entscheidung durch Internet-Abstimmung (Mailingliste) auf Open Inventor von SGI. Beginn der Unterstützung durch SGI, Netscape und Microsoft
- VRML Architecture Group (VAG; 8 technische Experten) wird im Anschluss an die SIGGRAPH'95 gegründet (August 1995)
- Anfang 1996 Aufruf zu Vorschlägen für VRML 2.0 durch die VAG
- August 1996: VRML 2.0-Spezifikation wird auf der SIGGRAPH'96 vorgestellt, nach offener Internet-Abstimmung auf Moving Worlds von SGI basierend; Gründung des VRML-Konsortiums (Vertreter von Firmen, Forschungseinrichtungen und Universitäten)
- Anfang 1997: Beginn der ISO-Standardisierung mit der Erarbeitung von VRML 97
- Ende 1997: VRML 97 wird standardisiert als ISO/IEC DIS 14772-1
- Weiterentwicklung: X3D

VRML-Browser

BS Contact VRML (Bitmanagement Software)
freie Testversion
http://www.bitmanagement.de

Cortona (Parallel Graphics), Download:
http://www.parallelgraphics.com/cortona/

Cosmo Player (wird nicht mehr weitergepflegt)
World View
....

VRML-Browser als Plugin zu MS Internet Explorer bzw. anderen Browsern

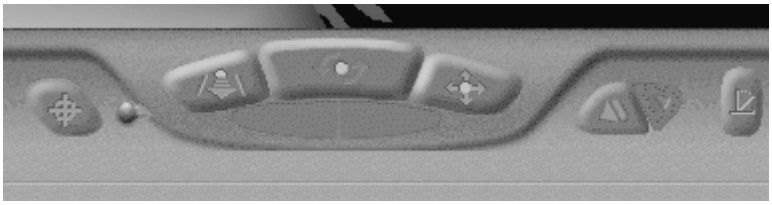
Jeder VRML-Browser stellt Navigationshilfen für den 3D-Raum zur Verfügung
Anordnung und Bezeichnungsweisen browserspezifisch, aber ähnliche Grundfunktionen.
z.B. für Cosmo Player:

6 Haupt-Navigationsmodi

Gehen Neigen Gleiten	Go Tilt Slide	Bewegungen in der horizontalen Ebene Änderung des Blickwinkels Bewegungen in der vertikalen Ebene
Rotieren Schwenken Zoomen	Rotate Pan Zoom	lässt die Szene rotieren Eigenbewegung um eine Szene, die im Blickfeld bleibt Änderung des Abstands von der Szene

Die eigentliche Navigation erfolgt mit der Maus. Die Form des Mauscursors zeigt an, welcher Navigationsmodus aktiv ist. Zwischen der oberen und der unteren Dreiergruppe wird mit einem virtuellen Hebel umgeschaltet ("Steuerelemente ändern"; auch durch Hotkeys (Tastatur) möglich).

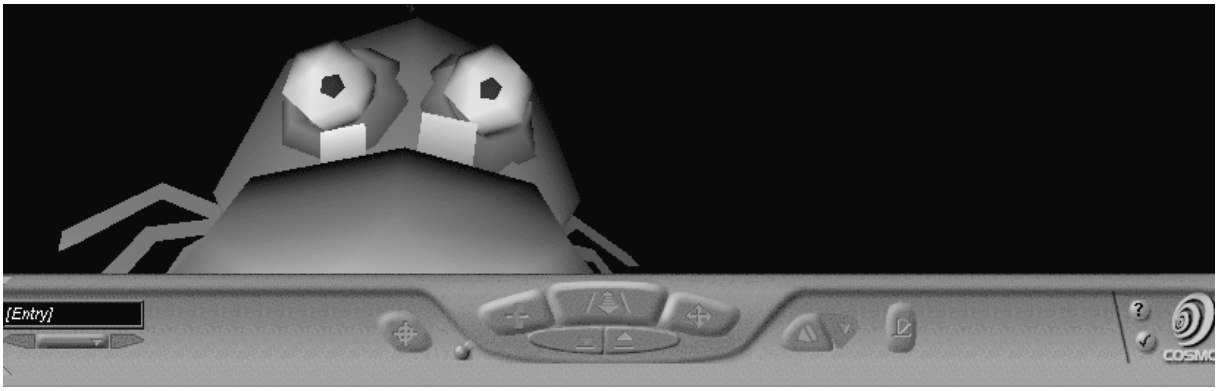
Schaltfläche mit Funktionen der ersten Gruppe



Schaltfläche mit Funktionen der zweiten Gruppe

Zusätzliche Sonderfunktionen:

Gravitation	Gravity	in hügeliger Landschaft am Boden bleiben (man folgt den Höhen und Tiefen der Oberfläche)
Treiben	Float	erlaubt, zu fliegen
Suchen	Seek	durch Anklicken eines Objekts bewegt man sich an dieses heran
Bewegung rückgängig machen	Undo move	
Bewegung wiederherstellen	Redo move	
Richten	Straighten	man wird in eine Orientierung senk-recht zur Horizontalebene gebracht
Viewpoint		man kann Blickpunkte speichern und mit Namen versehen
Preferences		Aktivierung des Voreinstellungs-Menüs
COSMO		Hyperlink zur Webseite von Cosmo
Help		Online-Hilfe (HTML)
Warnleuchten		werden aktiv bei Fehlern, *bei Anklicken erscheint die Fehlermeldung*



beim Cortona Viewer:
z.T. andere Namen für dieselben oder ähnliche Funktionen

walk, fly, study
plan, pan, turn, roll
goto, align, restore, fit

(interaktiv testen!)

Cortona Schaltleisten:

VRML-Dateien

Endung .wrl ("world")
ASCII-Datei (genauer: ab V. 2.0 UTF-8 Zeichensatz gem. ISO 10646-1:1993)

Aufbau einer VRML-Datei:

- Header (Version und Zeichensatzangabe, obligatorisch)
- Zeilenkommentare
- VRML-*Knoten*
- innerhalb der Knoten-Spezifikationen: *Felder* (= festgelegte Attributierungen von Knoten, denen Werte zugewiesen werden)
- PROTO-Statements
- ROUTE-Statements

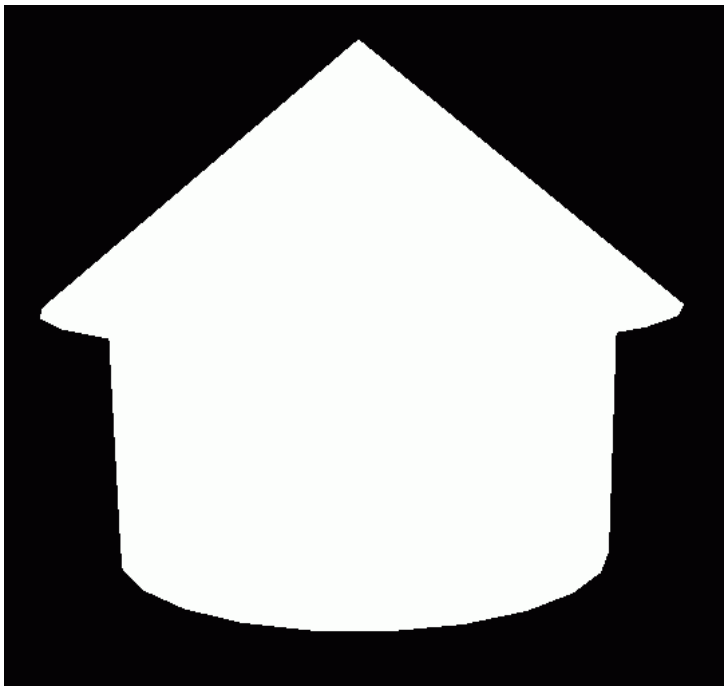
Zuordnung von Feldangaben zu Knoten mit geschweiften Klammern { ... }

Gruppierung von Knoten durch Gruppen-Knoten, "Kinder" in eckigen Klammern [...]

Beispiel einer VRML2.0 - Datei:

```
#VRML V2.0 utf8
Cylinder
    {
    height 2.0
    radius 2.0
    }
Transform
    {
    translation 0.0 2.0 0.0
    children
        [
        Cone
            {
            bottomRadius 2.5
            }
        ]
    }
```

Ergebnis:



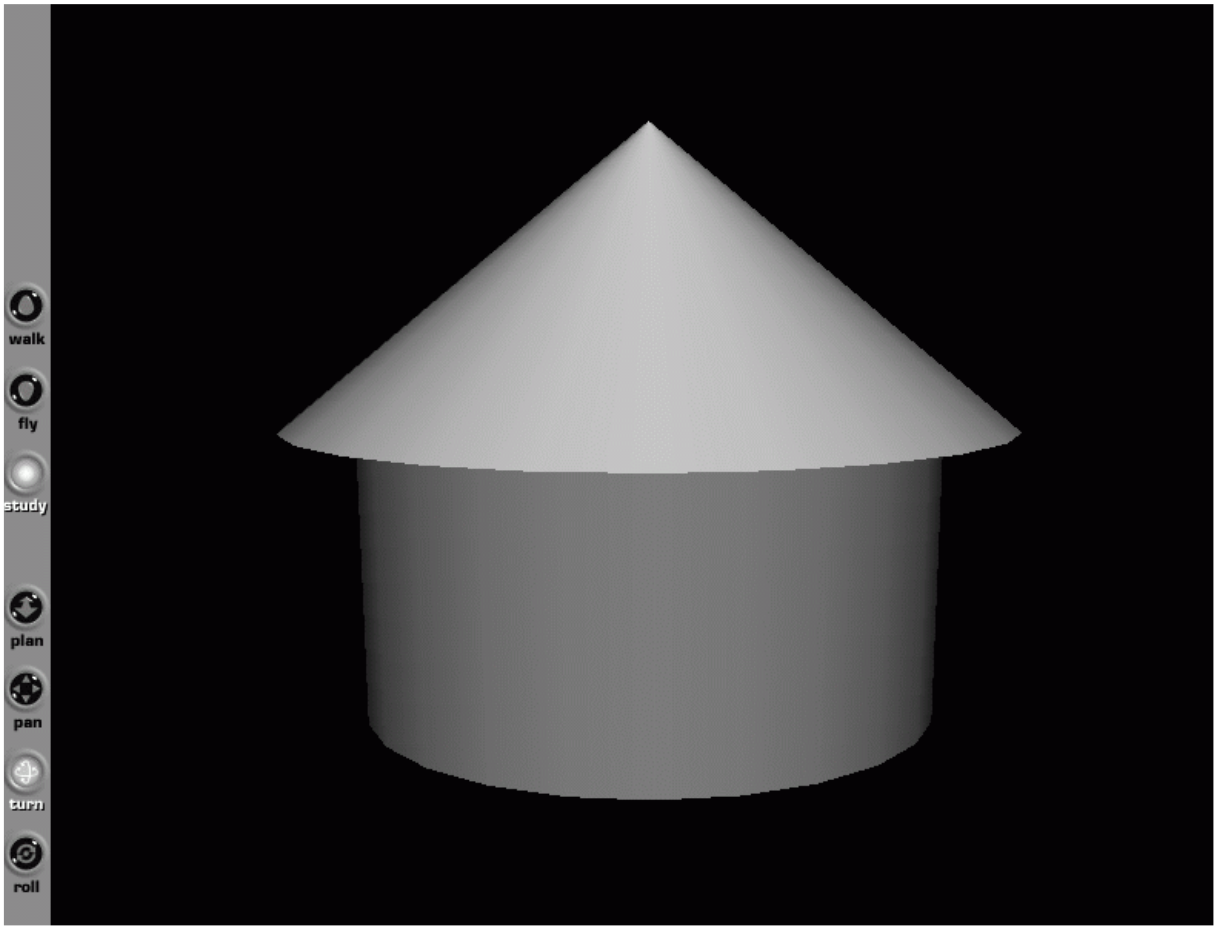
default für Material: weiß, "selbstleuchtend" (nicht reflektierend)

Mit selbst gesetzten Materialeigenschaften (hier: diffus reflektierende Oberfläche, grau bzw. weiß):

```
#VRML V2.0 utf8
Shape
   {
   geometry Cylinder
      {
      height 2.0
      radius 2.0
      }
   appearance Appearance
      {
      material Material
         { diffuseColor 0.5 0.5 0.5 }
      }
   }
Transform
   {
   translation 0.0 2.0 0.0
   children
      [
      Shape
         {
         geometry Cone
            {
            bottomRadius 2.5
            }
         appearance Appearance
             {
            material Material
                { diffuseColor 1.0 1.0 1.0 }
            }
         }
      ]
   }
```

Der Shape-Knoten gruppiert Geometrie und Material-eigenschaften!

Ergebnis:



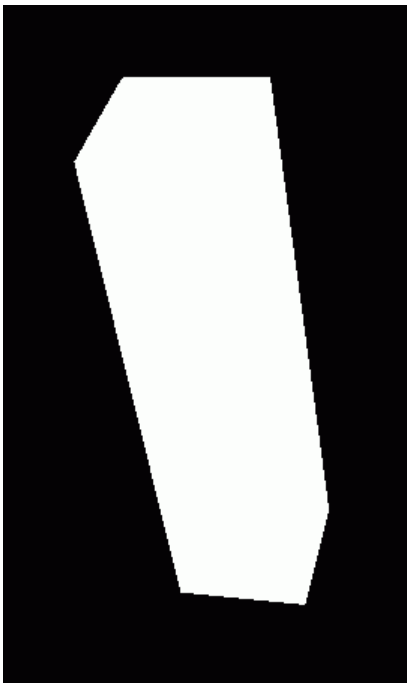
Einfachstes Objekt:

- Box (Quader)

nur die 3 Abmessungen (Länge, Höhe, Tiefe) werden hier angegeben:

```
#VRML V2.0 utf8
Box
    {
    size 1 4 1
    }
```

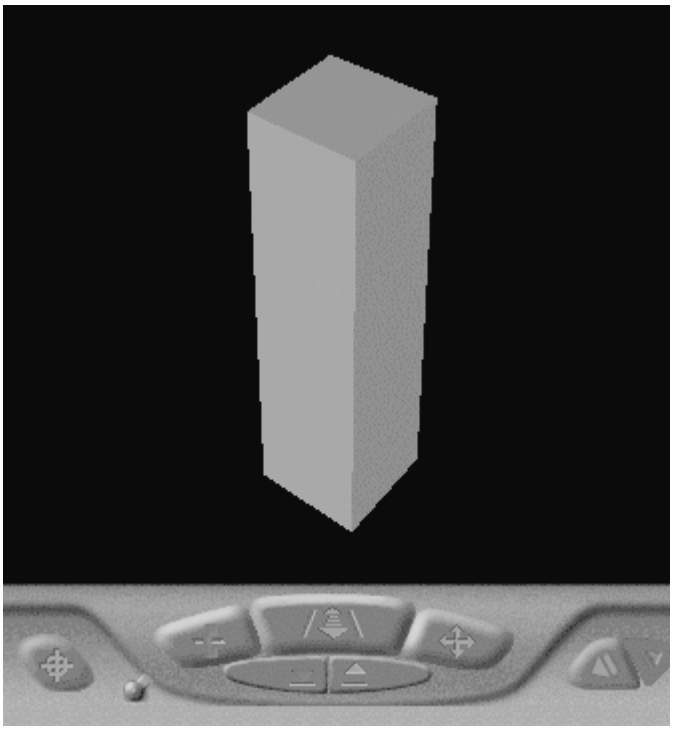
Ergebnis:



dasselbe mit Materialeigenschaften:

```
#VRML V2.0 utf8
Shape
    {
    geometry Box
        {
        size 1 4 1
        }
    appearance Appearance
        {
        material Material
            { diffuseColor 0.5 0.5 0.5 }
        }
    }
```

Ergebnis (hier im CosmoPlayer):



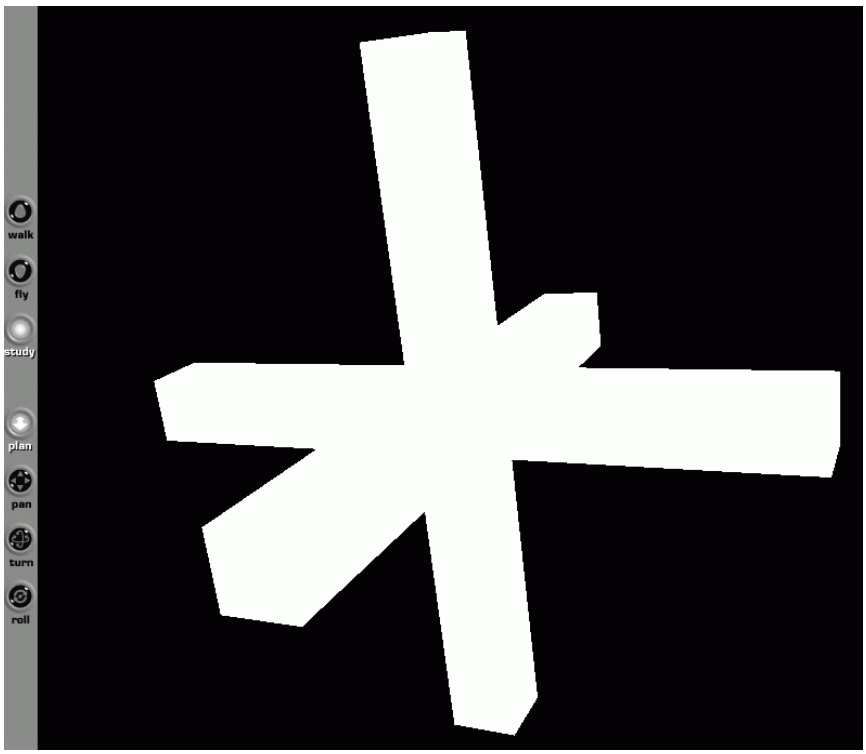
Felder, für die keine Spezifikation angegeben wird, werden automatisch mit Default-Werten besetzt.

Die *Primitiv-Objekte* (Box, Cone, Cylinder, Sphere) werden standardmäßig um den Nullpunkt zentriert.

Überlagerung von 3 verschieden dimensionierten Quadern zu einem "Koordinatenkreuz":

```
#VRML V2.0 utf8
Box
   { size 10 1 1 }
Box
   { size 1 10 1 }
Box
   { size 1 1 10 }
```

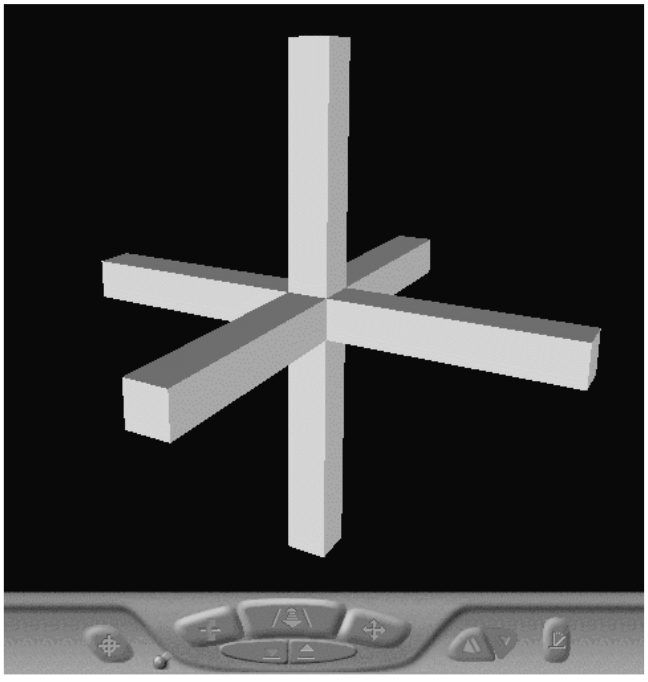
Ergebnis:



mit Materialfarbe:
es muss wieder mit **Shape**-Knoten gruppiert werden

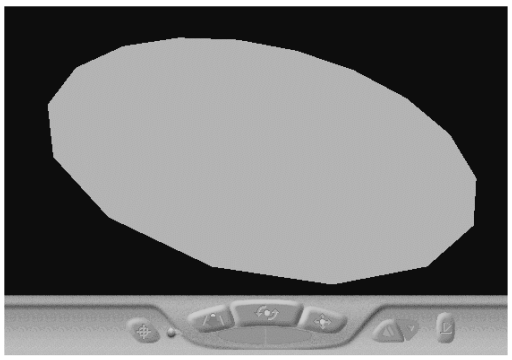
```
#VRML V2.0 utf8
Shape
   {
   geometry Box
      { size 12 1 1 }
   appearance Appearance
      {
      material Material { diffuseColor 1.0 1.0 1.0 }
      }
   }
Shape
   {
   geometry Box
      { size 1 12 1 }
   appearance Appearance
      {
      material Material { diffuseColor 1.0 1.0 1.0 }
      }
   }
Shape
   {
   geometry Box
      { size 1 1 12 }
   appearance Appearance
      {
      material Material { diffuseColor 1.0 1.0 1.0 }
      }
   }
```

Ergebnis:



Ansprechen von Teilen eines Primitivobjekts über spezielle Felder, z.B. beim Zylinder: **`side, top, bottom`**

```
#VRML V2.0 utf8
Cylinder
    {
    bottomRadius 3
    height 0.7
    side FALSE
    }
```



Beachte: es ist jeweils nur die Außenseite der Polygone sichtbar! (Hier also je nach Blickwinkel "top" *oder* "bottom", obwohl beide noch "vorhanden" sind)

```
#VRML V2.0 utf8
Cylinder
    {
    radius 3
    height 0.7
    top FALSE
    bottom FALSE
    }
```

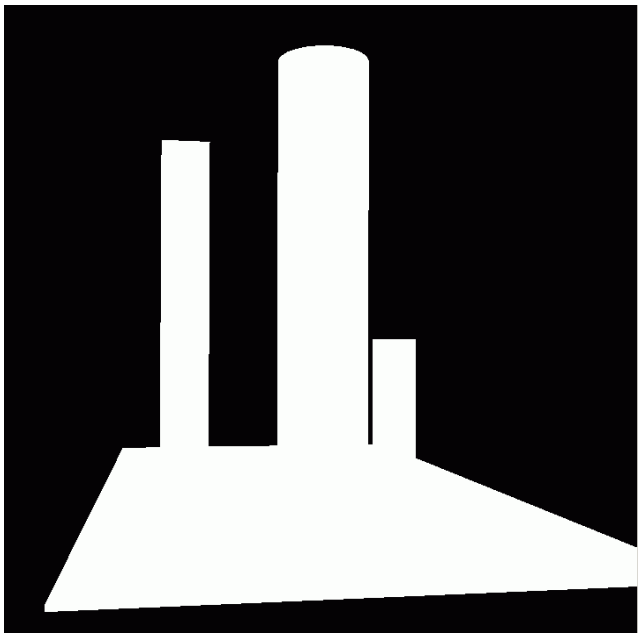
Ergebnis:

Verwendung von Transformationsknoten zur Positionierung von Objekten

Beispiel:

```
#VRML V2.0 utf8
Box { size 10 0.1 20 }    # Boden
Transform
   {
   translation -3 4 4
   children
      [
      Box { size 1 8 1 }
      ]
   }
Transform
   {
   translation 3 2 -1
   children
      [
      Box { size 1 4 1 }
      ]
   }
Transform
   {
   translation 0 5 4
   children
      [
      Cylinder { height 10 }
      ]
   }
```

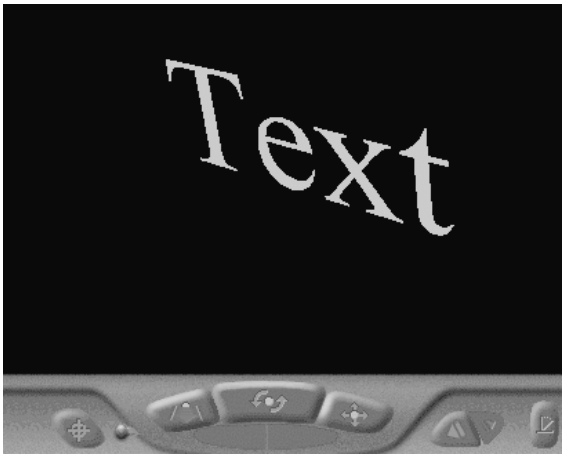
Ergebnis:



Textausgabe in VRML

```
#VRML V2.0 utf8
Text
    {
    string "Text"
    }
```

Ergebnis:



VRML-Knoten

Jeder VRML-Knoten kann 0 oder mehr Felder enthalten. Die Felder haben eine festgelegte Semantik, Typisierung und Default-Werte.

Typen in VRML 97

Standard-Feldtypen: Bezeichnung beginnt mit SF oder MF
SF = "single field", einzelner Wert
MF = "multiple field", Array

SFNode / MFNode	VRML-Knoten
SFBool	TRUE oder FALSE
SFColor / MFColor	3 Gleitkommazahlen zwischen 0.0 und 1.0 (RGB-Farbmodell)
SFFloat / MFFloat	Gleitkommazahl(en)
SFImage	Pixel-Beschreibung einer Bitmap
SFInt32 / MFInt32	32Bit-Ganzzahlen
SFRotation / MFRotation	4 Gleitkommazahlen: 3 für die Drehachse, letzte für den Drehwinkel in Bogenmaß
SFString / MFString	Zeichenkette (utf8-Zeichensatz)
SFTime / MFTime	Anzahl Sekunden seit 1. 1. 1970, 0 Uhr, als doppelt genaue Gleitkommazahl
SFVec2f / MFVec2f	2 Gleitkommazahlen als 2D-Vektor (bzw. Array solcher Vektoren)
SFVec3f / MFVec3f	3 Gleitkommazahlen als 3D-Vektor (bzw. Array solcher Vektoren)

MF-Werte werden in eckige Klammern [ ] eingeschlossen und innerhalb der eckigen Klammern durch Kommata oder Leerzeichen voneinander getrennt. Bei genau einem Wert können die eckigen Klammern auch weggelassen werden.

Beispiel:
Koordinatenknoten haben ein Feld vom Typ MFVec3f. Der Inhalt ist eine Liste von 3D-Ortsvektoren.

```
Coordinate
    {
    point [ x1 y1 z1, ..., xn yn zn ]
    }
```

Elementarknoten in VRML 97:

Knoten	*Felder*	*Feldtyp*	*Bedeutung*
Shape	**appearance** **geometry**	SFNode SFNode	Materialdaten Geometrie-Knoten (Primitiv-Obj., Mengen...)
Coordinate	**point**	MFVec3f	Liste v. 3D-Koord.
Normal	**point**	MFVec3f	Liste v. 3D-Koord. (zur Festlegung v. Normalenvektoren)

Zentral sind *Gruppen-* und *Transformations-Knoten* (für die Konstruktion des Szenengraphen). Sie gruppieren bzw. transformieren beliebige Unterbäume im Szenengraphen:

Group	**children**	MFNode	Array der Kind-Knoten
	addChildren	MFNode	für eventIn
	removeChildren	MFNode	für eventIn
Transform	**center**	SFVec3f	Zentr. für Rot. u. Skalierung
	scale	SFVec3f	Sk.-faktoren
	scaleOrientation	SFRotation	Rot. für scale
	rotation	SFRotation	Rotation
	translation	SFVec3f	Translation
	children	MFNode	Kind-Knoten
	addChildren	MFNode	für eventIn
	removeChildren	MFNode	für eventIn
	bboxCenter	SFVec3f	Zentr. d. bbox
	bboxSize	SFVec3f	bbox-Größe

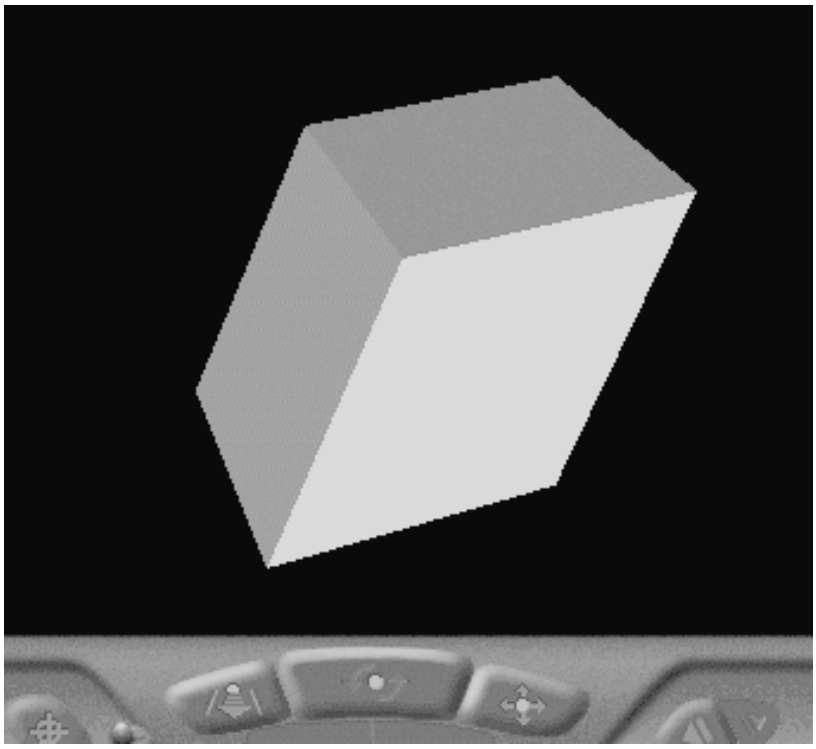
In VRML 1.0 gibt es auch die Möglichkeit, Transformationen direkt über eine Matrix zu spezifizieren:

MatrixTransform	**matrix**	MFFloat (16 Einträge)	Transf.-matrix (transponiert)

Beispiel:
Anwendung einer Scherung auf einen Würfel

```
#VRML V1.0 ascii
MatrixTransform
    {
    matrix 1   0 0 0
           0.5 1 0 0
           0   0 1 0
           0   0 0 1
    }
Cube { }
```

Ergebnis:



Grafische Primitive in VRML 97:

Box	**size**	SFVec3f	Kantenlängen eines geschlossenen Quaders
Cone	**bottomRadius**	SFFloat	Radius des Bodens
	height	SFFloat	Höhe des Kegels
	side	SFBool	Sichtbarkeit: Mantel
	bottom	SFBool	Sichtbarkeit: Boden
Cylinder	**bottomRadius**	SFFloat	Radius
	height	SFFloat	Höhe des Zylinders
	side	SFBool	Sichtb.: Mantel
	top	SFBool	Sichtb.: Deckel
	bottom	SFBool	Sichtb.: Boden
Sphere	**radius**	SFFloat	Radius der Kugel
Text	**string**	MFString	1 oder mehrere Zeichenketten
	fontStyle	SFNode	Font-Spezifikation
	length	MFFloat	max. Länge
	maxExtent	SFFloat	max. phys. Länge

Knoten für Punktmengen und boundary repr. in VRML 97:

Punktmenge (ohne Flächen und Kanten):

PointSet	**coord**	SFNode	Koordinatenknoten
	color	SFNode	Farbknoten

darin der Farbknoten:

Color	**color**	MFColor	RGB-Spezifikation(en)

Indizierte Kantenmenge:

IndexedLineSet	**coord**	SFNode	Koord.knoten
	coordIndex	MFInt32	Polylinien
	set_coordIndex	MFInt32	für eventIn
	color	SFNode	Farbzuordn.
	colorIndex	MFInt32	Farbzuordn.
	set_colorIndex	MFInt32	für eventIn
	colorPerVertex	SFBool	Farbe bez. auf Ecken (sonst aufKanten)

Indizierte Flächenmenge:

IndexedFaceSet	**coord**	SFNode	Koord.knoten
	coordIndex	MFInt32	Polygone
	set_coordIndex	MFInt32	für eventIn
	color	SFNode	Farbzuordn.
	colorIndex	MFInt32	Farbzuordn.
	set_colorIndex	MFInt32	für eventIn
	colorPerVertex	SFBool	Farbe bez. auf Ecken (sonst auf Flächen)

Die Indextabelle (**coordIndex**) enthält die Indices (Zählung beginnt bei 0) der zu einer Polylinie bzw. zu einem geschlossenen Polygon gehörenden Ecken. Trennung mehrerer Polylinien / Polygone durch den Eintrag "–1":
coordIndex [1,5,4,2, –1, 0,3,7,6, –1, 1,5,7,6]
= 3 Vierecke

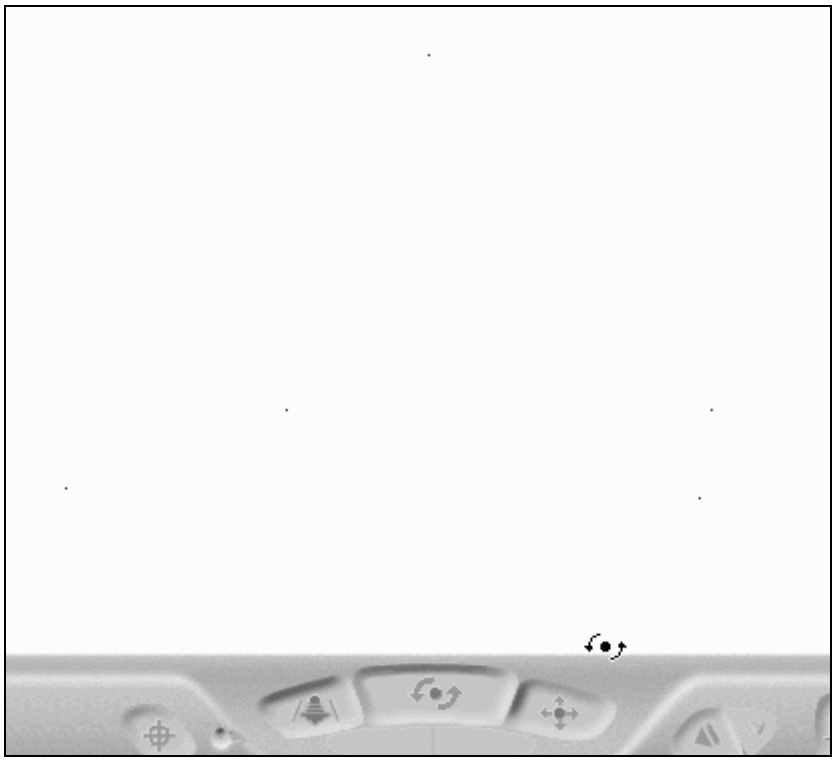
Orientierung ist wichtig – für die Normalenvektoren gilt die "Rechte-Hand-Regel"; die Rückseite eines Polygons (entgegengesetzt zum Normalenvektor) ist unsichtbar.

Beispiele:

Konstruktion einer Pyramide als Punktmenge

```
#VRML V2.0 utf8
# pyrpkt.wrl
Shape
    {
    geometry PointSet
        {
        coord Coordinate
            {
            point [ -2, 0, -2
                     2, 0, -2
                    -2, 0, 2
                     2, 0, 2
                     0, 3, 0  # Spitze
                  ]
            }
        }
    }
```

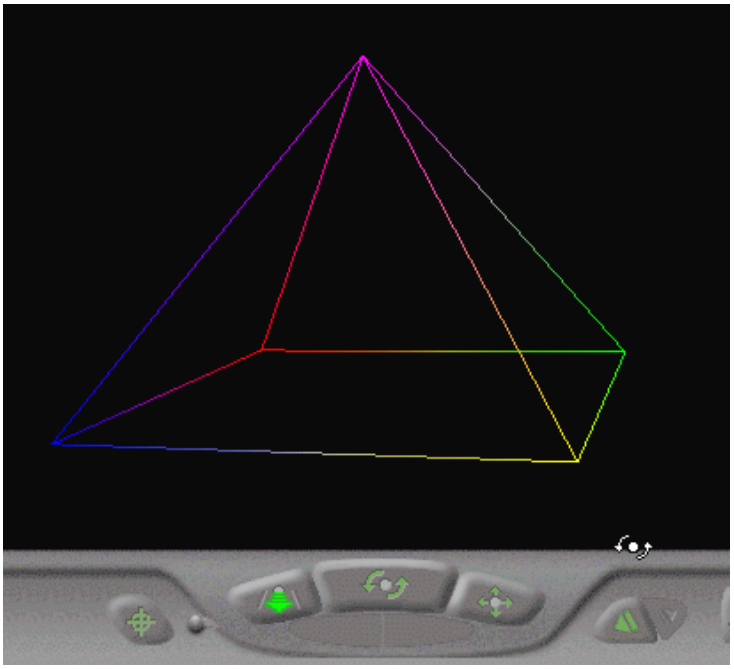
Ergebnis (Bild invertiert, da Eckpunkte sonst kaum sichtbar):



Pyramide als durch Eckenindices definierte Kantenmenge:

```
#VRML V2.0 utf8
# pyrlin3.wrl
Shape
   {
   geometry IndexedLineSet
      {
      coord Coordinate
         {
         point [ -2 0 -2 # Ecke 0
                  2 0 -2 # Ecke 1
                 -2 0 2  # Ecke 2
                  2 0 2  # Ecke 3
                  0 3 0  # Ecke 4 = Spitze
               ]
         }
      coordIndex [ 0 1 -1,
                   1 3 -1,
                   3 2 -1,
                   2 0 -1,
                   0 4 -1,
                   1 4 -1,
                   3 4 -1,
                   2 4
                 ]
      color Color
         {
         color [ 1 0 0,
                 0 1 0,
                 0 0 1,
                 1 1 0,
                 1 0 1 ]
         }
      }
   }
```

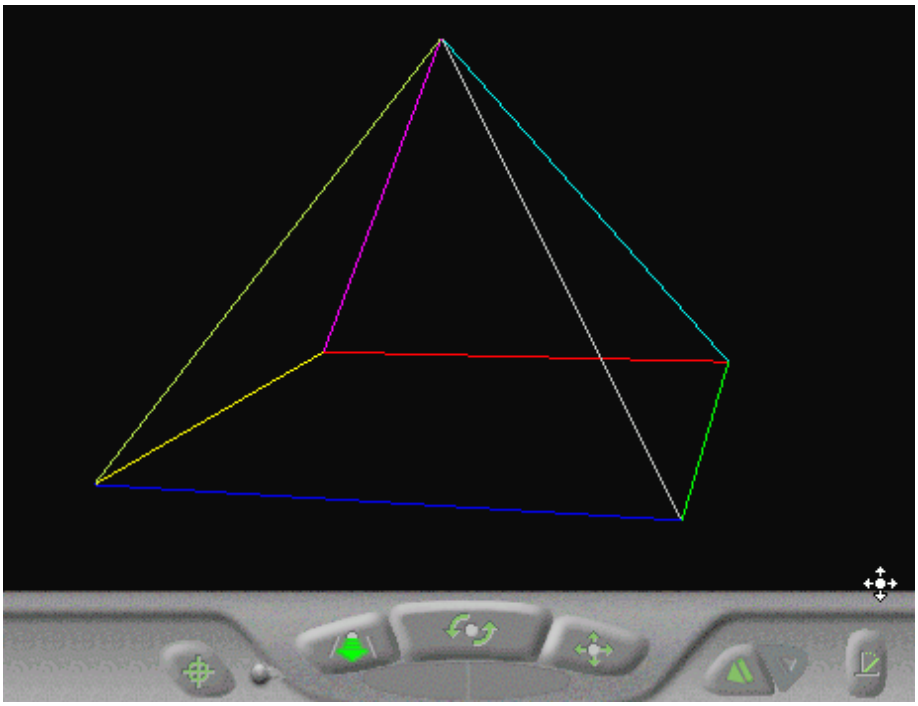
Ergebnis:



Die Farben werden hier zwischen den Ecken interpoliert.
Direkte Farbzuordnung zu den Kanten durch Einfügen von

`colorPerVertex FALSE`

in den **`IndexedLineSet`**-Knoten:



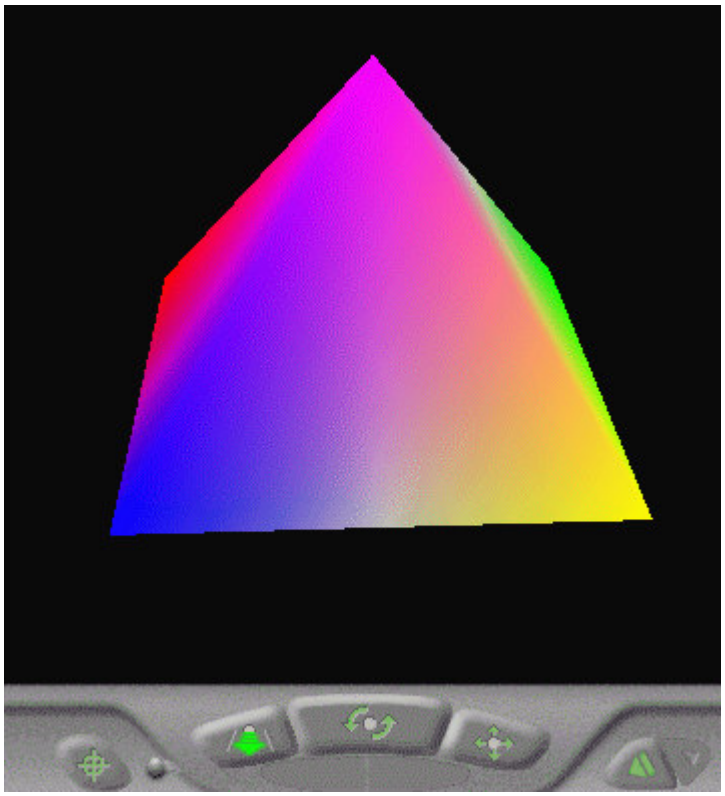
auch möglich: Polylinien (mehr als 2 Punkte) anstatt Kanten

```
coordIndex [ 4 1 0 -1,
             4 3 1 -1,
             4 2 3 -1,
             4 0 2 -1,
             2 0 1 3   # Boden
           ]
```

Pyramide als Körper mit durch Punktindices definierten Facetten:

```
#VRML V2.0 utf8
# pyrface1.wrl
Shape
   {
   geometry IndexedFaceSet
      {
      coord Coordinate
         {
         point [ -2 0 -2 # Ecke 0
                  2 0 -2 # Ecke 1
                 -2 0 2  # Ecke 2
                  2 0 2  # Ecke 3
                  0 3 0  # Ecke 4 = Spitze
               ]
         }
      coordIndex [ 4 1 0 -1,
                   4 3 1 -1,
                   4 2 3 -1,
                   4 0 2 -1,
                   2 0 1 3   # Boden
                 ]
      color Color
         {
         color [ 1 0 0,
                 0 1 0,
                 0 0 1,
                 1 1 0,
                 1 0 1 ]
         }
      }
   }
```

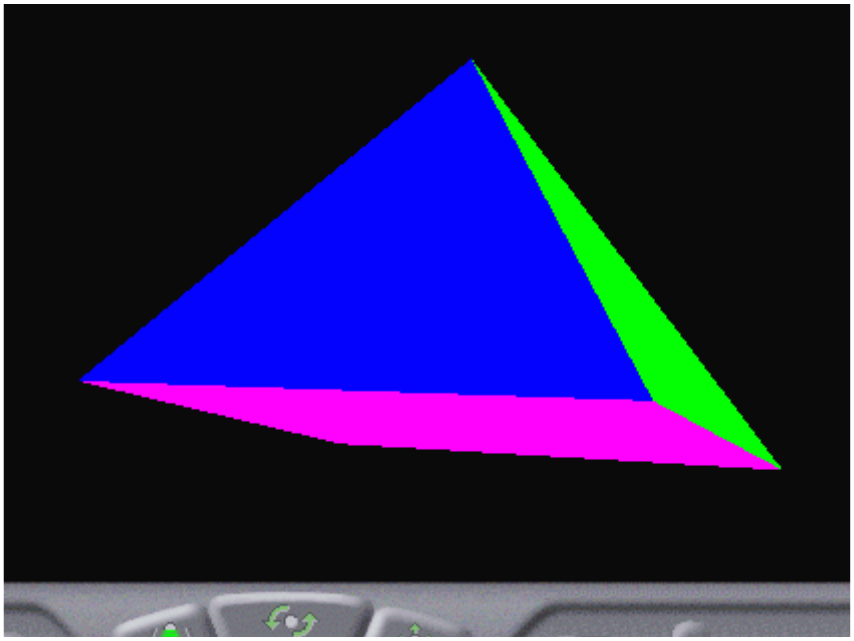
Ergebnis:



auch hier Farbinterpolation zwischen den Eckpunkten – direkte Farbzuordnung zu den Facetten wird durch den Switch

```
colorPerVertex FALSE
```

eingestellt:



Extrusion: Spur eines in der *xz*-Ebene def. Polygons entlang einer Raumkurve

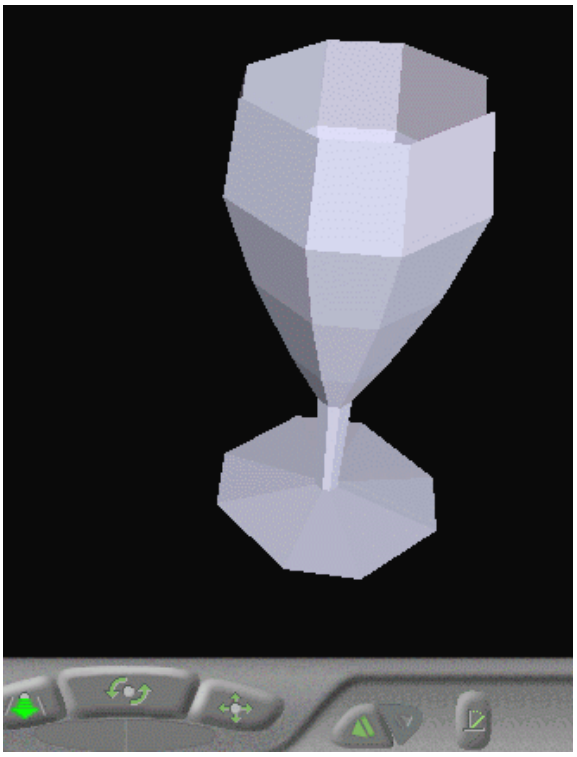
Extrusion	**crossSection**	MFVec2f	Polygon
	spine	MFVec3f	Stützpunkte der Kurve
	endCap	SFBool	Deckel exist.
	solid	SFBool	Fläche beidseitig sichtbar
	scale	MFVec2f	Skalierungs-Array
	orientation	MFRotation	Drehungen entl. Kurve

Beispiel:

```
#VRML V2.0 utf8
# wein1.wrl

Shape
   {
   geometry Extrusion
      {
      endCap FALSE
      solid  FALSE
      crossSection [ -1 -3, -3 -1, -3 1, -1 3,
                     1 3, 3 1, 3 -1, 1 -3, -1 -3 ]
      spine [ 0 0 0, 0 0.6 0, 0 7 0, 0 10 0,
              0 15 0, 0 20 0, 0 25 0 ]
      scale [ 2 2, 0.2 0.2, 0.3 0.3, 0.8 0.8,
              1.5 1.5, 2 2, 1.8 1.8 ]
      }
   appearance Appearance
      {
      material Material
         { diffuseColor 0.9 0.9 1 }
      }
   }
```

achteckige Grundfläche, Extrusionskurve (*spine*) mit 7 Stützpunkten

Ergebnis:



Modifikation: Drehung während der Extrusion
– füge in den Extrusion-Knoten für jeden Stützpunkt eine Orientierungsspezifikation ein:

```
orientation [ 0 1 0 0, 0 1 0 0, 0 1 0 0.3,
              0 1 0 0.6, 0 1 0 0.9, 0 1 0 1.2,
              0 1 0 1.5 ]
```

Ergebnis:

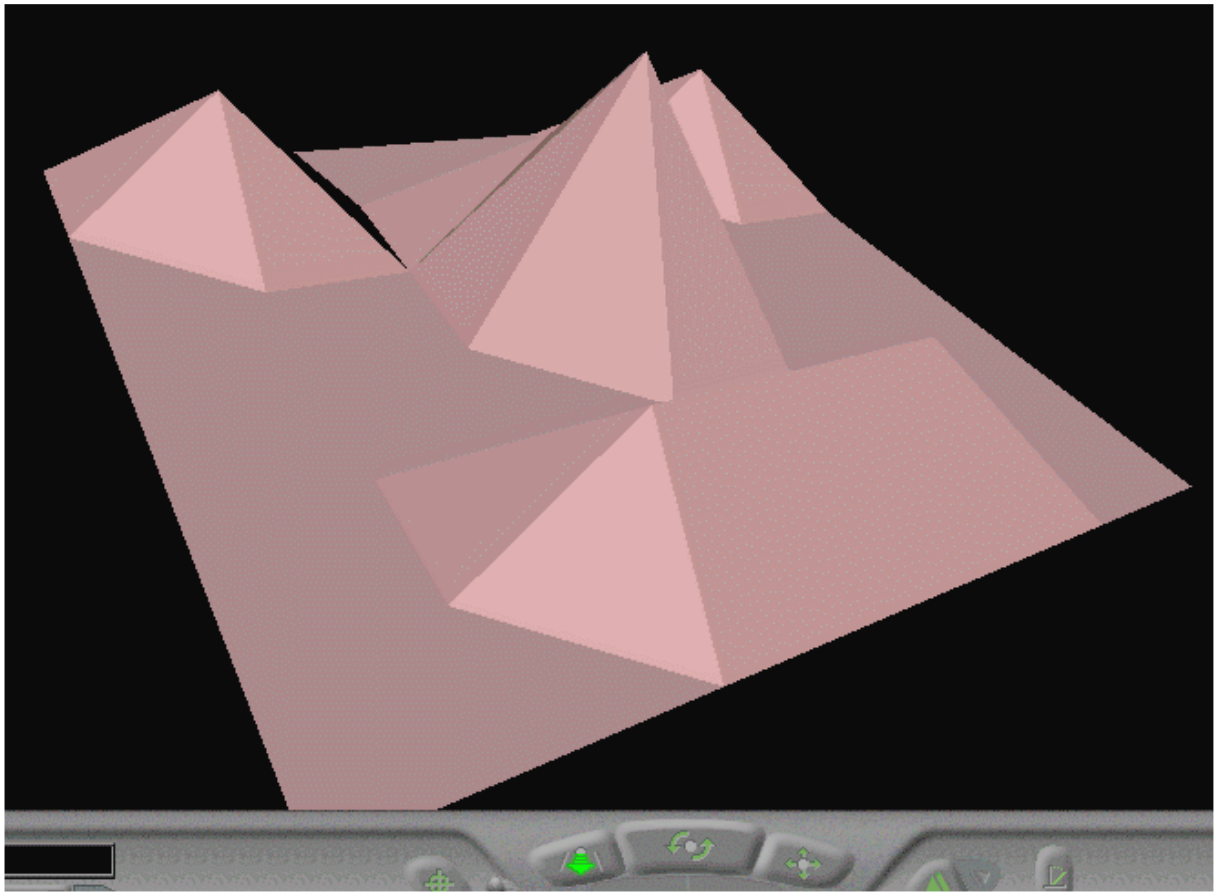
Höhengitter: Erhebungsgitter über der *xz*-Ebene

ElevationGrid	**xDimension**	SFInt32	Anz. Pkte in *x*-Richtung
	xSpacing	SFFloat	Abstand
	zDimension	SFInt32	Anz. Pkte in *y*-Richtung
	zSpacing	SFFloat	Abstand
	height	MFFloat	Höhenwerte-Array
	colorPerVertex	SFBool	Farb-Flag
	creaseAngle	SFFloat	Kanten-schärfe
	ccw	SFBool	counter-clockwise

Beispiel:

```
#VRML V2.0 utf8
# landsc1.wrl

Shape
   {
   geometry ElevationGrid
      {
      xDimension 7
      zDimension 7
      xSpacing 1
      zSpacing 1
      height [ 0 0 0 0 0 0 0
               0 1 0 0 0 1 0
               0 0 0 1 0 0 0
               0 0 0 2 0 0 0
               0 0 0 0 0 0 0
               0 0 1 1 1 0 0
               0 0 0 0 0 0 0 ]
      }
   appearance Appearance
      {
      material Material
         { diffuseColor 0.9 0.7 0.7 }
      }
   }
```

Ergebnis:



Hintergrund-Knoten:

Background	**groundColor**	MFColor	Farbwerte
	groundAngle	MFFloat	Winkel für Bodenfarben
	skyColor	MFColor	Farbwerte
	skyAngle	MFFloat	Winkel für Himmelsfarben
	backUrl	SFString	URL für hinteres,
	bottomUrl	SFString	unteres usw.
	frontUrl	SFString	Hintergrundbild
	leftUrl	SFString	
	rightUrl	SFString	
	topUrl	SFString	
	set_bind	SFBool	für eventIn
	isBound	SFBool	für eventOut

Anwendung auf das letzte Beispiel

wir hängen ans Ende an:

```
Background
    {
    skyAngle [ 1.6 ]
    skyColor [ 0 0 1, 0.4 0.4 1 ]
    groundAngle [ 1.6 ]
    groundColor [ 0.8 0.8 0.8, 0.01 0.025 0.001 ]
    frontUrl "wiessee.jpg"
    backUrl "wiessee.jpg"
    leftUrl "wiessee.jpg"
    rightUrl "wiessee.jpg"
    }
```

Ergebnis:

Hyperlink zu anderen VRML-Dateien:

Inline	**description**	SFString	Targetbeschreibung
	parameter	MFString	Parameter
	url	MFString	Liste der Targets

Hyperlink zu anderen Dateien (nicht notw. VRML):

Anchor	**description**	SFString	Targetbeschreibung
	parameter	MFString	Parameter
	url	MFString	Liste der Targets
	children	MFNode	Feld für Kindknoten
	addChildren	MFNode	für eventIn
	removeChildren	MFNode	für eventIn
	bboxCenter	SFVec3f	Zentr. der bounding box
	bboxSize	SFVec3f	Größe der b. box

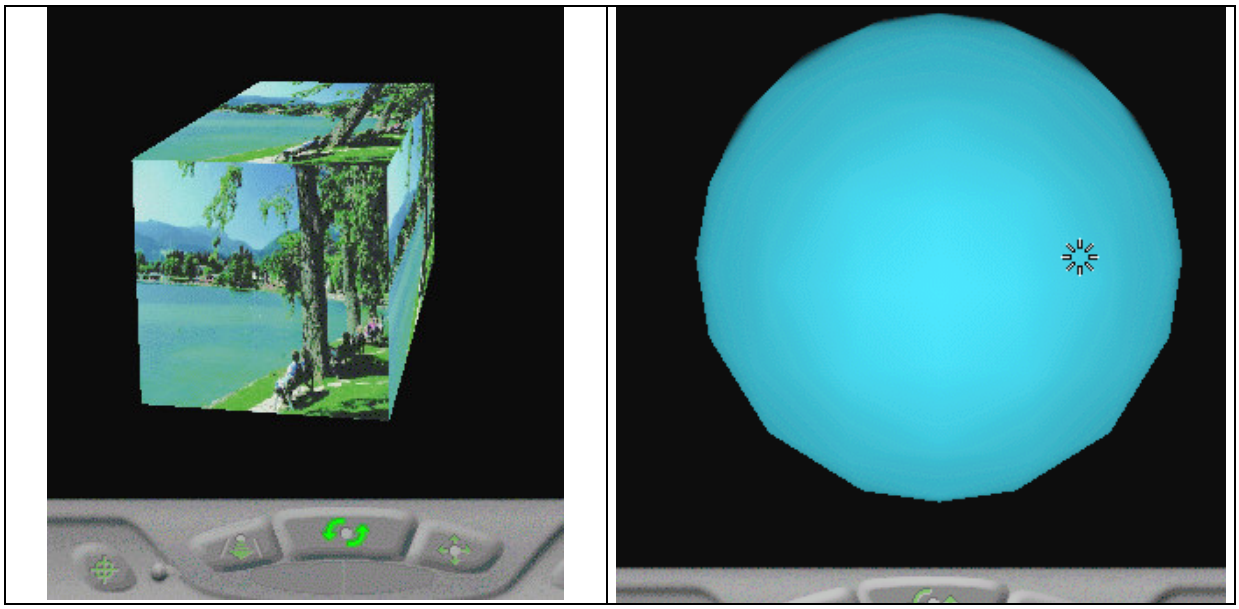
Beispiel:
2 Dateien, die wechselseitig verlinkt sind.
Erste Datei:

```
#VRML V2.0 utf8
# anchor1.wrl

Anchor
   {
   url "anchor2.wrl"
   description "Eintritt in eine neue Welt"
   children
      [
      Shape
         {
         geometry Box { size 1 1 2 }
         appearance Appearance
            {
            texture ImageTexture
               { url "wiessee.jpg" }
            }
         }
      ]
   }
```

Zweite Datei:

```
#VRML V2.0 utf8
# anchor2.wrl

Anchor
    {
    url "anchor1.wrl"
    description "Zurück in die alte Welt"
    children
        [
        Shape
            {
            geometry Sphere { radius 3 }
            appearance Appearance
                {
                material Material
                    { diffuseColor 0.3 0.9 1 }
                }
            }
        ]
    }
```

Ergebnis:



Die "Teleportation" von einer Welt in die andere erfolgt durch Anklicken des Anker-Objekts mit der Maus. (Der Mauscursor verändert seine Gestalt über dem Anker-Objekt, siehe rechts.)

Optische Eigenschaften von Objekten

Festlegung im **`appearance`**-Feld des **`Shape`**-Knotens.
Zugehöriger **`Appearance`**-Knoten:

`Appearance`	**`material`**	SFNode
	`texture`	SFNode
	`textureTransform`	SFNode

Ins **`material`**-Feld wird ein **`Material`**-Knoten eingebunden:

`Material`	**`ambientIntensity`**	SFFloat	Bruchteil der Reflexion v. ungerichtetem Umgebungslicht
	`diffuseColor`	SFColor	Farbe unter Bestrahlung mit weißem Licht
	`emissiveColor`	SFColor	Farbe unabh. von der Beleuchtung
	`specularColor`	SFColor	Farbe der spiegelnden Reflexion
	`shininess`	SFFloat	Spiegelungsexponent
	`transparency`	SFFloat	Durchsichtigkeit

Ambiente Reflexion: diffuses Licht ohne bestimmte Richtung (indirekte oder Hintergrundbeleuchtung), abhängig nur von Anzahl der Lichtquellen, nicht von ihren Positionen. Der ambiente Reflexionskoeffizient gibt an, welcher Anteil der ambienten Beleuchtung reflektiert wird (Farbe wie bei diffuser Reflexion).

Diffuse Reflexion: Licht wird von einer Fläche mit gleicher Intensität in alle Richtungen reflektiert, Helligkeit hängt ab von

- Winkel zwischen Flächennormale u. Richtung zur Lichtquelle
- Reflexionskoeffizienten für jede der 3 Grundfarben.

Spiegelnde Reflexion (*specular reflection*): Ausfallswinkel = Einfallswinkel (relativ zur Normalen).
Stärke des Intensitätsabfalls, wenn Objekt nicht exakt unter dem Reflexionswinkel betrachtet wird: *shininess* (Glanz)

- niedrige shininess: verschmierte, weiche Reflexe
- hohe shininess: kleine, harte Reflexe (glatte Oberfläche)

Transparenz: transparency-Koeffizient gibt an, welcher Anteil des auftreffenden Lichts die Fläche durchdringt
0 = undurchsichtig (default-Wert)
1 = total durchsichtig

Eigenleuchten (*emissive color*): inneres Glühen, wirkt aber nicht auf andere Objekte als Lichtquelle ein.

Licht- und Klangquellen:
Festlegung durch eigene Knoten (auf gleicher Ebene wie **`Shape`**)

`PointLight` (kugel-symmetrisch abstrahlende Punktlicht-quelle, u.U. rechenintensiv)	**`location`** **`on`** **`intensity`** **`ambientIntensity`** **`attenuation`** **`radius`** **`color`**	SFVec3f SFBool SFFloat SFFloat MFFloat SFFloat SFColor	Ort d. Lichtqu. ein/aus Stärke (0...1) Beitrag zum ambienten Licht (0...1) 3 Koeff.: konst., lin., quadr. Ab-schwächung Reichweite Farbe
`Direc-tionalLight` (unendlich ent-fernte Punkt-lichtquelle, pa-rallele Strah-len, geringste Rechenzeit)	**`direction`** **`on`** **`intensity`** **`ambientIntensity`** **`color`**	SFVec3f SFBool SFFloat SFFloat SFColor	Richtung der Lichtstrahlung ein/aus Stärke (0...1) Beitr. zum ambienten Licht (0...1) Farbe

SpotLight (Lichtkegel mit fester Begrenzung (**cutOffAngle**), größte Rechenzeit)	**location**	SFVec3f	Ort d. Lichtqu.
	direction	SFVec3f	Richtung d. Kegelachse
	on	SFBool	ein/aus
	intensity	SFFloat	Stärke (0...1)
	ambientIntensity	SFFloat	Beitr. zum ambienten Licht (0...1)
	cutOffAngle	SFFloat	halber Öffn.-winkel (Bogenmaß)
	beamWidth	SFFloat	Strahldicke
	radius	SFFloat	Reichweite
	color	SFColor	Farbe
Sound	**direction**	SFVec3f	Richtung der Abstrahlung
	intensity	SFFloat	Lautstärke
	location	SFVec3f	Ort der Quelle
	maxBack	SFFloat	hinterer Rand der Hörbar-keits-Ellipse
	maxFront	SFFloat	vord. Rand
	minBack	SFFloat	min. Reich-weite
	minFront	SFFloat	
	priority	SFFloat	Priorität
	source	SFNode	**AudioClip**-Knoten
	spatialize	SFBool	Lokalisierbarkeit
AudioClip	**description**	SFString	Name des Audioclips
	loop	SFBool	Endlos-schleife
	pitch	SFFloat	Abspiel-geschwindigk. (Tonhöhe)
	startTime	SFTime	
	stopTime	SFTime	
	url	MFString	Adresse(n)

	duration_changed **isActive**	SFTime SFBool	der Audio-datei(en)) für Kommu-) nikation

Gerichtetes Licht (**DirectionalLight**) wirkt nur auf Knoten des eigenen Teilbaumes, dafür aber unabhängig von der Entfernung.

Beispiel:
3 blaue Kugeln sollen durch weißes Licht angestrahlt werden

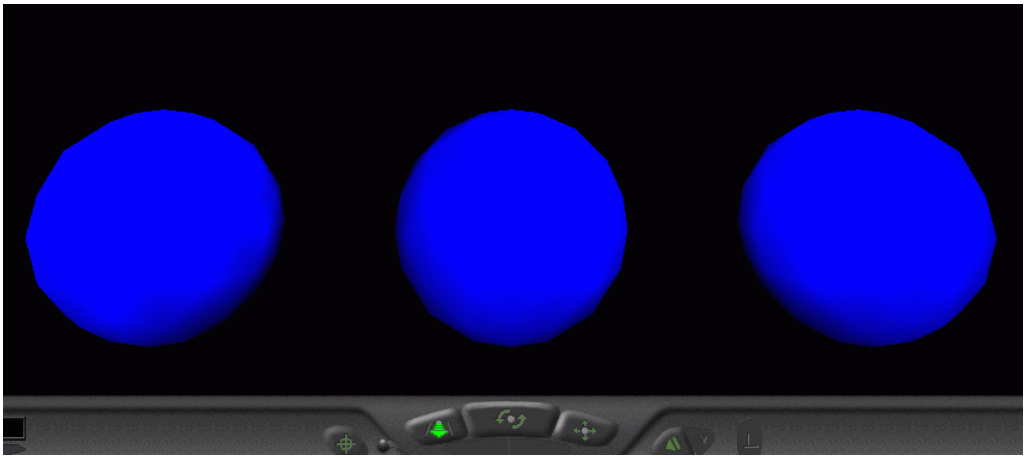
- gerichtetes Licht
- Punktlicht
- Spotlicht

Wirkung auf nur je eine Kugel (durch Teilbaum-Zugehörigkeit bei der 1., bzw. durch Radius-Angabe bei der 2. und 3.)

```
#VRML V2.0 utf8
Transform
   {
   translation -3 0 0
   children
      [
      Shape
         {
         geometry Sphere {}
         appearance Appearance
            {
            material Material
               { diffuseColor 0 0 1 }
            }
         }
      DirectionalLight
         { direction 0 -1 -1 }
      ]
   }

PointLight
```

```
    {
    location 0 3 3
    radius 3
    }
Shape
    {
    geometry Sphere {}
    appearance Appearance
       {
       material Material
          { diffuseColor 0 0 1 }
       }
    }

SpotLight
    {
    direction 0 -1 -1
    location 3 3 3
    radius 3
    }
Transform
    {
    translation 3 0 0
    children
       Shape
          {
          geometry Sphere {}
          appearance Appearance
             {
             material Material
                { diffuseColor 0 0 1 }
             }
          }
    }
```

Ergebnis:



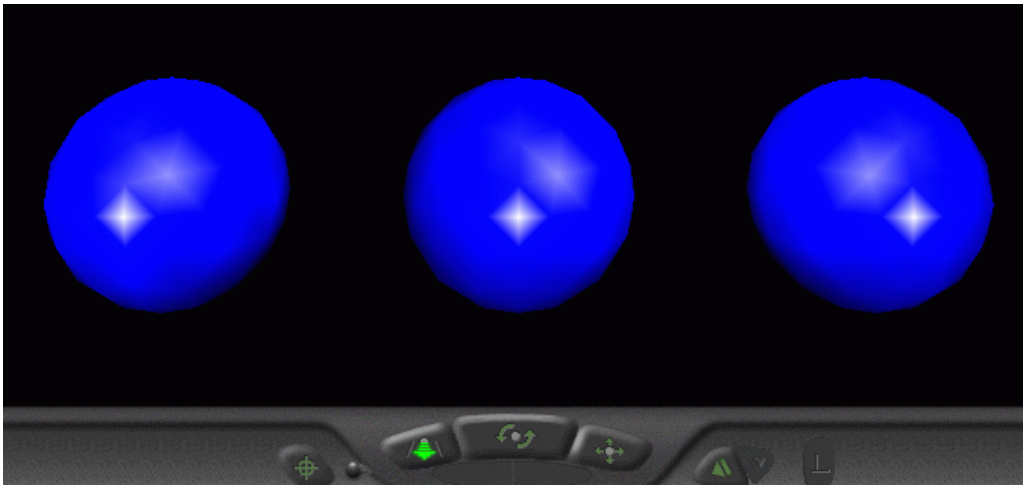
Modifikation, indem jeweils nach dem Feld

```
diffuseColor 0 0 1
```

unter den Materialeigenschaften noch eingefügt wird:

```
specularColor 1 1 1
shininess 1
```

Ergebnis:



Texturen

- werden als Texturknoten unter dem Appearance-Knoten eingebunden
- 3 Typen von Texturen: Bitmap, Grafikdatei (URL; Formate JPG, GIF oder PNG), digitales Video (URL; Format MPEG1).

PixelTexture	**image**	SFImage	Bitmap
	repeatS	SFBool	horiz. Muster-Wiederholung
	repeatT	SFBool	vertikale Wdh.
ImageTexture	**url**	MFString	Adresse(n) zum Auffinden der Texturdatei
	repeatS	SFBool	horiz. Wdh.
	repeatT	SFBool	vertik. Wdh.
MovieTexture	**loop**	SFBool	Endlosschleife
	speed	SFFloat	Abspiel-geschwindigkeit
	startTime	SFTime	
	stopTime	SFTime	
	url	MFString	Adresse(n) zum Auffinden der Movie-Datei
	repeatS	SFBool	horiz. Wdh.
	repeatT	SFBool	vertik. Wdh.
	duration_ changed	SFTime	) dienen der) Kommunikation
	isActive	SFBool	)

Texturen werden mit einem 2D-Koordinatensystem (*s*, *t*) versehen:

Blickpunkte (Kameras)

In einer Szene können beliebig viele Blickpunkte in Form von **`Viewpoint`**-Knoten plaziert werden ("Aussichtspunkte an interessanten Stellen")
– Browser verfügt über Schaltfläche, um Blickpunkte einer Szene nacheinander anzuspringen bzw. auszuwählen

`Viewpoint`	**`fieldOfView`**	SFFloat	Öffnungswinkel der Kamera
	`orientation`	SFRotation	Orientierung (Default: 0 0 1 0= Blickricht. neg. *z*-Achse)
	`position`	SFVec3f	Ort der Kamera
	`jump`	SFBool	
	`description`	SFString	Beschreibung
	`set_bind`	SFBool	eventIn-Feld
	`bindTime`	SFTime	eventOut-Feld
	`isbound`	SFBool	eventOut-Feld

Beispiel:
Def. zweier Blickpunkte von 2 Seiten (aus Richtung d. pos. und d. neg. *z*-Achse):

```
Viewpoint
    {
    position 0 0 50
    orientation 0 0 1 0  # Voreinstellung
    description "Sicht von Norden"
    }
Viewpoint
    {
    position 0 0 -50
    orientation 0 1 0 3.14
    description "Sicht von Süden"
    }
```

Mehrfachverwendung desselben Knotens
(*object instancing*)

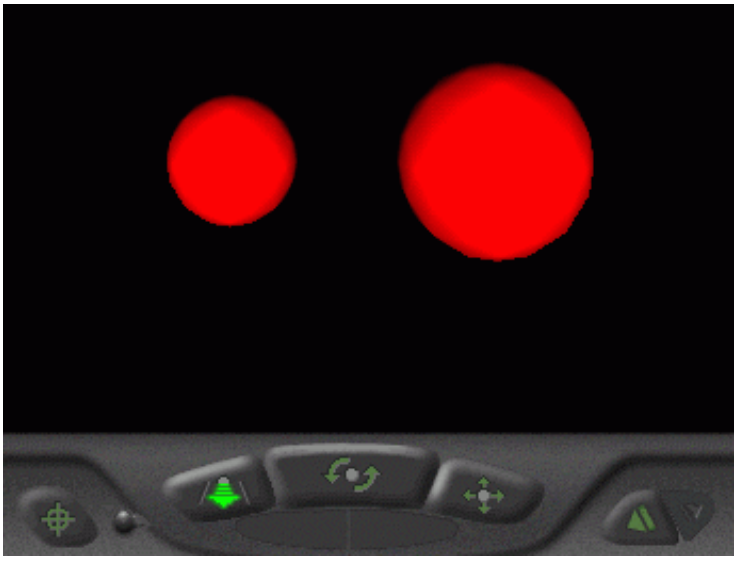
DEF und **USE**

- Dem Knoten muss mit **DEF** ein Name zugewiesen werden
- mit **USE** dann Referenz vermittels dieses Namens
- die neuen Instanzen sind lediglich Zeiger (Referenzen)
- können transformiert, aber nicht mit neuen Attributen versehen werden

Beispiel:

```
#VRML V2.0 utf8
DirectionalLight {}
DEF Ball Shape          # neuer Name
    {
    geometry Sphere { radius 0.5 }
    appearance Appearance
        {
       material Material
          { diffuseColor 1 0 0 }
        }
    }
Transform
    {
    scale 1.5 1.5 1.5
    translation 2 0 0
    children
        [
       USE Ball    # Referenz
       ]
    }
```

Ergebnis:



Definition eigener Knotentypen:
Prototypen

Schlüsselwort "PROTO"
2-teilige Definition:

- Deklaration der Felder, (z.T.) öffentlich
- Definition, privat (gekapselt) – eigentliche Objektbeschreibung
  – darin Bezugnahme auf die neuen Felder mit Schlüsselwort "IS"
  umfasst einzelnen Knoten oder ganze "Mini-Szene"
- Parametrisierung über den öffentlichen Deklarationsteil
- PROTO-Definition allein erzeugt noch kein neues Objekt!
- Verwendung in Szenen genauso wie die vorgegebenen Knotentypen

Jede Instanz eines Prototyps ist unabhängig von allen anderen, Änderungen einer Instanz betreffen nur sie selbst
(– Gegensatz zu DEF / USE !)

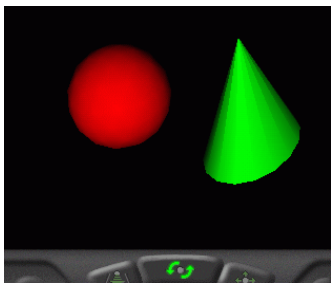
- der erste (oberste) Knoten in der Prototyp-Definition vererbt den Knotentyp (Verwendungszweck) an den Prototyp
- Gültigkeitsbereich von Namen innerhalb einer Prototyp-Def. ist auf diese Def. beschränkt

Beispiel 1:
Def. eines neuen Knotentyps für geometrische Objekte, um Position, Farbe etc. komfortabler angeben zu können
der neue Knotentyp soll "Koerper" heißen

```
#VRML V2.0 utf8

PROTO Koerper
   [            # Deklaration
   exposedField SFVec3f verschiebe 0 0 0
   exposedField SFVec3f verzerre 1 1 1
   exposedField SFColor farbe 0.8 0.8 0.8
   exposedField MFString adressen [ ]
   exposedField SFNode geometrie NULL
   ]
```

```
    {            # Definition
    Transform
       {
       translation IS verschiebe  # Schnittstelle
       scale IS verzerre          # nach außen!
       children
          [
          Shape
             {
             geometry IS geometrie
             appearance Appearance
                {
                material Material
                   { diffuseColor IS farbe }
                texture ImageTexture
                   { url IS adressen }
                }
             }
          ]
       }
    }
# Verwendung des Prototyps
Koerper  # rote Kugel
    {
    farbe 1 0 0
    verschiebe -2 0 0
    geometrie Sphere { }
    }
Koerper  # gruener Kegel, flachgedrueckt
    {
    farbe 0 1 0
    verschiebe 1 1 0
    verzerre 0.7 1 1
    geometrie Cone { }
    }
```

Ergebnis:



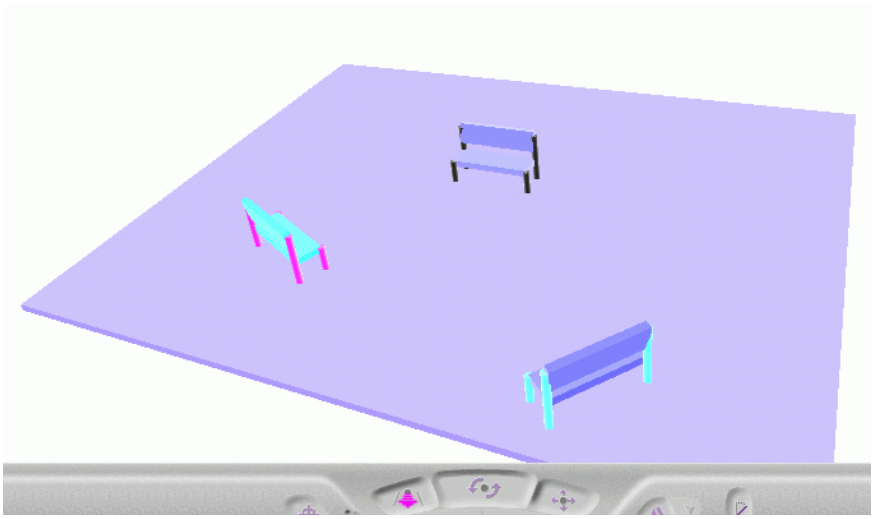
Beispiel 2:
Nicht nur ein einzelner Körper, sondern ein Arrangement (Mini-Szene) wird als Prototyp definiert – hier eine Sitzbank, anschließend werden mehrere Instanzen auf einer Bodenfläche verteilt

```
#VRML V2.0 utf8
# proto2.wrl: Drei Instanzen des Parkbank-Prototyps
PROTO Bank    # Prototyp einer Parkbank
   [
   exposedField SFColor SitzFarbe 1 0 0 # Deklaration
   exposedField SFColor BeinFarbe 0 1 0
   ]
   {
   Group                                  # Definition
      {
      children
         [
         Transform     # Sitz
            {
            translation 0 0.03 0
            children
               DEF Sitz Shape
                  {
                  appearance Appearance
                     {
                     material Material
                        { diffuseColor IS SitzFarbe }
                     }
                  geometry Box
                     { size 0.6 0.05 0.2 }
                  }
            }
         Transform     # Lehne
            {
            translation 0 0.2 -0.1
            rotation 1 0 0 1.4
            scale 1 1 0.8
            children USE Sitz # interne
                              # Wiederverwendung!
            }
```

```
            Transform     # kurzes Bein
               {
               translation -0.3 -0.05 0.1
               children
                  DEF Bein Shape
                     {
                     appearance Appearance
                        {
                        material Material
                           { diffuseColor IS BeinFarbe }
                        }
                     geometry Cylinder
                        { height 0.2 radius 0.025 }
                     }
               }
            Transform     # 2. kurzes Bein
               {
               translation 0.3 -0.05 0.1
               children USE Bein
               }
            Transform     # 1. langes Bein
               {
               translation -0.3 0.05 -0.1
               scale 1 2 1
               children USE Bein
               }
            Transform     # 2. langes Bein
               {
               translation 0.3 0.05 -0.1
               scale 1 2 1
               children USE Bein
               }
            ]
         }                # Group
      }                   # Prototyp Bank - Ende
   DEF Szene Group
      {
      children
         [
         DEF Bank1 Transform   # Instanz 1
            {
            translation -1 -0.85 2
            rotation 0 1 0 2.7
            children Bank { }
            }
```

```
   DEF Bank2 Transform   # Instanz 2
      {
      translation 0 -0.85 0.7
      children Bank
         {
         SitzFarbe 0.5 0.5 0
         BeinFarbe 1 1 1
         }
      }
   DEF Bank3 Transform   # Instanz 3
      {
      translation 1.3 -0.85 2.5
      rotation 0 1 0 4
      children Bank
         {
         SitzFarbe 0.7 0.7 0
         BeinFarbe 0.7 0 0
         }
      }

   Transform    # Boden: Extra-Element (nicht aus
                # Prototyp)
      {
      translation 0 -1 1
      children
         Shape
            {
            appearance Appearance
               {
               material Material
                   { diffuseColor 0.4 0.5 0.01 }
               }
            geometry Box
               { size 5 0.05 4 }
            }
      }
   ]

}  # Szene (Group)
```

Ergebnis (hier wegen besserer Darstellbarkeit farblich invertiert):



Prototyp-Definitionen können auch in externe Dateien verlagert werden
(⇒ Szenengestaltung als Kombination und Variation vordefinierter Objektsammlungen aus Bibliotheken):

EXTERNPROTO-Definition

- Definitionsteil besteht nur noch aus URL-String
- Default: Zugriff auf die erste Prototyp-Definition in der referenzierten Datei
- Prototyp-Name kann an den Dateinamen mit "#" angehängt werden

Beispiel:

```
#VRML V2.0 utf8
# exproto1.wrl: Zugriff auf externen Prototyp
EXTERNPROTO Parkbank     # externer Prototyp
   [
   exposedField SFColor SitzFarbe   # Dekl.
   ]
   "pro_bank.wrl#Bank"   # Endung "#Bank" hier
                         # optional
```

```
DEF Szene Group
   {
   children
      [
      DEF Bank1 Transform   # Instanz 1
         {
         translation -2 -0.85 2.5
         rotation 0 1 0 2.7
         children Parkbank {}
         }
```
(... usw.)

Beachte: Die Namen der Prototypen in der Datei und in der EXTERNPROTO-Deklaration müssen nicht übereinstimmen.

Knoten, die Darstellungen in Abhängigkeit vom Benutzerstandpunkt ändern

1. *LOD (level of detail) - Knoten*

dient der Bereitstellung alternativer, verschieden genauer Darstellungen von Objekten für unterschiedliche Entfernungsintervalle

LOD	**range**	MFFloat	Reichweite-Intervallgrenzen (in aufsteigender Folge) – eine weniger als die Zahl der Intervalle
	level	MFNode	Auflistung der alternativen Objekte

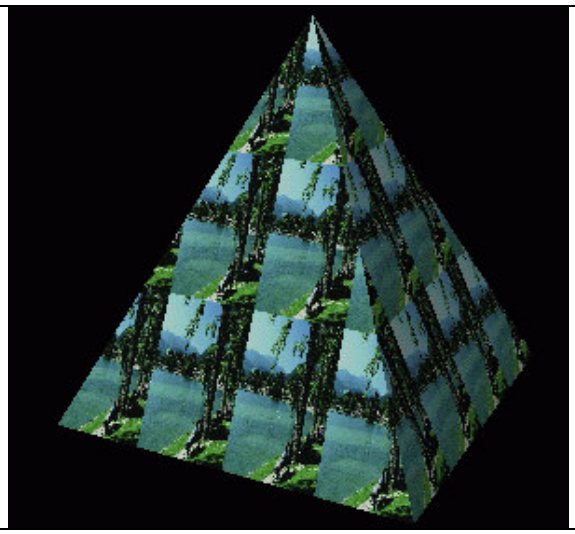
Beispiel: Eine Pyramide wird aus der Nähe mit genauem Texturmuster (aus einer Bilddatei) und aus mittlerem Abstand einfarbig dargestellt. In großem Abstand verschwindet sie ganz.

Das Beispiel zeigt außerdem die Verwendung von Texturkoordinaten zur systematischen Plazierung und Wiederholung von Texturen auf **IndexedFaceSet**-Flächen.

```
#VRML V2.0 utf8
PROTO Pyramide
   [
   exposedField SFNode app NULL
   ]
   {
   Shape
      {
      appearance IS app
      geometry IndexedFaceSet
         {
         coord Coordinate
            {
            point [ 1 0 0, 0 0 1, -1 0 0, 0 0 -1,
                    0 1.5 0 ]
            }
         coordIndex [ 0 1 2 3 -1,
                      1 0 4 -1,
                      2 1 4 -1,
                      3 2 4 -1,
                      0 3 4 ]
         texCoord TextureCoordinate
            {
            point [ -2 -2, 2 -2, -2 0, 2 0,
                    0 3 ]
            }
         texCoordIndex
            [ 0 1 2 3 -1,
              2 3 4 -1,
              2 3 4 -1,
              2 3 4 -1,
              2 3 4 ]
         }
      }
   }
LOD
   {
   range [ 7, 15 ]
   level
      [
      Pyramide # level 0: bis Abst. 7 vom Betrachter
         {
         app Appearance  # genauestes Modell
            {
            material Material { }
```

```
            texture ImageTexture
               { url "wiessee.jpg" }
            }
         },
      Pyramide  # level 1: zwischen Abstand 7 und 15
         {
         app Appearance  # groeberes Modell
            {
            material Material
               { diffuseColor 0 0.7 0.6 }
            }
         },
      Shape { } # level 2: Abstand > 15 (nichts mehr)
   ]
}
```

Ergebnis:



aus der Nähe

aus mittlerem Abstand

2. *Billboard-Knoten*

- Projektion eines 2D-Bildes auf ein total transparentes Objekt
- das Objekt weist dem Betrachter immer dieselbe Seite zu (Rotation um festgelegte Achse, oder unabhängig von jeglicher Ausrichtung: *Screen Alignment*)
- nützlich bei Objekten, die als komplettes 3D-Modell sehr viel Platz und Zeit erfordern würden und wo man "nicht genau hinsieht" (z.B. Bäume im Hintergrund)

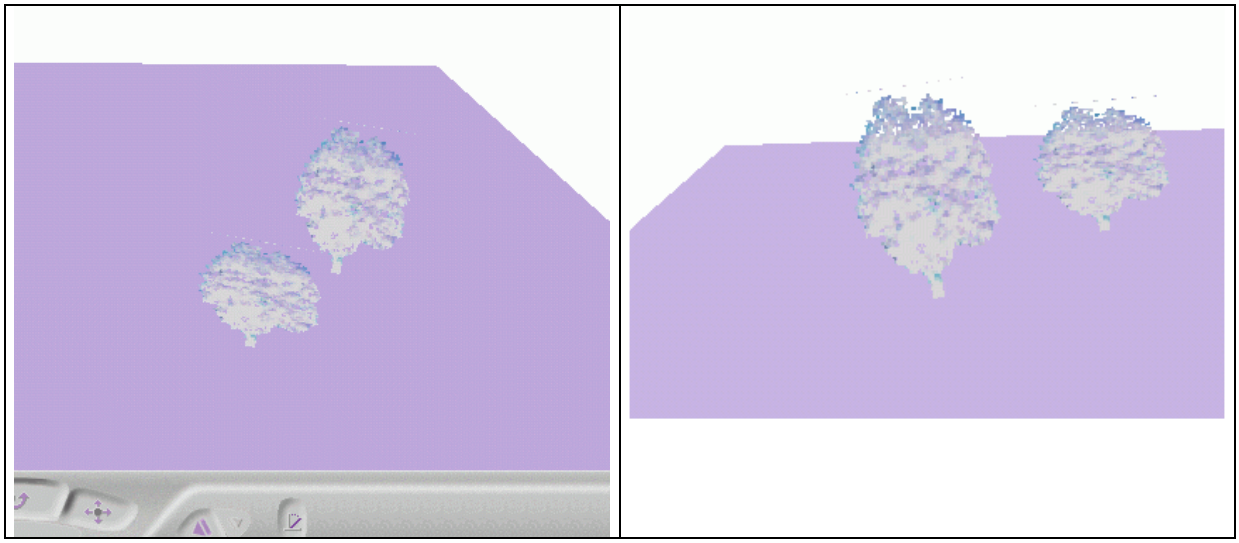
Billboard	**axisOfRotation**	SFVec3f	Drehachse (Default: 0 1 0, 0 0 0 = Screen Alignment)
	children	MFNode	Kindknoten

Beispiel: 2 Bäume als Billboards (flache Tafeln), Verwendung eines Fotos "baum.gif"

```
#VRML V2.0 utf8
# billboard.wrl: Baum als Billboard

Transform     # Baum
   {
   translation 3 -0.5 3
   children
      [
      DEF Baum Billboard   # Billboard-Anfang
         {
         axisOfRotation 0 1 0
         children
            Shape
               {
               appearance Appearance
                  {
                  texture ImageTexture
                     {
                     url "baum.gif"
                     repeatS FALSE
                     repeatT FALSE
                     }
                }
               geometry Box
                   { size 1 1 0.01 }
               }
         }                  # Billboard-Ende
      ]
   }
```

```
Transform                    # Baum 2
    {
    translation 4 -0.25 2
    scale 1 1.5 1
    children USE Baum    # Instanziierung
    }

Transform                    # Boden
    {
    translation 0 -1 1
    children
        Shape
            {
            appearance Appearance
                {
                material Material
                    { diffuseColor 0.6 0.8 0.3 }
                }
            geometry Box
                { size 14 0.01 8 }
            }
    }
```

Ergebnis aus 2 Ansichten (Farben wieder invertiert):



3. *Nebel*

- einstellbar: Farbe des Nebels, Stärke, Nebeltyp (exponentielle oder lineare Abnahme der Sichtbarkeit)
- auch für Dunst geeignet

Fog	**color** **fogType** **visibilityRange**	SFColor SFString SFFloat	Farbe "LINEAR", "EXPONENTIAL"

Als Beispiel wird eine Szene mit einer Kirche und einer Laterne in Nebel getaucht:

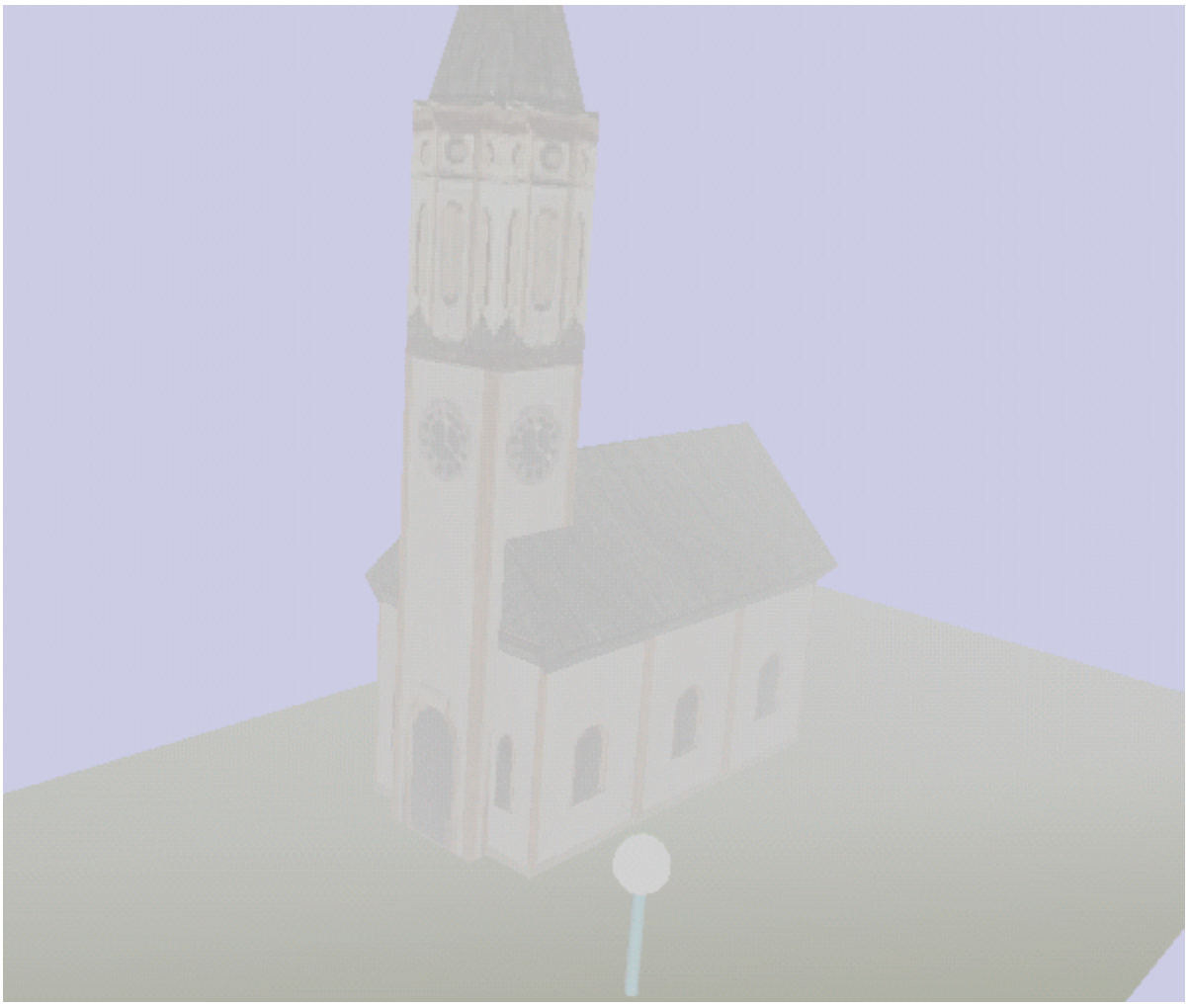
```
#VRML V2.0 utf8
# kirche.wrl

DEF Laterne Transform
   {
   translation -2.5 0 2.5
   children
      [
      PointLight {}  # Punktlichtquelle
      DEF Lampe Shape
         {
         appearance Appearance
            {
            material Material
               { emissiveColor 1 1 1 }
            }
         geometry Sphere
            { radius 0.2 }
         }
      DEF Mast Transform
         {
         translation 0 -0.55 0
         children
            Shape
               {
               appearance Appearance
                  {
                  material Material
                     {
                     ambientIntensity 1
```

```
                    diffuseColor 0.3 0.9 1
                    }
                }
            geometry Cylinder
                {
                radius 0.05
                height 0.9
                }
            }
        }
    ]
}

DEF Kirche Transform
    {
    translation 0 0 0
    children
        [
        DEF Schiff Shape
            {
            appearance Appearance
                {
                texture ImageTexture
                    {
                    url "seite.jpg"
                    repeatS FALSE  # Verzerren
                    repeatT FALSE  # statt Wiederholen
                    }
                }
            geometry Box
                { size 4 2 2 }
            }
        DEF Dach Transform
            {
            rotation 1 0 0 0.785  # 45-Grad-Drehung
            translation 0 1 0
            children
                Shape
                    {
                    appearance Appearance
                        {
                        texture ImageTexture
                            { url "dach.jpg" }
                        }
                    geometry Box { size 3.95 1.75 1.75 }
                    }
            }
```

```
      DEF Turm Transform
         {
         translation -1.7 2 0
         children
            Shape
               {
               appearance Appearance
                  {
                  texture ImageTexture
                     {
                     url "turm.jpg"
                     repeatS FALSE
                     repeatT FALSE
                     }
                  }
               geometry Box { size 1 6 1 }
               }
         }
      DEF Kuppel Transform
         {
         translation -1.7 6 0
         children
            Shape
               {
               appearance Appearance
                  {
                  texture ImageTexture
                     { url "dach.jpg" }
                  }
               geometry Cone
                  {
                  bottomRadius 0.6
                  height 2
                  }
               }
         }
      ]
   }
DEF Boden Transform
   {
   translation 0 -1 1
   children
      Shape
         {
         appearance Appearance
            {
            material Material
               { diffuseColor 0.5 0.6 0.01 }
```

```
            }
         geometry Box
            { size 14 0.01 8 }
         }
   }

Background
   {
   skyColor [ 0.8 0.8 0.9 ]
   }
Fog
   {
   color 0.8 0.8 0.8
   fogType "EXPONENTIAL"
   visibilityRange 15
   }
```

Ergebnis:



Interaktion zwischen Knoten

Mehrere VRML-Knotentypen können Ereignisse (*events*) empfangen und/oder senden.
Ereignisse enthalten einen Wert und eine Zeitmarke (*timestamp*).

- Gesendete Ereignisse können die Änderung von Werten von Feldern mitteilen
- Empfangene Ereignisse können Werte von Feldern ändern

Ereignisse sind ihrerseits als Felder von Knoten definiert.

Ereignis, das die Änderung eines Feldes mitteilt:
Schlüsselwort **`eventOut`**
Ereignisname meist = Feldname und Suffix "**`_changed`**"
(Ausnahmen: Ereignisse mit den Typen **`SFBool`** und **`SFTime`**.)

Ereignis, das ein Feld ändern kann:
Schlüsselwort **`eventIn`**
Ereignisname meist = Feldname und Präfix "**`set_`**"
(Ausnahmen: **`addChildren`**, **`removeChildren`**, Ereignisse vom Typ **`SFTime`**).

Bei Feldern, zu denen beide Typen von Ereignissen gehören, kann statt der Definition des Feldes und beider Ereignisse eine Kurzform verwendet werden: Voranstellen des Schlüsselwortes "**`exposedField`**" vor den Feldnamen. Beispiel:
Bei einem Feld "**`position`**" innerhalb einer Knotendefinition kann statt der drei Angaben

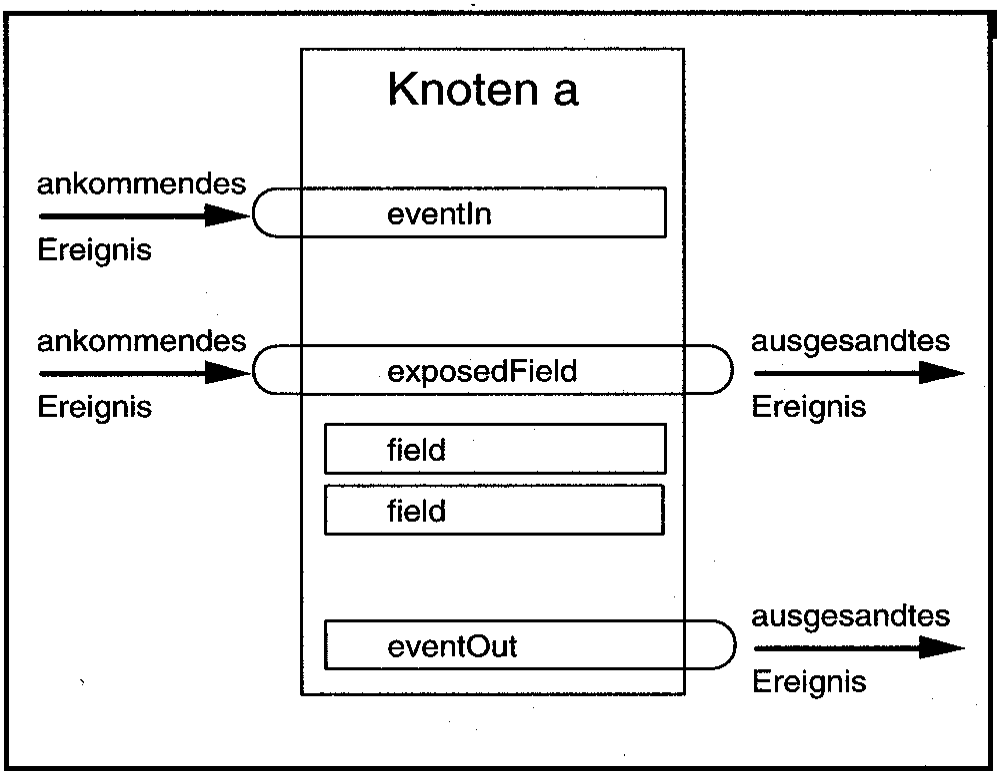
```
field position
eventIn set_position
eventOut position_changed
```

abgekürzt geschrieben werden:

```
exposedField position
```

Die Schlüsselwörter **`field`**, **`exposedField`**, **`eventIn`**, **`eventOut`** geben die *Zugriffsart* eines Feldes oder Ereignisses an.

Schnittstellen-Schema eines Knotens:



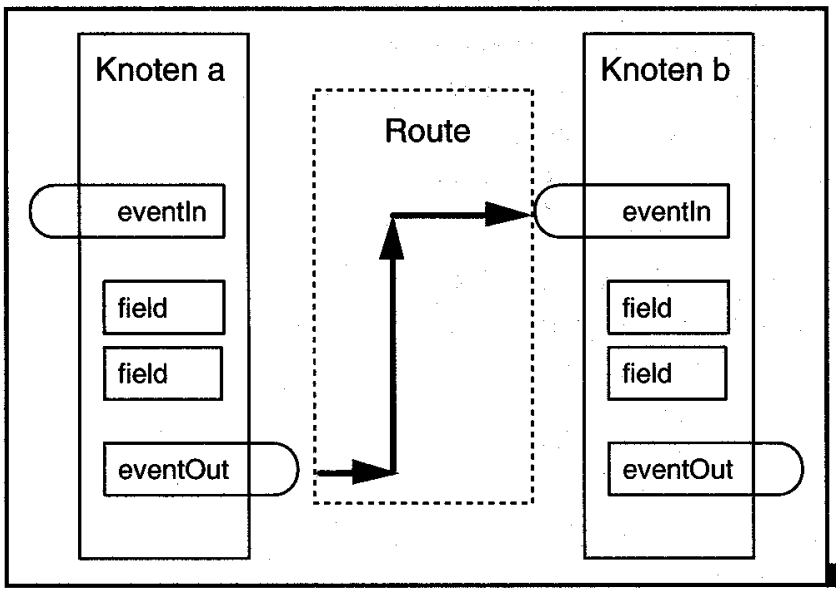
(Bei der Def. neuer Felder im Rahmen von Prototypen sind die Namenskonventionen für Präfix bzw. Suffix nicht zwingend vorgeschrieben, werden aber zur Verbesserung der Konsistenz und Lesbarkeit empfohlen.)

Ereignistyp: Typ des geänderten bzw. des zu ändernden Feldes.

Ereignisse werden durch *Routen* zwischen zwei Knoten übertragen.
Dabei können nur Knoten verwendet werden, die mit dem Schlüsselwort **`DEF`** einen Namen zugeordnet bekommen haben (siehe Kursteil 3a).
Voraussetzung: Die Ereignistypen des Output- und des Input-Ereignisfeldes müssen übereinstimmen.

Syntax der Routen-Vereinbarung:

`ROUTE` *NameKnotenA.feld1*`_changed` **`TO`** *NameKnotenB*`.set_`*feld2*



Ausnahmen von der Namenskonvention "**_changed**" für Output- und "**set_**" für Inputereignisse:

Ereignisse mit booleschen Werten: **is**... (z.B. **isActive**)
Ereignisse mit Zeitwerten: ...**Time** (z.B. **touchTime**)
Ereignisse zum Hinzufügen und Entfernen von Kindknoten:
add_children, **remove_children**

In der Routendefinition können "**set_**" und "**_changed**" bei Ereignissen von Feldern mit der Zugriffsart **exposedField** fortgelassen werden.
Beispiel:
bei

```
DEF Schalter TouchSensor { enabled TRUE }
DEF Licht DirectionalLight { on FALSE }
ROUTE Schalter.enabled_changed TO Licht.set_on
```

kann die letzte Zeile vereinfacht werden zu:

```
ROUTE Schalter.enabled TO Licht.on
```

Ein per Route weitergeleitetes Ereignis kann bei der Verarbeitung durch den empfangenden Knoten ein weiteres Ereignis auslösen, das wiederum weitergeleitet wird, usw.: *Ereignis-Kaskade*.

Alle Ereignisse einer Kaskade haben dieselbe Zeitmarke.
Bei der Konstruktion von Ereignis-Kaskaden dürfen keine Schleifen gebildet werden!

Zwei oder mehr Ereignisse gehen vom selben **`eventOut`**-Ereignis an verschiedene Knoten: "*fan-out*", erlaubt.

Zwei oder mehr Ereignisse mit derselben Zeitmarke werden an dasselbe **`eventIn`**-Ereignis geleitet: "*fan-in*", Ergebnis undefiniert, deshalb verboten.

Animation (seit VRML 2.0)

Antrieb der Veränderung durch spezielle Knoten, z.B. **`TimeSensor`**, die Zeitwerte an Interpolator-Knoten senden. Diese erzeugen die gebrauchten (Zwischen-) Werte für Felder animierter Objekte.

`TimeSensor`	**`enabled`**	**`SFBool`**	Start/Stop
	`cycleInterval`	**`SFTime`**	Zykluszeit
	`loop`	**`SFBool`**	unendl. Wiederholung
	`startTime`	**`SFTime`**	Start des Timer-Ablaufs
	`stopTime`	**`SFTime`**	Lebensdauer des Timers
	`fraction_changed`	**`SFFloat`**	**`eventOut`**-Feld, Anteil des bereits verstrichenen Zeitintervalls (zw. 0 und 1)
	`isActive`	**`SFBool`**	**`eventOut`**
	`time`	**`SFTime`**	**`eventOut`**
	`cycleTime`	**`SFTime`**	**`eventOut`**, Start eines Zyklus

Die Interpolator-Knoten stellen anwendungsspezifische Interpolationswerte bereit:

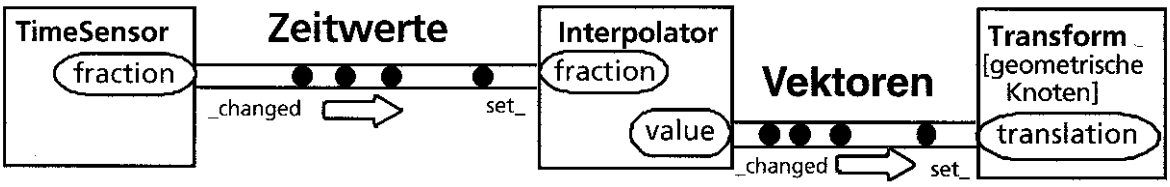
OrientationInterpolator	**set_fraction** **key** **keyValue** **value_changed**	**SFFloat** **MFFloat** **MFRotation** **SFRotation**	**eventIn** Liste von Zwischenwerten (mindest. 0 und 1) Liste v. Zwischenwerten für die interpolierte Größe **eventOut**
PositionInterpolator	**set_fraction** **key** **keyValue** **value_changed**	**SFFloat** **MFFloat** **MFVec3f** **SFVec3f**	
CoordinateInterpolator **NormalInterpolator**	**set_fraction** **key** **keyValue** **value_changed**	**SFFloat** **MFFloat** **MFVec3f** **MFVec3f**	
ScalarInterpolator	**set_fraction** **key** **keyValue** **value_changed**	**SFFloat** **MFFloat** **MFFloat** **SFFloat**	
ColorInterpolator	**set_fraction** **key** **keyValue** **value_changed**	**SFFloat** **MFFloat** **MFColor** **SFColor**	

Wirkung: Wert von **key** (zwischen 0 und 1) wird in Wert von **keyValue** "übersetzt" (durch lineare Interpolation zwischen den passenden Stützstellen). Die Länge der **keyValue**-Liste muss ein ganzzahliges Vielfaches der Länge der **key**-Liste sein (meist stimmen die Längen beider Listen überein). Die

Kommunikation erfolgt über **`set_fraction`** (für den aktuellen Input) und **`value_changed`** (für den zugehörigen Output).

Beachte: Bei der Interpolation von Farben wird intern das HSV-Modell benutzt (obwohl die Farbwerte in **`keyValue`** und **`value_changed`** als RGB-Farbwerte spezifiziert werden).

Die Animation einer Szene mittels Interpolator-Knoten erfordert somit mindestens 2 ROUTE-Definitionen. Beispielsweise bei einer Positionsveränderung (Vektoren als Zwischenwerte):

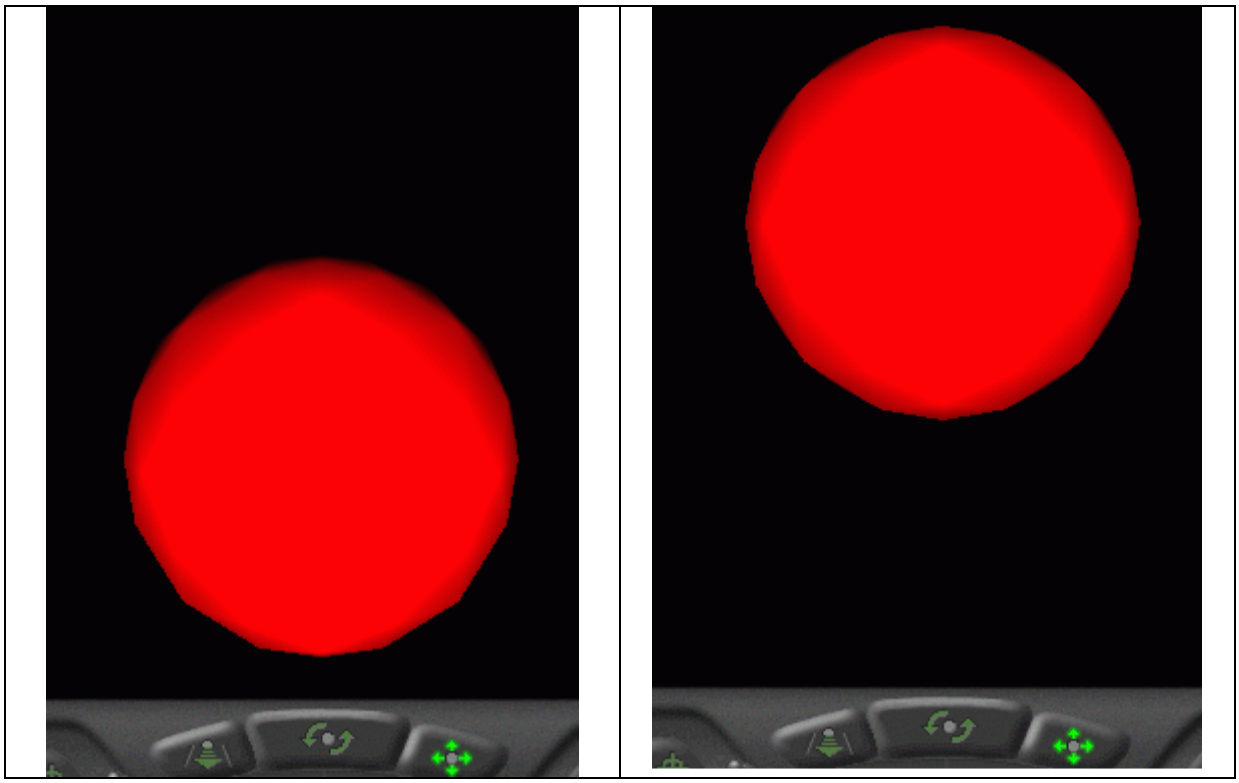


Beispiel:

Animation eines ständig hüpfenden Balls. Das Zeitintervall für den Vorgang (der unendlich wiederholt wird) wird dem **`TimeSensor`**-Knoten mit 2 Sek. vorgegeben. Der **`PositionInterpolator`**-Knoten enthält äquidistante Stützstellen im Feld **`key`** und zugehörige Punkte auf einer umgekehrten Parabel im Feld **`keyValue`**, um bei der Bewegung des Balles den Eindruck von Schwerkraft zu vermitteln.

```
#VRML V2.0 utf8
#Animation: Huepfender Ball

DirectionalLight
   { direction 0 0 -1 }
DEF Chronos TimeSensor
   {
   cycleInterval 2
   loop TRUE
   startTime 1
   }
DEF PosCalc PositionInterpolator
   {
   key [ 0 0.1 0.2 0.3 0.4 0.5
         0.6 0.7 0.8 0.9 1 ]
   keyValue [ 0 0 0, 0 0.09 0, 0 0.16 0,
              0 0.21 0, 0 0.24 0, 0 0.25 0,
              0 0.24 0, 0 0.21 0, 0 0.16 0,
              0 0.09 0, 0 0 0 ]
   }
DEF Verschieb Transform
   {
   children
      [
      Shape
         {
         geometry Sphere { radius 0.2 }
         appearance Appearance
            {
            material Material
               { diffuseColor 1 0 0 }
            }
         }
      ]
   }
ROUTE Chronos.fraction_changed TO
   PosCalc.set_fraction
ROUTE PosCalc.value_changed TO
   Verschieb.set_translation
```

Ergebnis (2 Snapshots aus der Animationssequenz):



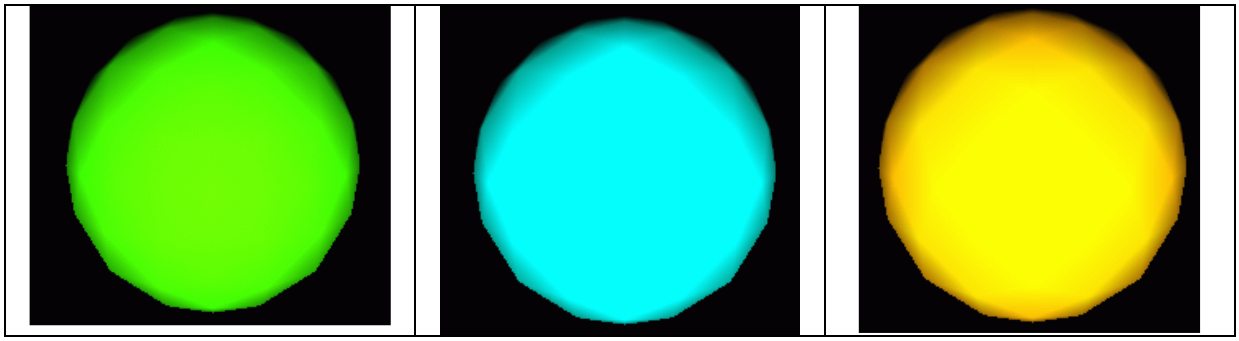
Beispiel einer Farbinterpolation:
Farbwechsel einer Kugel

Die hierfür benötigten Ergebnisse des **`ColorInterpolator`**-Knotens werden an eines der Felder im **`Material`**-Knoten weitergeleitet, hier an **`diffuseColor`** (ein **`exposedField`**):

```
#VRML V2.0 utf8
#Animation: Farbwechsel einer Kugel

DEF Chronos TimeSensor
    {
    cycleInterval 10
    loop TRUE
    startTime 1
    }
```

```
DEF FarbCalc ColorInterpolator
    {
    key [ 0 0.333 0.667 1 ]
    keyValue [ 1 0 0, 0 1 0, 0 0 1, 1 0 0 ]
    }
Transform
    {
    children
        [
        DirectionalLight { }
        Shape
            {
            geometry Sphere { }
            appearance Appearance
              {
              material DEF KugelFarbe Material { }
              }
            }
        ]
    }
ROUTE Chronos.fraction_changed TO
    FarbCalc.set_fraction
ROUTE FarbCalc.value_changed TO
    KugelFarbe.set_diffuseColor
```

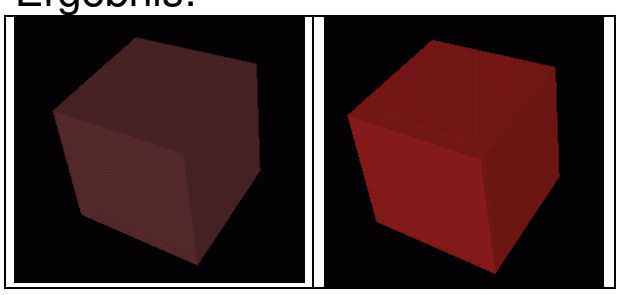
Ergebnis (3 Schnappschüsse):



Beispiel eines Blinkers:

```
#VRML V2.0 utf8

Transform
    {
    children
        [
        Shape
            {
            geometry Box { }
            appearance Appearance
                {
                material DEF Blinker Material { }
                }
            }
        ]
    }
DEF Timer TimeSensor
    {
    cycleInterval 5.0  # 5 Sek. Zyklusintervall
    loop TRUE
    }
DEF Farbgeber ColorInterpolator
    {
    key [ 0.0, 1.0 ]
    keyValue [ 0 0 0, 1 0 0 ]
    }
ROUTE Timer.fraction_changed TO
    Farbgeber.set_fraction
ROUTE Farbgeber.value_changed TO
    Blinker.set_diffuseColor
```

Ergebnis:



Interaktion

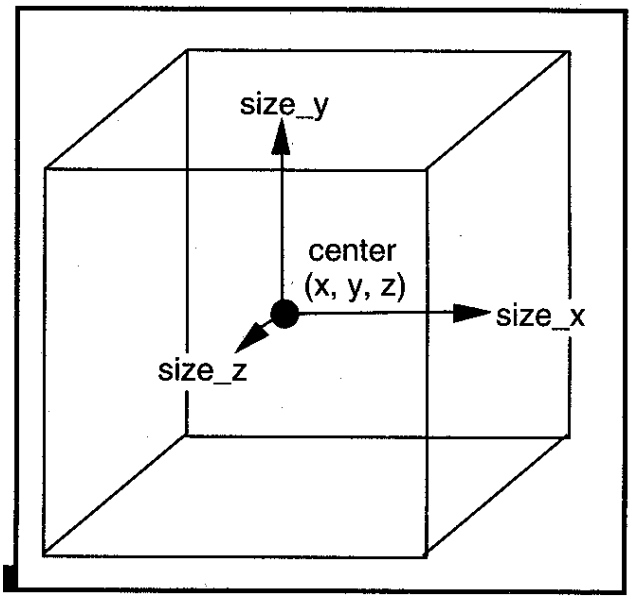
Die Einbeziehung von Benutzer-Eingaben erfordert *Sensorknoten* (von denen der **`TimeSensor`** bereits ein Sonderfall war):

`TouchSensor`	Berührungssensor (wird aktiv bei Anklicken mit der Maus)
`ProximitySensor`	Annäherungssensor, spezifiziert durch umgebende Box
`PlaneSensor`	Abbildung der Mauscursor-Bewegung in eine Ebene parallel zur *xy*-Ebene
`CylinderSensor`	Abb. der Mauscursor-Bewegung auf eine Drehung einer Zylinderfläche mit Zylinderachse parallel zur y-Achse
`SphereSensor`	Abb. der Mauscursor-Bewegung auf eine Drehung einer Kugelfläche
`VisibilitySensor`	Ermittlung der Sichtbarkeit einer Box

`Touch-Sensor`	**`enabled`**	**`SFBool`**	Start/Stop des Sensors
	`hitPoint_changed`	**`SFVec3f`**	)
	`hitNormal_changed`	**`SFVec3f`**	)
	`hitTexCoord_changed`	**`SFVec2f`**	) **`eventOut`**-
	`isActive`	**`SFBool`**	) Felder
	`isOver`	**`SFBool`**	)
	`touchTime`	**`SFTime`**	)
`Proxi-mity-Sensor`	**`center`**	**`SFVec3f`**	Mittelpkt. der Box
	`size`	**`SFVec3f`**	Ausdehnung
	`enabled`	**`SFBool`**	Start/Stop
	`position_changed`	**`SFVec3f`**	)
	`orientation_changed`	**`SFRota-tion`**	) **`eventOut`**-) Felder
	`enterTime`	**`SFTime`**	)
	`exitTime`	**`SFTime`**	)
	`isActive`	**`SFBool`**	)

Plane-Sensor	**autoOffset**	**SFBool**	Flag für Additivität der Bewegungskoordinaten
	enabled	**SFBool**	Start/Stop
	offset	**SFVec3f**	Offset zur Koord.-ausg.
	maxPosition	**SFVec2f**	Einschränkung der Sensoraktivität
	minPosition	**SFVec2f**	
	trackPoint_changed	**SFVec3f**	) **eventOut**-
	translation_changed	**SFVec3f**	) Felder
	isActive	**SFBool**	)
Cylinder-Sensor	**autoOffset**	**SFBool**	analog zum **Plane-Sensor**-Knoten
	enabled	**SFBool**	
	diskAngle	**SFFloat**	
	offset	**SFFloat**	
	maxAngle	**SFFloat**	
	minAngle	**SFFloat**	
	trackPoint_changed	**SFVec3f**	
	rotation_changed	**SFRotation**	
	isActive	**SFBool**	
Sphere-Sensor	**autoOffset**	**SFBool**	analog zum **Plane-Sensor**-Knoten
	enabled	**SFBool**	
	offset	**SFRotation**	
	trackPoint_changed	**SFVec3f**	
	rotation_changed	**SFRotation**	
	isActive	**SFBool**	
Visibility-Sensor	**center**	**SFVec3f**	Mittelpkt. Box
	size	**SFVec3f**	Ausdehnung
	enabled	**SFBool**	Start/Stop
	enterTime	**SFTime**	)
	exitTime	**SFTime**	) **eventOut**
	isActive	**SFBool**	)

Skizze zur Bedeutung der Felder des **ProximitySensor**-Knotens:

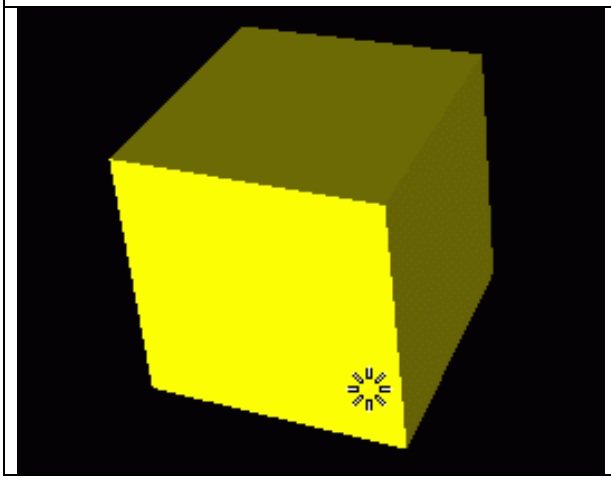
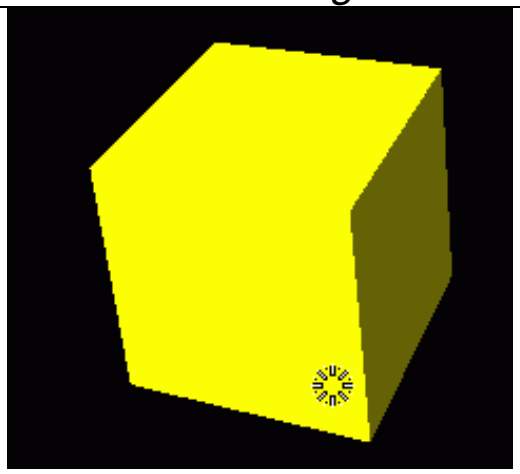


Beispiel:
Einschalten einer Lichtquelle mit dem **TouchSensor**

```
#VRML V2.0 utf8
# Lichtschalter

PointLight
   { location 0 0 10 }
DEF Licht2 PointLight
   {
   location 0 2 0
   on FALSE
   }
Transform
   {
   children
      [
      Shape
         {
         geometry Box { }
         appearance Appearance
```

```
            {
            material Material
               { diffuseColor 1 1 0 }
            }
         }
      DEF Schalter TouchSensor { }
      ]
   }
ROUTE Schalter.isActive TO Licht2.on
```

Ergebnis:

ohne Benutzereingriff (Licht von oben ist ausgeschaltet):	*Benutzer hat durch Anklicken des Würfels mit der Maus das Licht von oben eingeschaltet:*
	

Kollisionserkennung

Generell sind alle Objekte außer IndexedLineSet-, PointSet- und Text-Knoten mit der Fähigkeit des Erkennens von *Kollisionen* mit dem virtuellen Betrachter versehen.
Beispiele: Simulation des Anstoßens an eine Wand, Berührung von Objekten, "realistisches" Begehen von Labyrinthen etc.

Beachte: "Kollision" bezieht sich hier immer auf die Interaktion des Nutzers (bzw. seiner virtuellen Repräsentation im Cyberspace) mit einer (Objekt-) Geometrie, nicht auf Objekt-Objekt-Kollisionen!

Die Kollisionserkennung ist als Default *ein*gestellt.
Das Verhalten bei einer erfolgten Kollision ist vom Browser abhängig.
Die Reichweite der Prüfung auf Kollisionen zwischen Nutzer und virtuellen Objekten wird in einem **`NavigationInfo`**-Knoten im ersten Wert des Feldes **`avatarSize`** festgelegt.

Objektspezifische Einstellungen des Kollisionsverhaltens erfolgen mittels des **`Collision`**-Knotens.

Aufgaben dieses Knotens:

- Ein- und Ausschalten der Kollisionserkennung für seine Kindknoten
- Spezifikation eines Ersatzobjekts (**`proxy`**) mit einfacherer Geometrie oder einer bounding box zur Beschleunigung der Kollisionserkennung
- Aussenden von Ereignissen an andere Knoten bei Kollision, um Effekte zu generieren
- Setzen "unsichtbarer Wände" (wenn keine **`children`**, sondern nur ein **`proxy`**-Knoten definiert ist)

Felder des **Collision**-Knotens:

children	**MFNode**	Kindknoten
collide	**SFBool**	Ein-Aus-Schalter
proxy	**SFNode**	Ersatzobjekt
bboxCenter	**SFVec3f**	) boundig box-Spezifikation
bboxSize	**SFVec3f**	)
addChildren	**MFNode**	**eventIn**
removeChildren	**MFNode**	**eventIn**
collideTime	**SFTime**	**eventOut** (Zeitmarke der Kollision)

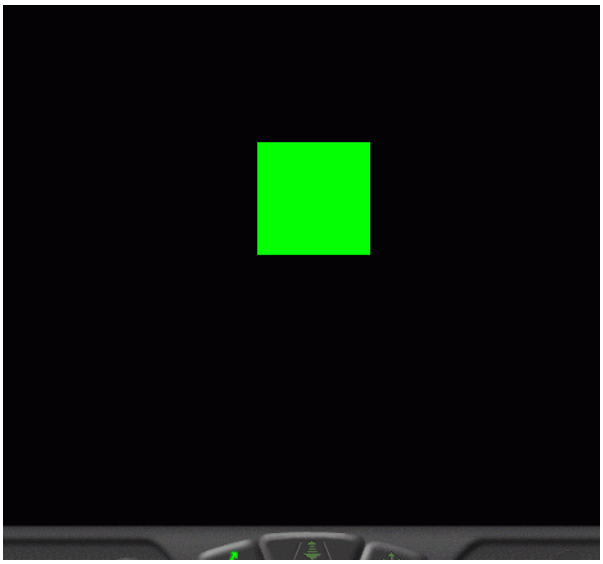
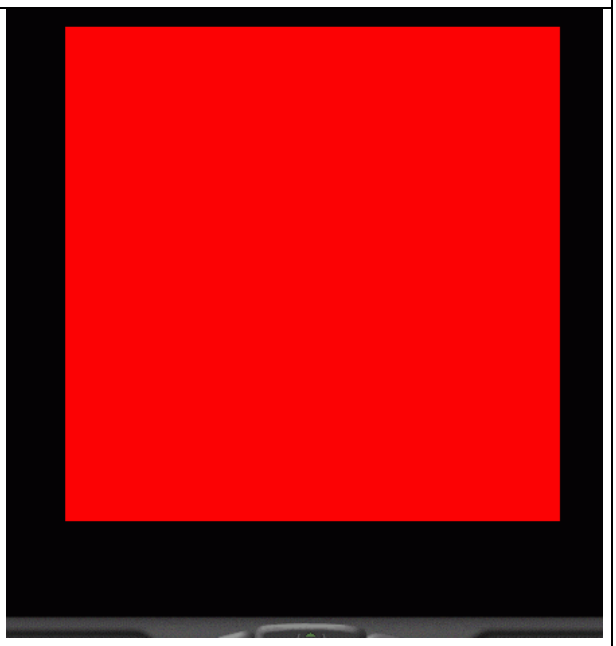
Ist der **Collision**-Knoten der Wurzel-Knoten einer VRML-Szene und steht **collide** auf FALSE, so ist die Kollisionserkennung für die gesamte Szene deaktiviert, unabhängig davon, ob darunterliegende Knoten **collide** auf TRUE gesetzt haben oder nicht.
Die bounding box wird deaktiviert durch den (Default-) Wert **-1 -1 -1** für **bboxSize**.
Ist **collide** auf TRUE gesetzt und **proxy** nicht definiert (= **NULL**), so werden die **children**-Knoten zur Kollisionserkennung verwendet, sonst der **proxy**-Knoten, welcher aber nicht visuell dargestellt wird.

Beispiel:
Ein Würfel wird von einem größeren **proxy**-Würfel umgeben. Bei Kollision mit dem unsichtbaren, äußeren Würfel ändert der innere Würfel seine Farbe von Grün nach Rot:

```
#VRML V2.0 utf8

Transform
   {
   translation 0 0 -5
   children
      [

      DEF Tabu_Zone Collision
         {
         collide TRUE
         children
            [
            Shape
               {
               geometry Box { }
               appearance Appearance
                  {
                  material DEF BoxFarbe
                     Material
                     { diffuseColor 0 1 0 }
                  }
               }
            ]
         proxy Shape
            {
            geometry Box { size 8 8 8 }
            }
         }
      ]
   }
DEF Timer TimeSensor { }
DEF FarbCalc ColorInterpolator
   {
   key [ 0 1 ]
   keyValue [ 1 0 0, 1 0 0 ]
   }
ROUTE Tabu_Zone.collideTime_changed TO
   Timer.startTime
```

```
ROUTE Timer.fraction_changed TO
   FarbCalc.set_fraction
ROUTE FarbCalc.value_changed TO
   BoxFarbe.set_diffuseColor
```

Ergebnis (abhängig von Bewegungen des Benutzers):

Benutzer hält Abstand ein	*Benutzer ist mit dem unsichtbaren Außen-Würfel kollidiert*
	

Eine größere Annäherung als auf dem rechten Bild ist nicht mehr möglich.

Switch-Knoten

Funktion: Alternative Objekt-Auswahl je nach Zahlenwert eines Diskriminator-Feldes **`whichChoice`** (vgl. switch-Konstrukt in Java und C)

Felder des **`Switch`**-Knotens:

`choice`	**`MFNode`**	Liste der Alternativ-Knoten
`whichChoice`	**`SFInt32`**	Diskriminator: Index des darzustellenden Kind-Knotens

Die implizite Indizierung der Kind-Knoten ist 0; 1; 2; ... in der Reihenfolge ihrer Auflistung. Ist der Wert von **`whichChoice`** kleiner als 0 oder größer als die Anzahl der vorhandenen Kindknoten – 1, so erfolgt gar keine Wiedergabe.
(Beispiel nach Einführung des **`Script`**-Knotens.)

Einbau von Skripten

Der **`Script`**-Knoten ermöglicht die Kommunikation mit prozeduralen und objektorientierten Programmen und damit die Verfügbarmachung einer intelligenteren Programmierlogik in VRML-Szenen, als es mit den Mitteln von VRML allein möglich wäre.

Anwendungen:

- mathematische Operationen (Arithmetik, vektorielle Berechnungen)
- Hilfsoperationen (z.B. Typkonvertierungen, Zerlegen zusammengesetzter Werte in ihre Komponenten u. umgekehrt)
- Zwischenspeichern von Werten, die bestimmte Zustände beschreiben
- Flexibilisierung der Ereignisverarbeitung
- Simulationen
- Multi-User-Anwendungen

unterstützte Programmiersprachen:

- Javascript (ECMAScript) und VRMLScript (inline oder in externen Dateien)
- Java (nur externe Bytecode-Dateien, Endung **`.class`**)

`Script`-Knoten werden ähnlich wie Prototypen definiert: In ihrer Schnittstellen-Beschreibung können beliebig viele Felder und Ereignisse definiert werden, wobei die Felder zur Speicherung von Werten und die Ereignisse der Kommunikation mit Knoten der Szene dienen.
Die Zugriffsart **`exposedField`** wird in **`Script`**-Knoten nicht unterstützt.
Zu und von den Ereignissen der **`Script`**-Knoten können die üblichen Routen vereinbart werden. Darüberhinaus verfügen **`Script`**-Knoten über eine direkte Interaktionsmöglichkeit mit anderen Knoten, sofern diese direkt in einem seiner Felder definiert sind (Feldtyp **`SFNode`** oder **`MFNode`**) oder wenn aus einem solchen Feld eine Referenz mit **`USE`** auf einen benannten Knoten (anderswo def.) hergestellt wurde. Für die direkte Interaktion muss das Feld **`directOutput`** auf **`TRUE`** gesetzt sein.

Felder des **`Script`**-Knotens:

`url`	**`MFString`**	eigentliches Skript (inline oder als url)
`directOutput`	**`SFBool`**	ermöglicht direkte Interaktion mit Knoten
`mustEvaluate`	**`SFBool`**	steuert das Laufzeitverhalten
beliebige Anzahl von:		
Feldname	Feldtyp (beliebig)	+ Angabe von Defaultwert
Ereignisname	**`eventIn`**	
Ereignisname	**`eventOut`**	

Beispiele verschiedener Typen von Codereferenzen im **url**-Feld:

```
Script
{
url [
    "javascript: .... "  # JavaScript-Protokoll, inline
    "file://test.js"     # JavaScript-Protokoll aus Datei
    "file://test.class" ]  # Java Bytecode aus Datei
....
}
```

Das Scripting ermöglicht sehr vielfältige Interaktionsformen; hier nur ein sehr einfaches Beispiel:

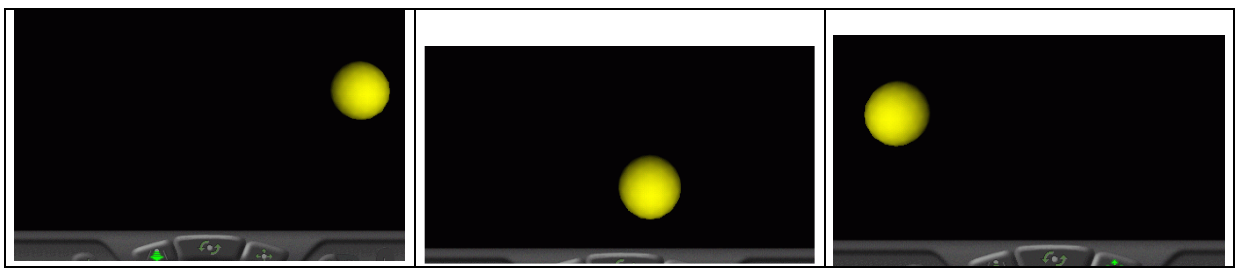
Eine Kugel soll sich auf einer Ellipse bewegen. Im Script-Knoten werden die Koordinaten berechnet unter Rückgriff auf trigonometrische Funktionen in JavaScript:

```
#VRML V2.0 utf8

DEF Timer TimeSensor
    {
    cycleInterval 5
    loop TRUE
    }
DEF Verschieb Transform
    {
    children
       [
       Shape
          {
          geometry Sphere { radius 0.2 }
          appearance Appearance
             {
             material Material
                { diffuseColor 1 1 0 }
             }
```

```
            }
        ]
    }
DEF Berechnung Script
    {
    eventIn SFFloat set_fraction
    eventOut SFVec3f value_changed
    url "javascript:
        // Funktionsberechnungen elliptische Bahn
        function set_fraction (wert, zeit)
            {
            value_changed[0] =
               Math.sin(wert * 6.283);
            value_changed[1] =
               0.5 * Math.cos(wert * 6.283);
            value_changed[2] = 5;
            }
        "
    }

ROUTE Timer.fraction_changed TO
    Berechnung.set_fraction
ROUTE Berechnung.value_changed
    TO Verschieb.set_translation
```

Der Name der JavaScript-Funktion entspricht demjenigen des zu verarbeitenden **`eventIn`**-Feldes (hier: **`set_fraction`**). An die Funktion wird immer ein Wert (der Wert des ankommenden Ereignisses) und eine Zeitmarke übergeben (die nicht notwendig verwendet werden müssen).

Ergebnis des Beispiels (drei snapshots):



Beispiel für die Anwendung des Script-Knotens zur Typumwandlung (hier: von SFFloat in SFInt32), zugleich Beispiel für einen Switch-Knoten:
Ein Objekt verwandelt sich fortlaufend von einem Würfel in eine Kugel und umgekehrt.

```
#VRML V2.0 utf8

DEF Auswahl Switch
    {
    choice
        [
        Shape
            { geometry Box { } }
        Shape
            { geometry Sphere { } }
        ]
    }
DEF Timer TimeSensor
    {
    cycleInterval 5 # Sekunden
    loop TRUE
    }
DEF Berechnung Script
    {
    eventIn SFFloat set_fraction
    eventOut SFInt32 value_changed
    url "javascript:
        // Typkonvertierung
        function set_fraction (wert, zeit)
            {
            value_changed = wert+0.5;
            }
        "
    }
ROUTE Timer.fraction_changed TO
    Berechnung.set_fraction
ROUTE Berechnung.value_changed TO
    Auswahl.set_whichChoice
```

Ergebnis (2 snapshots):