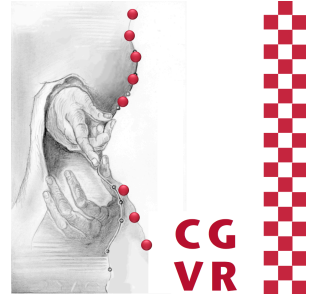


Bremen



Virtual Reality

The Fields & Routes Concept

by way of X3D / VRML



G. Zachmann

University of Bremen, Germany

cgvr.cs.uni-bremen.de

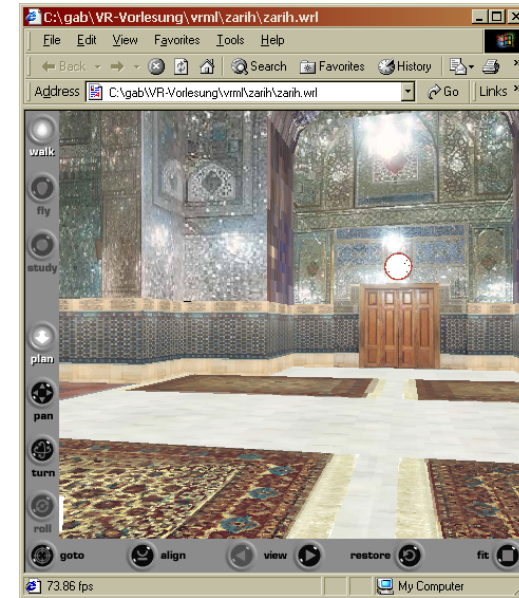
- Was ist X3D/VRML?
 - Scenegraph & File-Format, plus ...
 - Multimedia-Support
 - Hyperlinks
 - Verhalten und Animationen
- Achtung: VRML \neq VR !
- Varianten:
 - VRML 1.0 (1995) (= Inventor, also kein VRML)
 - VRML 2.0 (1996)
 - VRML97 (1997) – ISO Standard, praktisch identisch zu VRML2
 - X3D (2003): ISO Standard, im wesentlichen andere Syntax, nämlich XML

- Die Spezifikation von VRML ist an einigen Stellen nicht eindeutig
 - In X3D präzisiert
- X3D hat 100+ Knoten (aufgeteilt in Components / Profiles)
 - VRML hat nur 54 Knoten
- X3D hat 3 verschiedene sog. "File Encodings":
 - **Classic**: sieht aus wie VRML; Suffix = **.wrl** oder **.x3dv**
 - Jede Software, die X3D lesen kann, kann (im Prinzip) auch VRML lesen
 - **XML**; Suffix = **.x3d**
 - das ist das Format, das man i.A. unter "X3D" versteht
 - **Binary** (XML braucht sehr viel Platz); Suffix = **.x3db**
 - Trick, falls man einen binären VRML-File editieren möchte: einfach xxx.wrl umbenennen nach xxx.wrl.gz, dann mit 'gunzip xxx.wrl.gz' auspacken ;-)

- InstantReality (www.instantreality.org):
 - Läuft auf allen 3 Plattformen
 - Implementiert (angeblich) V3.1 von X3D komplett
- FreeWRL (freewrl.sourceforge.net):
 - Läuft auf Linux, Mac OS X, Android, Windows
 - Implementiert das Subset (Profile) "Interchange" von X3D
- Castle Game Engine (<http://castle-engine.sourceforge.net/>):
 - Alle 3 Plattformen & Android (beta)
 - Lädt auch Collada, OBJ, 3DS, ...
- Octaga (www.octaga.com):
 - Windows & Mac OS X
- BS Contact (www.bitmanagement.com):
 - Alle 5(!) Plattformen



VRMLCity



Zarih

- Bücher:
 - Don Brutzman, Leonard Daly:
X3D: Extensible 3D Graphics or Web Authors. Morgan Kaufman, 2007.
 - Andrea L. Ames, David R. Nadeau, and John L. Moreland:
The VRML 2.0 Sourcebook. John Wiley & Sons, 1996.
 - Hartman, Jed, and Wernecke:
The VRML 2.0 Handbook. Addison-Wesley, 1996.
- Online: Auf der Homepage zur Vorlesung
 - The Annotated VRML97 Reference
 - Der X3D-Standard: Knoten, Javascript, Java
- Die online Doku zu InstantReality:
 - Tutorials
 - Übersicht aller Knoten

- Die Web-Seite zum X3D-Buch:

www.x3dgraphics.com

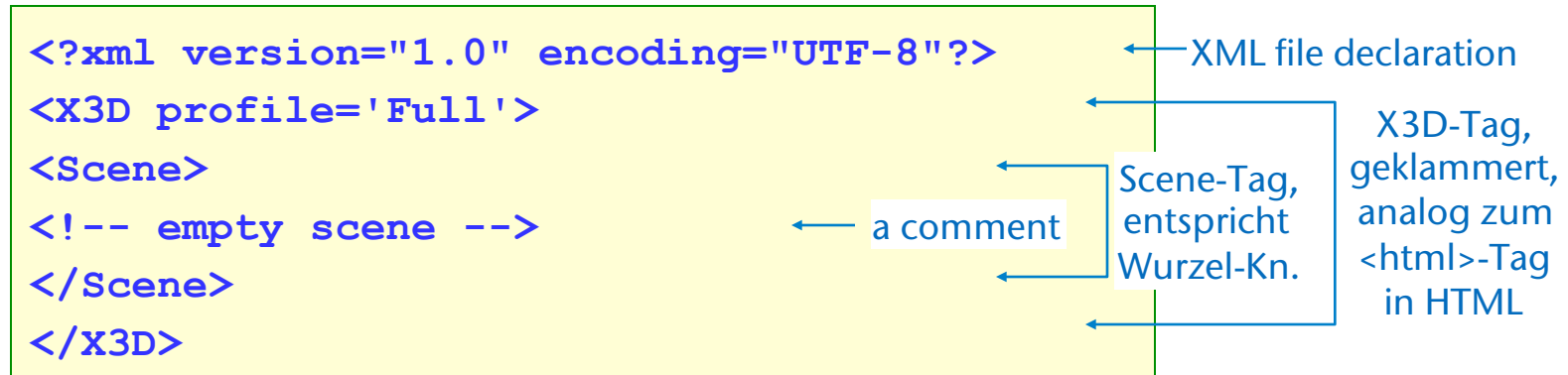
mit Beispielen, Tools, ...

- Eine "Meta"-Seite beim Web3D-Konsortium:

www.web3d.org/x3d/content/examples/X3dResources.html

mit Links zu Software für Viewer, Konverter, Authoring-Tools, Plugins, Beispielen, Büchern, etc.

- Als X3D-Encoding:



- Als ClassicVRML-Encoding:

```
#X3D V3.1 utf8
PROFILE Full
# empty scene
```

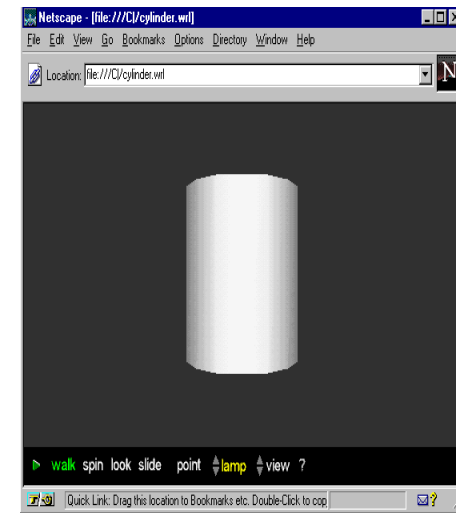
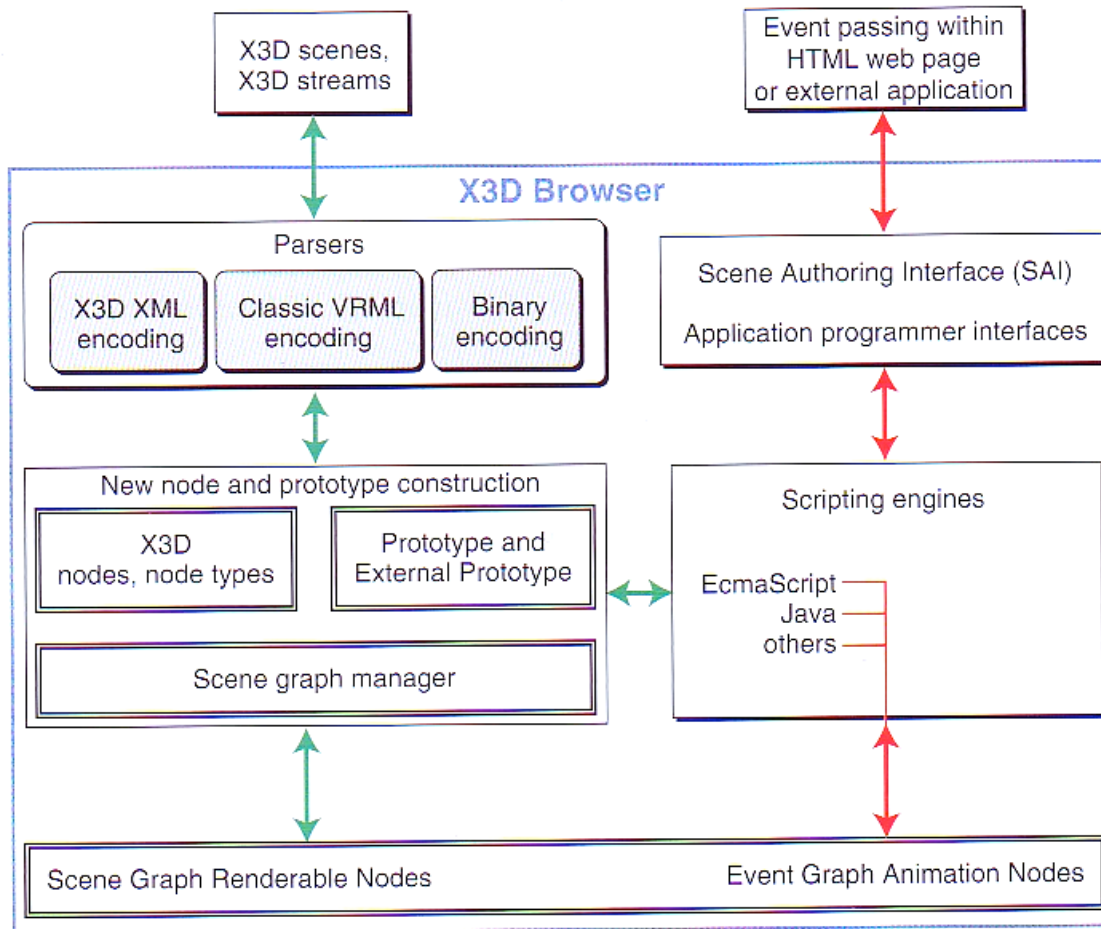
- und als VRML97:

```
#VRML V2.0 utf8
# empty scene
```

- Der Wurzel-Knoten für die gesamte Szene ist in VRML implizit!
- Keine Profiles und viel weniger Knoten in VRML97

- Gründe für das XML-Encoding:
 - Ähnlichkeit zu HTML (Tags und Attribute: `<tag attr="val">...</tag>`)
 - XML ist ASCII (wie VRML97), also im Prinzip "human readable" (im Gegensatz zu binären Formaten)
 - XML ist ein weit verbreiteter Standard zur Beschreibung von Daten
 - XML ist eine Familie von Technologien: CSS, XSLT, Xpointer, ...
 - XML ist Lizenz-frei
- Gründe für das ClassicVRML-Encoding:
 - Legacy-Daten ("Altlasten")
 - Für Menschen leichter zu lesen und zu schreiben

- Der X3D-"Browser" (stand-alone oder embedded):



- In X3D (genauer: XML-Encoding):

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D profile='Immersive'>
<Scene>
  <Shape>
    <Text string="Hello" "world!" />
  </Shape>
</Scene>
</X3D>
```



- In VRML:

```
#X3D V3.1 utf8
Shape {
  geometry Text {
    string [ "Hello" "world!" ]
  }
}
```

Tip: ASCII-Editor verwenden, der *matching brackets* erkennt und als Texteinheit behandeln kann

- Knoten dienen zur Beschreibung ...
 - ... des **Szenengraphen** (die üblichen Verdächtigen):
 - Geometry, Transform, Group, Lights, LODs, ...
 - ... des **Verhaltensgraphen (behavior graph)**, d.h., des Verhaltens der Objekte und bei User-Input
- Knoten := Menge von Feldern
 - "Single-valued Fields" und "Multiple-valued Fields"
 - Jedes Feld eines Knotens hat einen eindeutigen Namen
 - Diese Namen sind per Spezifikation vordefiniert
- Feldarten:
 - **field** = Daten im File
 - **eventIn, eventOut** = s.u., werden nicht gespeichert
 - **exposedField** = Kombination der drei (xxx, set_xxx, xxx_changed)

- Alle Feldtypen gibt es als "single valued"- (**SF**...) und als "multiple valued"-Variante (**MF**...)
- Beispiel für ein SFField:

```
<Material diffuseColor="0.1 0.5 1" />
```

X3D

```
material Material {  
    diffuseColor 0.1 0.5 1  
}
```

VRML

- MFField's sind im Prinzip nichts anderes als **Arrays**
 - Falls der **Grundtyp** ein **Tuple** ist (z.B. Farbe oder Vektor), sollte man in X3D die einzelnen Elemente mit Komma trennen. Beispiel:

```
"1 0 0, 0 1 0, 0 0 1"
```

- In VRML müssen MFField's mit [] geschrieben werden. Beispiel:

```
[ 1 0 0, 0 1 0, 0 0 1 ]
```

- Grundtypen: die üblichen Verdächtigen:

Field type	X3D example	VRML example
SFBool	true / false	TRUE / FALSE
SFInt32	12	-17
SFFloat	1.2	-1.7
SFDouble	3.1415926535	
SFString	"hello"	"world"

Erinnerung:
zu jedem
SF-Feld
gibt es ein
MF-Feld

- Etwas höhere Datentypen:

Field type	Beispiel
SFColor	0 0.5 1.0
SFColorRGBA	0 0.5 1.0 0.75
SFVec3f	1.2 3.4 5.6
SFMatrix3f	1 0 0 0 1 0 0 0 1
SFString	"hello"

Anmerkungen:

- Die Werte in **SFColor** müssen in $[0,1]$ liegen
- Analog gibt es die Varianten ***2f**, ***3f** und ***4f**.

- Spezielle Feld-Typen:

Field type	X3D example	VRML example
SFImage	enthält spezielle Pixel-Encodings	
SFNode	<code><Shape> ... </Shape></code>	<code>Shape { ... }</code>
MFNode	<code><Shape>... , <Group>...</code> oder <code><Transform>...</code>	<code>Transform {</code> <code>children [...] }</code>
SFRotation	<code>0 1 0 3.1415</code>	
SFTime	<code>0</code>	

- Anmerkungen zu **SFImage**:

- Der Wert des Feldes ist eine Folge von Zahlen: Breite, Höhe, Anzahl Komponenten pro Pixel, Pixel, Pixel, ...
- Pro Kanal Werte im Bereich [0,255]
- Beispiel (bei 3 Komponenten): **0xFF0000** = Rot, **0x00FF00** = Grün, ...
- Beispiel für ein vollständiges **SFImage**:

```
2 4 3 0xFF0000 0xFF00 0 0 0 0 0xFFFFFFFF 0xFFFF00  
# w·h·c    red    green black..  white  yellow
```

- **SFImage** ist nur für sehr kleine Texturen gedacht und kommt nur im Knoten **PixelTexture** vor
 - Hintergrund: man wollte eine Möglichkeit haben, Texturen algorithmisch zu erzeugen (mittels Java)
- Für große ("richtige") Texturen verwende man PNGs oder JPGs und den Knoten **ImageTexture**

- Generelle Anmerkungen zum Design:
 - Das Design ist insofern **orthogonal**, als es zu jedem **SF**-Typ einen **MF**-Typ gibt
 - Das Design ist insofern **nicht orthogonal**, als manche Typen generisch sind (z.B. **SFBool**, **SFVec3f**), andere wiederum eine festgelegte Semantik haben (z.B. **SFColor**, **SFTime**, etc.)
 - Es ist nicht ganz klar, ob dies gut/schlecht ist ...

Die Spezifikation der Knoten

- Knoten werden definiert durch ihre Felder und deren Bedeutung
- Die Syntax zur Definition von Knoten (vorerst):

```
Name_of_Node_Class {  
    type_of_field name_of_field_1 default_value  
    type_of_field name_of_field_2 default_value  
    ...  
}
```

- Bemerkungen:
 - Die Defaults werde ich im Folgenden meist weglassen
 - Auch werde ich nicht alle Felder aufzählen, nur die wichtigsten
 - Im folgenden werden nur einige wenige Knoten besprochen
- Fazit: schauen Sie in die Doku und das Tutorial!

- Beispiel:

```
Cone {  
    SFFloat  bottomRadius 1  
    SFFloat  height      2  
    SFBool   side        TRUE  
    SFBool   bottom      TRUE  
}
```

- Verwendung:

```
Cone { bottomRadius 1 height 2 }
```

VRML-Syntax

```
<Cone bottomRadius="1" height="2" />
```

XML-Syntax

- Bemerkung: Cone ist in XML-Syntax ein sog. **Singleton-Element**, d.h., es gibt kein öffnendes/schließendes Tag-Paar!

Knoten zur Beschreibung des graph. Szenengraphen

- Alle Geometrie-Knoten müssen Kind eines **Shape**-Knotens sein
- Definition:

```
Shape {  
    SFNode geometry NULL  
    SFNode appearance NULL  
}
```

- Achtung:
 - Das Feld **geometry** darf nur Geometrie-Knoten enthalten (es gibt etliche Klassen von Geometrie-Knoten)
 - Das Feld **appearance** darf nur einen Appearance-Knoten enthalten (es gibt nur eine Klasse von Appearance-Knoten)
- **Shape**-Knoten dienen dazu, Geometrie mit einer **Appearance** zu verknüpfen

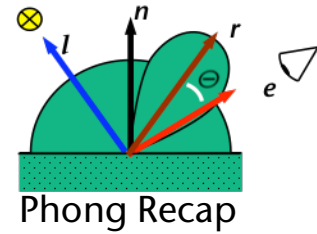
- Der Appearance-Knoten dient zur Spezifikation des Aussehens einer Geometrie
- Definition:

```
Appearance {  
    SFNode  material      NULL  
    SFNode  texture      NULL  
    SFNode  fillProperties NULL  
    ...  
}
```

- Auch hier gilt wieder: die Werte eines Feldes (hier: Instanzen einer Knotenklasse) müssen vom "richtigen" Typ (d.h., der richtigen Klasse) sein

- Der Material-Knoten:

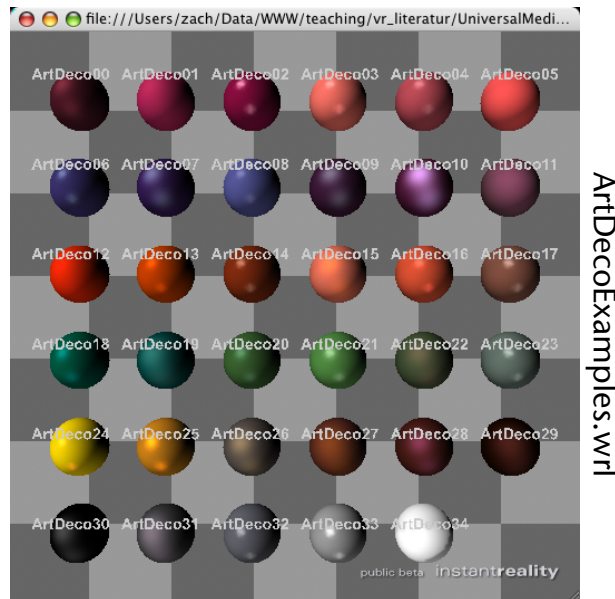
```
Material {  
  SFFloat ambientIntensity 0.2  
  SFColor  diffuseColor      0.8 0.8 0.8  
  SFColor  emissiveColor     0 0 0  
  SFColor  specularColor     0 0 0  
  SFFloat  shininess         0.2  
  SFFloat  transparency      0  
}
```



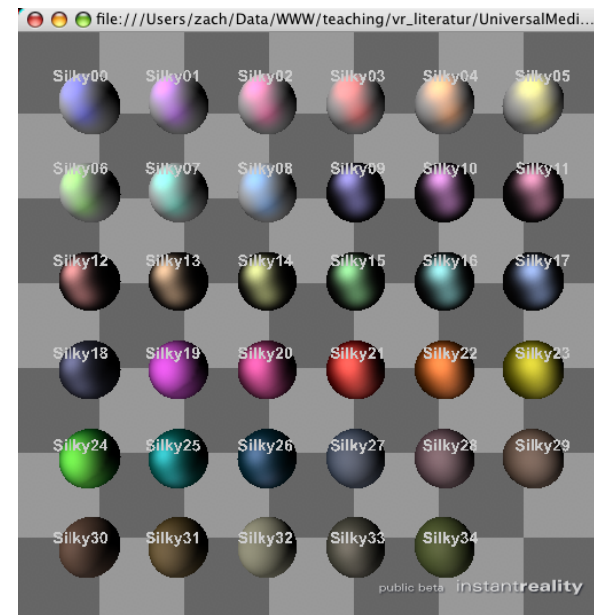
- Der Textur-Knoten:

```
ImageTexture {  
  MFString url      [ ]  
  SFBool  repeatsS  TRUE  
  SFBool  repeatT   TRUE  
}
```

- Auf der Homepage der Vorlesung finden Sie unter "*Online Literatur und Resources im Internet*" ein großes Archiv mit Materialien
- Beispiele:



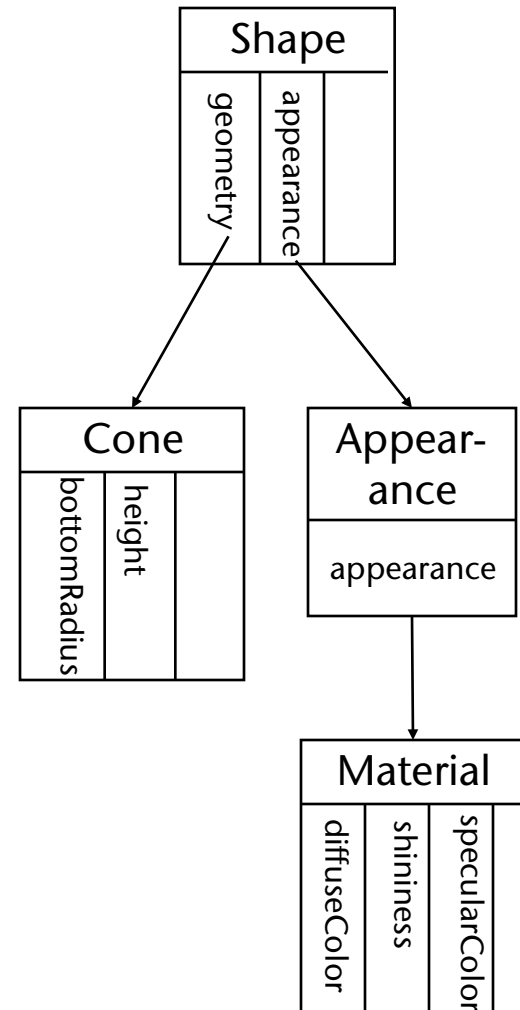
ArtDecoExamples.wrl



SilkyExamples.wrl

- Weitere Resource: ein Material-Editor (in Java)
<http://tog.acm.org/resources/applets/vrml/pellucid.html>

```
#X3D V3.1 utf8
Shape {
  geometry Cone {
    bottomRadius 1
    height      2
  }
  appearance Appearance {
    material Material {
      ambientIntensity 0.256
      diffuseColor      0.029 0.026 0.027
      shininess         0.061
      specularColor     0.964 0.642 0.980
    }
  }
}
```



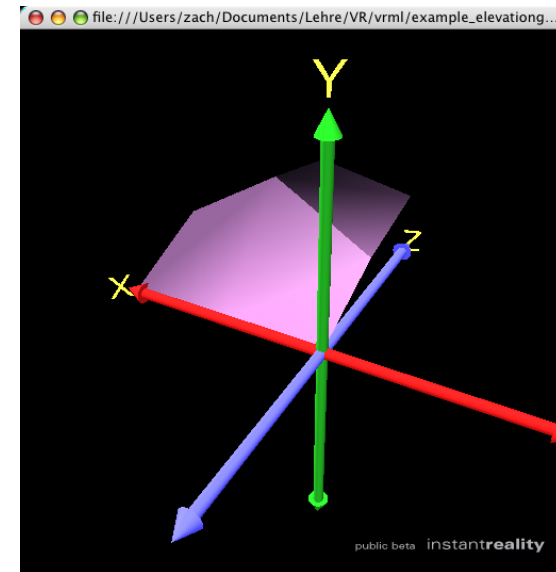
- Allgemein für (diskrete) Flächen, die sich als Funktion über einer Ebene beschreiben lassen
- Definition:

```
ElevationGrid {  
    SFFloat normalPerVertex    TRUE  
    SFFloat creaseAngle        0.0  
    MFFloat height             []  
    SFInt32  xDimension         0  
    SFFloat  xSpacing           1.0  
    SFInt32  zDimension         0  
    SFFloat  zSpacing           1.0  
    SFFloat  solid              TRUE  
}
```

- Bedeutung der Felder:
 - **normalPerVertex** schaltet Beleuchtung pro Vertex mit Gouraud-Shading ein (die Normalen werden i.A. vom Browser berechnet)
 - **solid = TRUE** schaltet Backface-Culling ein
 - Tip: bei Terrain ausschalten
 - Alle Winkel zwischen 2 Polygonen über eine Kante hinweg (*dihedral angle*), die größer als **creaseAngle** sind, werden erhalten, d.h., die beteiligten Vertices werden für die beiden Polygone mit jew. einer eigenen Normale gerendert
- Anmerkungen:
 - Aus Matlab kann man Plots als ein solches VRML-ElevationGrid exportieren
 - Achtung: die Vierecke sind i.A. nicht planar → Flackern und andere Artefakte!

■ Beispiel:

```
Shape {  
  appearance Appearance { ... }  
  geometry ElevationGrid {  
    height [  
      0.0 0.0 0.0  
      0.2 0.5 0.2  
      0.3 0.4 0.1 ]  
    xDimension 3  
    zDimension 3  
    xSpacing 0.5  
    zSpacing 0.5  
    solid false  
    #creaseAngle 1.5  
  }  
}
```



example_elevationgrid.wrl

- Die allgemeinste Geometrie
- Für Dreiecke (und Vierecke) gibt es viele Varianten; hier nur 2
- Die einfachste Variante: **TriangleSet**
- Definition:

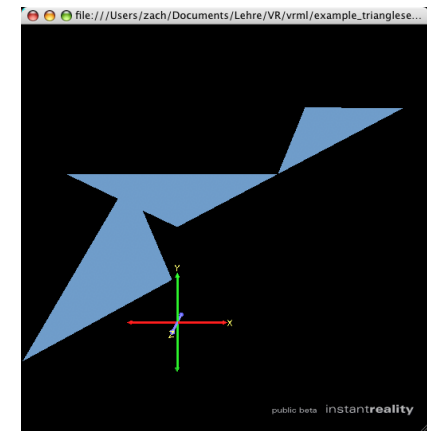
```
TriangleSet {
  SFNode   coord           NULL
  SFBool   ccw             TRUE
  SFBool   normalPerVertex TRUE
  SFBool   solid           TRUE
  SFFloat  creaseAngle     0.0
}
```

```
Coordinate {
  MFVec3f  point []
}
```

- **coord** → **point** ist eine Liste von Koordinaten; je 3 aufeinanderfolgende ergeben einen Vertex; davon je 3 aufeinanderfolgende ergeben ein Dreieck
- **ccw** (*counter-clockwise*) gibt an, ob die Vertices im Uhrzeigersinn vorliegen oder nicht

- Beispiel:

```
Shape {  
  appearance Appearance { ... }  
  geometry TriangleSet {  
    coord Coordinate {  
      point [ -2 0 3, -0 1 1, -1 3 0,  
              0 2 0, 2 3 1, -2 3 1,  
              3 5 -2, 2 3 1, 4 4 2 ]  
    }  
    solid FALSE  
    ccw TRUE  
  }  
}
```



[example_triangleset.wrl](#)

- Bemerkung:

- das Komma ist in X3D/VRML ein *Whitespace*
- könnte man also weglassen; sollte man bei hand-geschriebenen Szenen aber nicht

- Ein häufig vorkommender Knoten ist **IndexedTriangleSet**:

```

IndexedTriangleSet {
  SFNode   coord      NULL
  MFInt32  index      []
  SFBool   ccw        TRUE
  SFBool   normalPerVertex TRUE
  SFBool   solid      TRUE
  SFFloat  creaseAngle 0.0
}

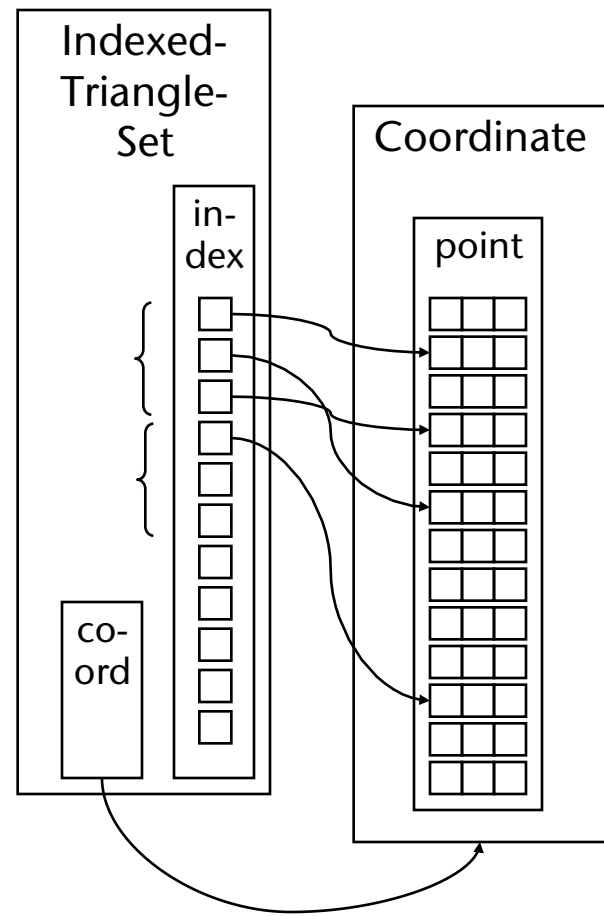
```

```

Coordinate {
  MFVec3f point []
}

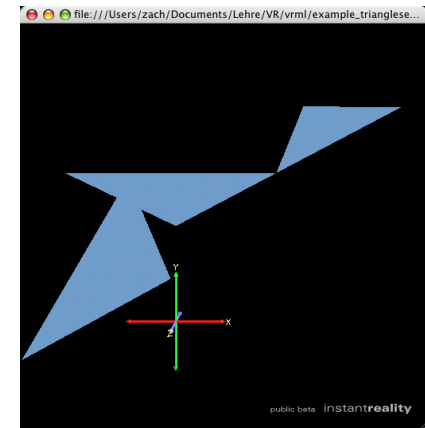
```

- Großer Vorteil: Speicher-Einsparung
- Denn: bei "normalen" Dreiecks-Meshes wird jeder Vertex im Schnitt von 6 Dreiecken "benutzt"



- Dasselbe Beispiel nochmal, diesmal mit **IndexedTriangleSet**:

```
Shape {  
  appearance Appearance { ... }  
  geometry IndexedTriangleSet {  
    coord Coordinate {  
      point [ -2 0 3, -0 1 1, -1 3 0,  
              0 2 0,  2 3 1, -2 3 1,  
              3 5 -2, 4 4 2 ]  
    }  
    index [ 0 1 2,  3 4 5,  6 4 7 ]  
    solid FALSE  
    ccw TRUE  
  }  
}
```



[example_indexedtrianglese.wrl](#)

- Der häufigste Knoten-Typ ist (unnötigerweise) das **IndexedFaceSet**:

```

IndexedFaceSet {
  SFNode   coord      NULL
  MFInt32  coordIndex []
  SFBool   ccw        TRUE
  SFBool   normalPerVertex TRUE
  SFBool   solid      TRUE
  SFFloat  creaseAngle 0.0
}

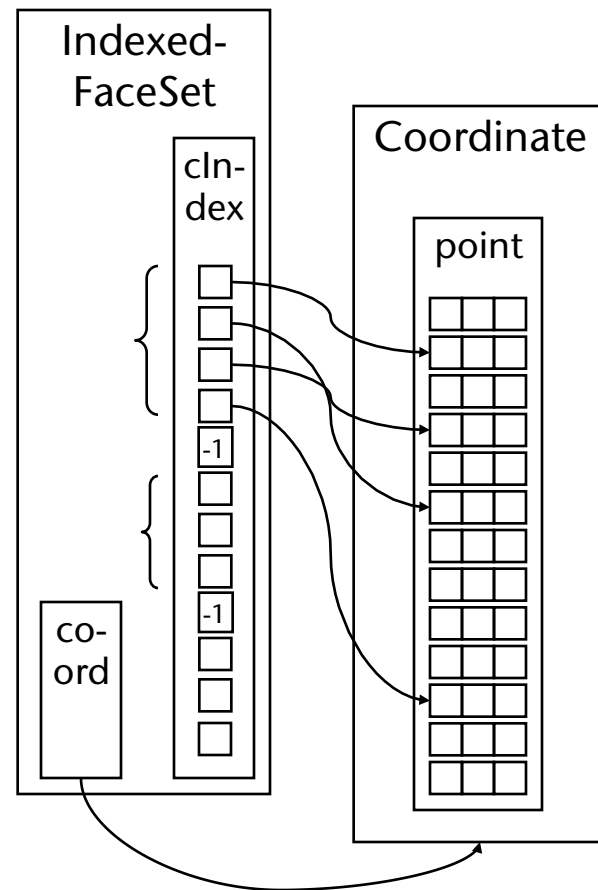
```

```

Coordinate {
  MFVec3f point []
}

```

- Unterschied zu **IndexedTriangleSet**: die -1 als "*Sentinel*"



- Vorteil: beliebige Polygone
- Anmerkung: viele Exporter exportieren **IndexedFaceSet** obwohl alle Pgone Dreiecke sind → Speicherverschwendung & langsameres Rendering!
- Das Beispiel von vorhin nochmals als **IndexedFaceSet**:

```
Shape {
  appearance Appearance { ... }
  geometry IndexedFaceSet {
    coord Coordinate {
      point [ -2 0 3, -0 1 1, -1 3 0,
              0 2 0,  2 3 1, -2 3 1,
              3 5 -2, 4 4 2 ]
    }
    coordIndex [ 0 1 2 -1  3 4 5 -1  6 4 7 -1 ]
    solid FALSE
    ccw TRUE
  }
}
```

Spezifikation weiterer Attribute pro Vertex

- In allen Geometrie-Knoten kann man weitere Vertex-Attribute spezifizieren, z.B., Normalen oder Texturkoord. pro Vertex
- Hier am Beispiel Texturkoord. im **IndexedFaceSet**:

```

IndexedFaceSet {
  SFNode coord
  MFInt32 coordIndex
  SFNode texCoord
  MFInt32 texCoordIndex
  SFBool ccw
  SFBool normalPerVertex
  SFBool solid
}

```

```

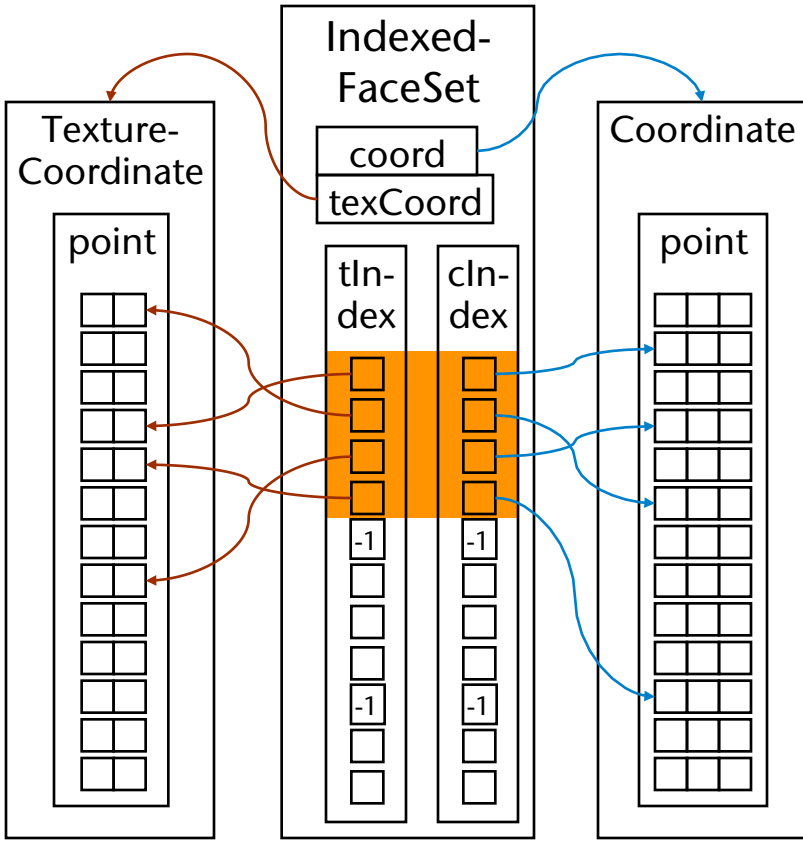
TextureCoordinate {
  MFVec2f point []
}

```

```

Coordinate {
  MFVec3f point []
}

```



- Es gibt noch viele weitere:

PointSet, LineSet, QuadSet, ...

IndexedLineSet, IndexedQuadSet, ...

TriangleStripSet, IndexedTriangleStripSet, ...

Box, Sphere, Cylinder, ...

Text, Extrusion, ...

- Viele 2D-Knoten, z.B.: **Arc2D, Polyline2D, ...**

- CAD-Knoten: **CADAssembly, NurbsPatchSurface, ...**

- Einfache Gruppen-Knoten:

```
Group {  
    MFNode children []  
}
```

- Die Knoten im Feld **children** dürfen wieder **Group**-Knoten sein oder **Shape**-Knoten
- Beispiel: ...

- Transformationen:

```

Transform {
  MFNode      children      []
  SFVec3f     center        0 0 0
  SFRotation  scaleOrientation 0 0 1 0
  SFVec3f     scale         1 1 1
  SFRotation  rotation      0 0 1 0
  SFVec3f     translation   0 0 0
}

```

C
*R*₁
S
*R*₂
T

- Alle Kinder unter einem **Transform**-Knoten werden transformiert
 - Oft hat ein **Transform**-Knoten nur 1 Kind
- Bedeutung:

$$p' = T \cdot C \cdot R_2 \cdot R_1 \cdot S \cdot R_1^{-1} \cdot C^{-1} \cdot p$$

- scaleOrientation erlaubt also eine Skalierung entlang beliebiger (lokaler) Achsen, nicht nur entlang der lokalen Koord.achsen

- Ein "include"-Mechanismus mittels des **Inline**-Knotens:

```
Inline {  
    SFBool    load    TRUE  
    MFString url      []  
}
```

- Mit **load=FALSE** kann man das Laden der Teil-Szene aufschieben; bei **TRUE** wird die Teil-Szene beim Parsen der Parent-Szene geladen
 - Die erste gefundene URL im Feld **url** wird genommen
- Beispiel:

```
Transform {  
    scale 0.5  
    children [  
        Inline {  
            url [ "coordAxes.wrl"  
                "http://my.site.com/coordAxes.wrl" ]  
        }  
    ]  
}
```

- Mit dem **Switch**-Knoten kann man eines aus mehreren Kindern einschalten
- Definition:

```
Switch {  
    SFInt32 whichChoice -1  
    MFNode  children   []  
}
```

- **whichChoice=-1** schaltet alle Kinder ab, **whichChoice=0** schaltet das erste Kind an

- Mit diesem Knoten kann man Hinweistafeln u.ä. erstellen:

```
Billboard {  
    SFVec3f axisOfRotation 0 1 0  
    MFNode  children      []  
}
```

- Dieser Knoten erzeugt in jedem Frame eine Transformation, die dafür sorgt, daß die lokale z-Achse zum aktuellen Viewpoint zeigt
- **axisOfRotation** wird im lokalen Koordinatensystem spezifiziert
- Falls **axisOfRotation** = (0,0,0) ist, dann wird zusätzlich die lokale y-Achse parallel zur y-Achse des Viewers ausgerichtet

Wiederverwendung von Szenengraphenteilen

- Beispiele, wo Wiederverwendung Sinn macht:
 - Ein Teil der Geometrie kommt mehrfach in der Szene vor (i.A. an verschiedenen Positionen)
 - Dieselbe Appearance (Material / Textur) soll auf verschiedene Geometrien angewendet werden

- Mechanismus in X3D/VRML:

- Namen für einen Knoten definieren:

```
DEF nodeName NodeType {  
    fields ...  
}
```

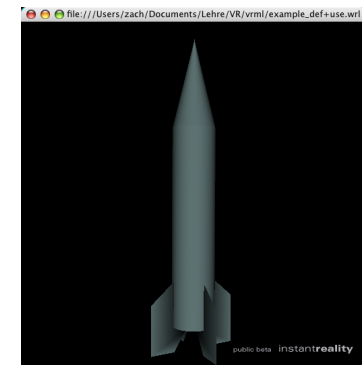
- An jeder Stelle, wo ein Knoten des entsprechenden Typs stehen kann, kann nun einfach

```
USE nodeName
```

verwendet werden

■ Beispiel:

```
example_def+use.wrl (~/.Documents/Lehre/VR/vrml) - VIM
Shape {
  geometry Cylinder {
    height 4
    radius 0.4425
    top FALSE
  }
  appearance DEF Cammi Appearance {
    material Material {
      diffuseColor 0.45 0.55 0.55
    }
  }
}
Transform {
  translation 0 2.9 0
  children [
    Shape {
      geometry Cone {
        bottomRadius 0.4425
        height 1.8
      }
      appearance USE Cammi
    }
  ]
}
DEF TailFin Transform {
  translation 0.175 -2.5 0
  children [
    Shape {
      geometry IndexedFaceSet {
        coordIndex [ 0 1 2 3 4 5 -1 ]
        solid FALSE
        coord Coordinate {
          point [ 0 0.4 0 0.25 0 0 0.75 0 0 0.75 1 0 0 1.65 0 0 0.4 0 ]
        }
      }
      appearance USE Cammi
    }
  ]
}
Transform {
  rotation 0 1 0 1.57
  children [
    USE TailFin
  ]
}
Transform {
  rotation 0 1 0 3.14
  children [
    "example_def+use.wrl" [converted] 58L, 931C written
  ]
}
```



vrml/examples/
example_def+use.wrl

- Die Bezeichnung **DEF** ist sehr unglücklich
- Wahre Semantik / Eselsbrücke: **DEF** \approx "Name", **USE** \approx Pointer!
- Scope: reicht vom **DEF** bis zum Ende des Files — Klammern (**{ }** **[]**) spielen keine Rolle!
- **DEF** muß im File **vor USE** kommen (logisch), aber nicht notwendigerweise auf demselben Level im Szenengraph
 - Dadurch könnte man sogar Zyklen im Graph erzeugen!
- Tip: sinnvolle Namen vergeben

- "Animationen" (i.e., dynamische Szenengraphen) sind Veränderungen des Szenengraphen; z.B.:
 - Änderungen von Transformationen, z.B. die Position von Objekten oder die Bewegung eines Roboterarmes,
 - Änderungen des Materials, z.B. der Farbe oder der Texturkoord. eines Objektes,
 - Deformation eines Objektes, d.h., Änderungen der Vertex-Koord.,
- Alle diese Veränderungen sind äquivalent zur **Änderung eines Feldes eines Knotens zur Laufzeit**

- Der Mechanismus in X3D zur Veränderung des Szenengraphen:
 - Felder können miteinander durch sog. **Routes** verbunden werden
 - Es gibt spezielle Knoten, deren Felder sich ändern
 - Eine *Änderung* eines Feldes erzeugt einen sog. **Event**
 - Bei Auftreten eines Events wird der *Inhalt* des Feldes vom Route-Anfang zum Feld des Route-Endes *kopiert* ("der Event wird propagiert")
- Andere Bezeichnungen: *data flow paradigm*

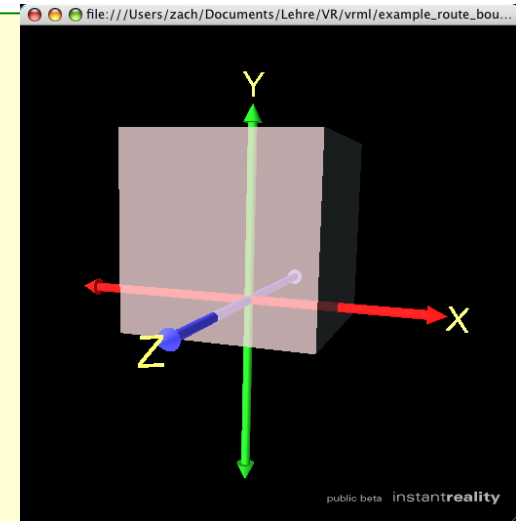
- Felder haben nicht nur Typ und Wert, sondern auch einen sog. *"access type"*:

VRML97	X3D
<code>eventIn</code>	<code>inputOnly</code>
<code>eventOut</code>	<code>outputOnly</code>
<code>field</code>	<code>initializeOnly</code>
<code>exposedField</code>	<code>inputOutput</code>

- Felder mit einem Namen **zzz**, die den Access-Type `exposedField` haben, haben implizit den Namen **zzz_changed**, wenn sie als Ausgabe-Feld verwendet werden, und den Namen **set_zzz**, wenn sie als Eingabe-Feld verwendet werden
 - Viele der vordefinierten Felder in `vordef`. Knoten sind **exposedField's**

- Ein einfaches Beispiel:

```
DEF ts TimeSensor {  
  loop TRUE  
  cycleInterval 5  
}  
  
DEF pi PositionInterpolator {  
  key      [ 0      0.5    1      ]  
  keyValue [ 0 -1 0, 0 1 0, 0 -1 0 ]  
}  
  
DEF tr Transform {  
  translation 0 0 0  
  children [  
    Shape { geometry Box { } }  
  ]  
}  
  
ROUTE ts.fraction_changed TO pi.set_fraction  
ROUTE pi.value_changed TO tr.set_translation
```



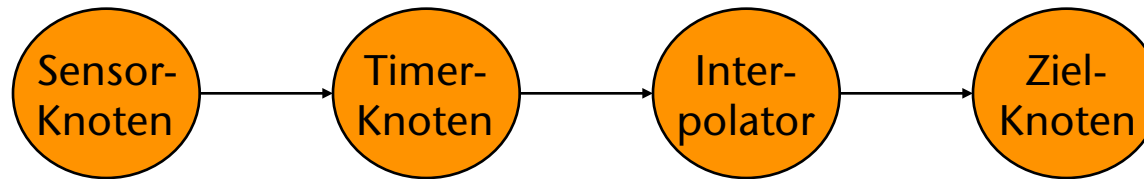
[example_route_bounce.wrl](#)

- Syntax der Routes:

```
ROUTE nodeName.outputFieldName TO nodeName.inputFieldName
```

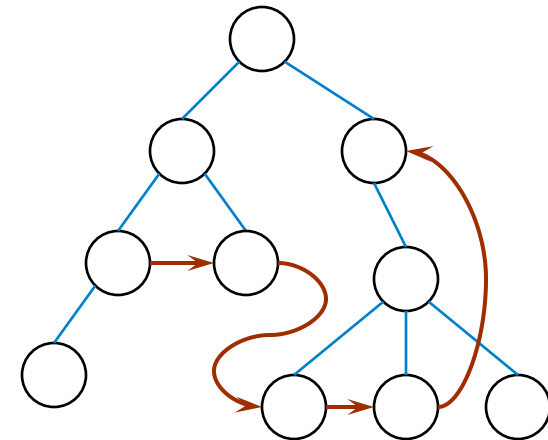
- Die Knoten müssen früher im File mit **DEF** spezifiziert worden sein
- Routes dürfen nur von Ausgabe-Feldern (d.h., **eventOut** oder **exposedField**) zu Eingabe-Feldern (d.h., **eventIn** oder **exposedField**) gezogen werden
- Felder dürfen *fan-in* und *fan-out* haben (mehrere eingehende / ausgehende Routes)
 - Verhalten bei gleichzeitiger Ankunft mehrerer Events ist undefiniert!

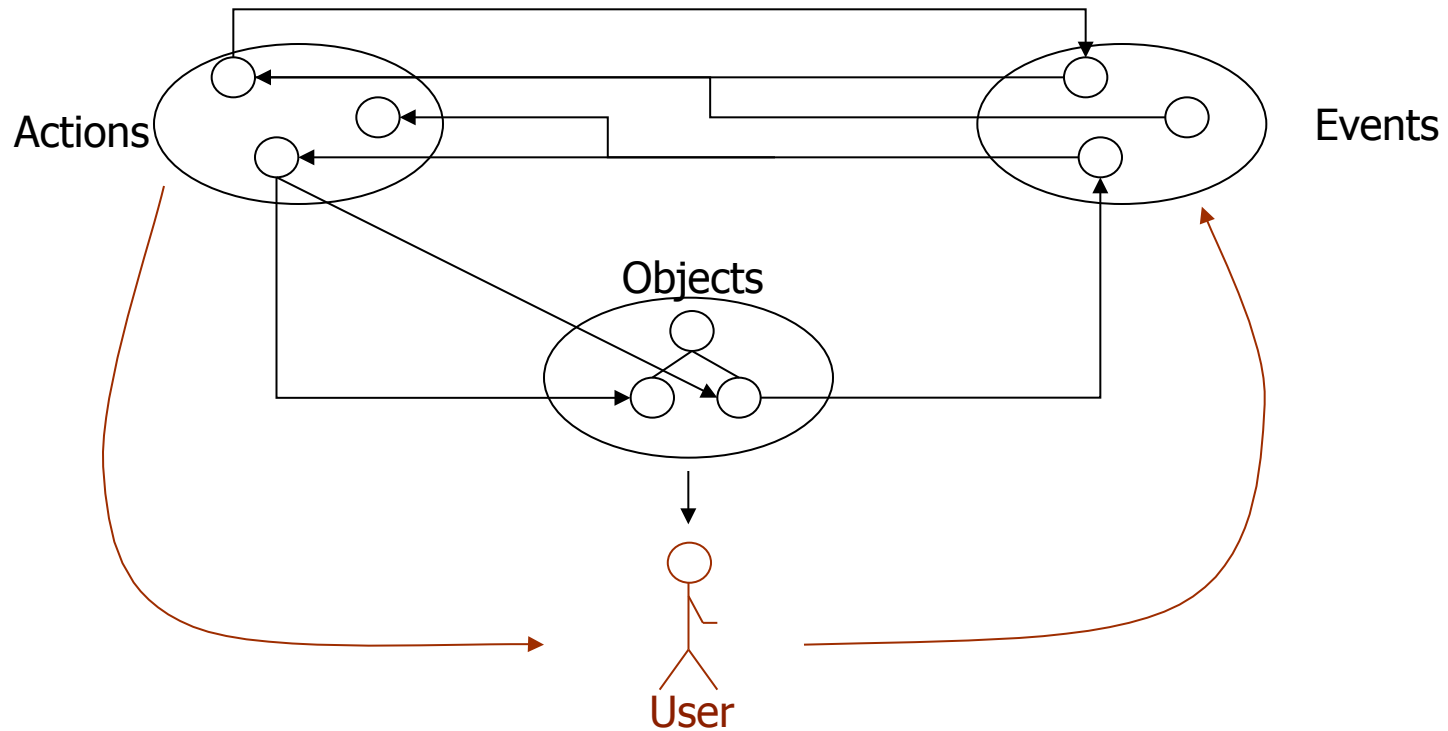
- Eine Folge von Routes verläuft oft nach diesem Schema:



- Der Behavior-Graph:

- Ergibt sich durch die Menge aller Routes
- Heißt auch **Route-Graph**, oder **Event-Graph**
- Ist ein zweiter, dem Szenengraphen **überlagerter** Graph

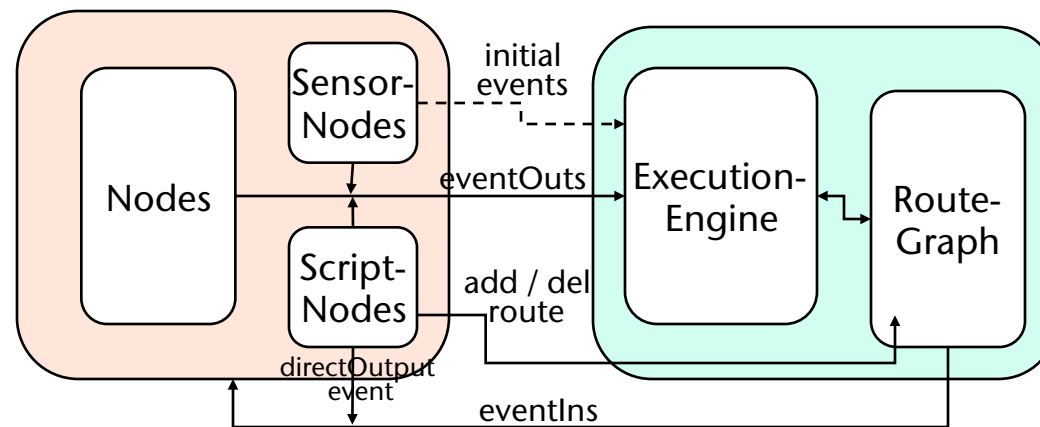




- In X3D/VRML:
 - Actions & objects sind alle Knoten im selben Scenegraph
 - Events sind flüchtige "Ereignisse", haben keine greifbare Repräsentation

■ Die Event Cascade:

- Initialer Event (von Script, Sensor, oder Timer)
- Propagiere an alle angeschlossenen **eventIn**'s
- Knoten (z.B. Interpolator) können als Folge weitere Events generieren über **eventOut**'s
- Alle diese Events sind Teil derselben Kaskade
- Propagiere so lange, bis die Kaskade leer ist



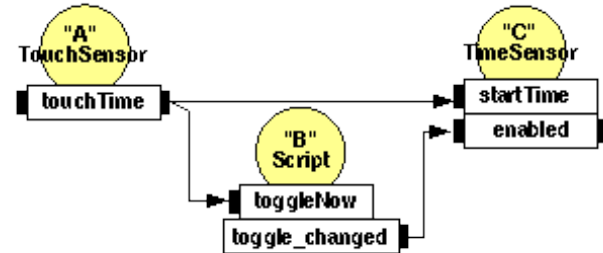
- Pro Frame können mehrere Kaskaden auftreten (durch verschiedene initiale Events ausgelöst)

- Routes induzieren eine Abhängigkeit der Knoten:

- Propagiere in der "richtigen" Reihenfolge

- Algo:

- Breadth-first traversal
 - Sortiere aktuelle Front gemäß Abhängigkeiten



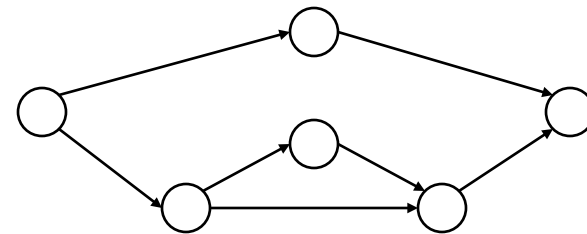
- Zyklen:

- Sind erlaubt (manchmal sogar sinnvoll)

- Loop breaking rule:*

Jedes Feld darf nur 1x pro Event-Kaskade "feuern";

m.A.W.: jede Route wird nur 1x pro Event-Kaskade "bedient"

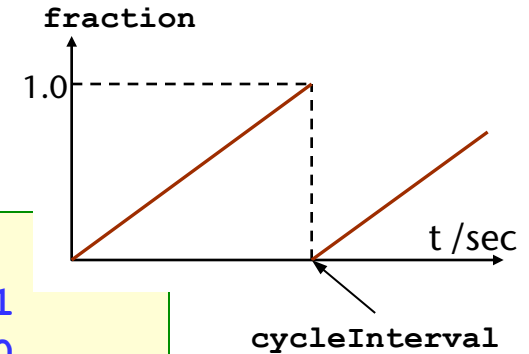


- Der **TimeSensor**-Knoten:

```

TimeSensor {
  exposedField STime  cycleInterval  1
  exposedField STime  startTime      0
  exposedField SFBool loop           false
  eventOut     STime  fraction_changed
  eventOut     STime  time
  eventOut     SFBool isActive
  eventOut     STime  cycleTime
  ...
}

```



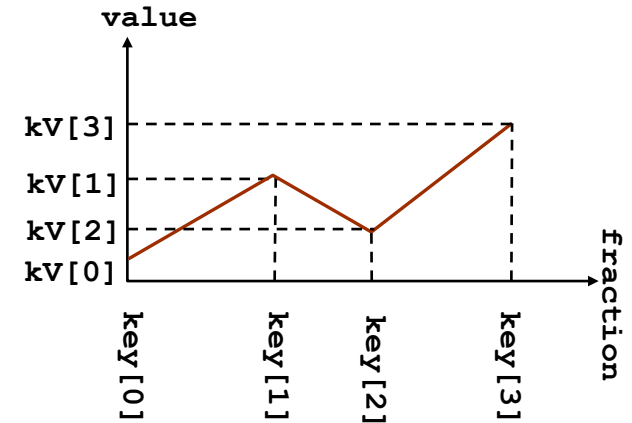
- Der Timer wird aktiv, sobald die System-Zeit > **startTime** wird
- Falls **startTime=-1**, ist der Timer inaktiv
 - Das ist die beste Art, einen Timer zu de-/aktivieren (s. Bsp. drehende_quadrate)
- Mit Hilfe von **isActive** kann man Animationen aneinanderketten
 - **isActive** zeigt an, ob ein Timer gerade läuft
- cycleTime** sendet einen **STime**-Event bei jedem **cycle**-Beginn

- Ein Knoten zur Interpolation von Skalaren:

```

ScalarInterpolator {
  MFFloat exposedField key
  MFFloat exposedField keyValue
  SFFloat eventIn      set_fraction
  SFFloat eventOut     value_changed
}

```



- Weitere Interpolationsknoten:

```

ColorInterpolator {
  MFVec3f expF  keyValue
  SFColor  out  value_changed
}

```

```

CoordinateInterpolator {
  MFVec3f expF  keyValue
  MFVec3f out  value_changed
}

```

```

PositionInterpolator {
  MFVec3f expF  keyValue
  SFVec3f out  value_changed
}

```

```

OrientationInterpolator {
  MFRotation expF  keyValue
  SFRotation out  value_changed
}

```

- Achtung: man sollte darauf achten, daß man den richtigen Interpolator zum "richtigen" Knoten verbindet
 - Es ist z.B. nicht erlaubt, einen **ColorInterpolator** mit der **translation** eines **Transform**-Knotens zu verbinden
- Der **CoordinateInterpolator** ist dazu gedacht, Geometrie zu animieren (also animierte Deformation)
 - Achtung:

$$\text{Anzahl Vec3f's im Feld value_changed} = \frac{\text{Anzahl Vec3f's im Feld keyValues}}{\text{Anzahl key's}}$$

und diese Anzahl muß natürlich mit dem Empfänger-Feld übereinstimmen

```

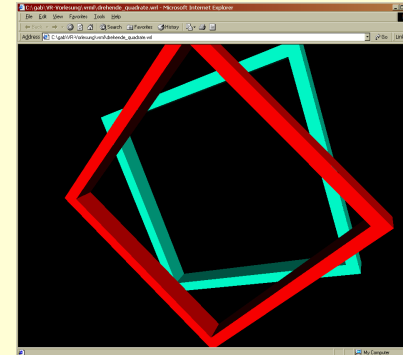
DEF Frame1 Transform {
  translation 0.0 0.0 -0.5
  children [ Shape { ... } ]
}
DEF Frame2 Transform {
  translation 0.0 0.0 +0.5
  children [ Shape { ... } ]
}

DEF Rot1 OrientationInterpolator {
  key      [ 0.0,          0.5,          1.0          ]
  keyValue [ 0.0 0.0 1.0 0.0, 0.0 0.0 1.0 3.14, 0.0 0.0 1.0 6.28 ]
}
DEF Rot2 OrientationInterpolator {
  key      [ 0.0,          0.5,          1.0          ]
  keyValue [ 0.0 0.0 1.0 0.0, 0.0 0.0 1.0 3.14, 0.0 0.0 1.0 6.28 ]
}

DEF Timer1 TimeSensor { cycleInterval 10.0 loop TRUE startTime -1 }
DEF Timer2 TimeSensor { cycleInterval 11.0 loop TRUE startTime -1 }

ROUTE Timer1.fraction_changed TO Rot1.set_fraction
ROUTE Timer2.fraction_changed TO Rot2.set_fraction
ROUTE Rot1.value_changed TO Frame1.set_rotation
ROUTE Rot2.value_changed TO Frame2.set_rotation

```



[drehende_quadrate.wrl](#)


```
wiggle.wrl + (~/.Documents/Lehre/VR/vrml) - VIM
    children USE Wig }
Transform { translation 3.0 0.0 0.0 rotation 1.0 0.0 0.0 3.1415
    children USE Wig }

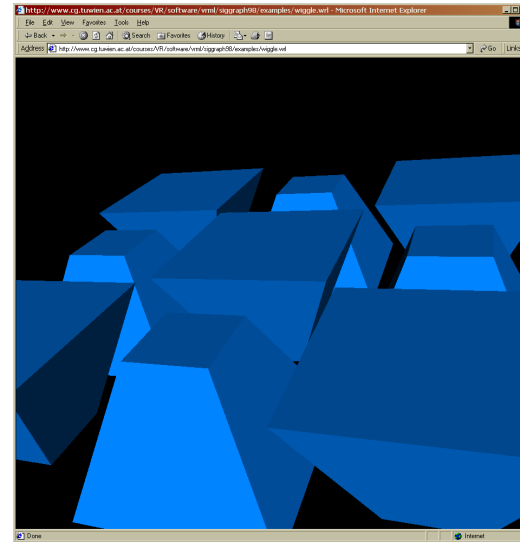
Transform { translation 0.0 0.0 -3.0 rotation 1.0 0.0 0.0 3.1415
    children USE Wig }
Transform { translation -3.0 0.0 -3.0
    children USE Wig }
Transform { translation 3.0 0.0 -3.0
    children USE Wig }

Transform { translation 0.0 0.0 3.0 rotation 1.0 0.0 0.0 3.1415
    children USE Wig }
Transform { translation -3.0 0.0 3.0
    children USE Wig }
Transform { translation 3.0 0.0 3.0
    children USE Wig }

DEF Clock TimeSensor {
    cycleInterval 2.0
    loop TRUE
    startTime 1.0
    stopTime 0.0
}

DEF Interpolator CoordinateInterpolator {
    key [ 0.0, 0.25, 0.5, 0.75, 1.0 ]
    keyValue [
# 1st coordinate set
-1.0 1.0 1.0,
 1.0 1.0 1.0,
 1.0 1.0 -1.0,
-1.0 1.0 -1.0,
-1.0 -1.0 1.0,
 1.0 -1.0 1.0,
 1.0 -1.0 -1.0,
-1.0 -1.0 -1.0,
# 2nd coordinate set
-0.5 1.0 0.5,
 0.5 1.0 0.5,
 0.5 1.0 -0.5,
-0.5 1.0 -0.5,
-1.5 -1.0 1.5,
 1.5 -1.0 1.5,
 1.5 -1.0 -1.5,
-1.5 -1.0 -1.5,
# 3rd coordinate set
-1.0 1.0 1.0,
 1.0 1.0 1.0,
 1.0 1.0 -1.0,
-1.0 1.0 -1.0,
-1.0 -1.0 1.0,
 1.0 -1.0 1.0,
 1.0 -1.0 -1.0,
-1.0 -1.0 -1.0,
# 4th coordinate set
```

Beispiel für
CoordinateInterpolator



wiggle.wrl

Beispiel für
CoordinateInterpolator



vrml/student_projects/WallClock.wrl

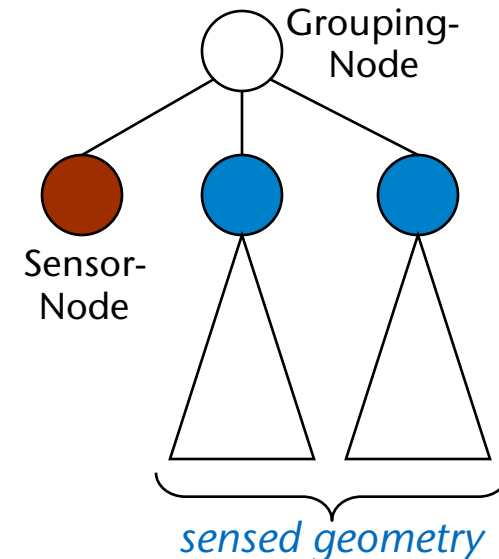
- Zur Abfrage von User-Input gibt es eine ganze Reihe von **Sensor**-Knoten
- Die meisten haben folgende Felder (zusätzlich zu ihren spezifischen):

```
SFString exposedField description ""
SFBool   eventOut      isOver
SFBool   eventOut      isActive
SFBool   exposedField  enabled      true
```

- Mit **enabled** kann man einen Sensor de-/aktivieren
- Das Feld **description** is freiwillig, sollte aber unbedingt ausgefüllt werden (sonst weiß man 2 Wochen später überhaupt nicht mehr, welche Funktion ein Sensor hat!)
- **isActive=true** gdw. alle Vorbedingungen für einen bestimmten Sensor sind erfüllt
- **isOver=true** gdw. die Maus sich über dem "heißen" Bereich des Sensors befindet

- Aktivierung:

- alle Sensor-Knoten sind einem Teil des Szenengraphen zugeordnet
- der Sensor ist aktiv gdw. der User irgend eine Geometrie in diesem Teil (die *sensed geometry*) angeklickt hat (bzw. hält)



- Mapping:

- Die meisten Sensor-Knoten mappen die 2D-Maus-Pos. auf eine 3D-Position in der VE (was natürlich nicht notw. die Hauptaufgabe ist)
- Dieses Mapping geschieht durch Schnitt des Strahls vom Viewpoint durch die Maus mit der Szene
- Diesen Schittpunkt kann man mit den Feldern **offset_changed**, **autoOffset**, und **trackPoint_changed** abfragen
 - Diese Felder ex. in allen Sensorknoten; die Bedeutung kann vom konkreten Mapping abhängen

■ TouchSensor:

- erzeugt Ausgaben, sobald die Geometrie geklickt wird
- wird oft zum Auslösen von Timern verwendet
 - Bsp.: ein virtueller Button soll eine Tür öffnen

■ PlaneSensor:

- konvertiert die select-and-drag-Bewegung in eine 3D-Bewegung, die aber auf eine Ebene im Raum eingeschränkt ist
- diese Ebene ist die lokale $z=0$ Ebene
- wird oft zum Bewegen von Geometrie auf einer Ebene verwendet
 - Verbinde das Feld `translation_changed` mit `set_translation` eines `Transform's`
- Die Koordinaten von `translation_changed` sind **relativ zum lokalen Koord.system!**

- **CylinderSensor:**

- Funktioniert ähnlich wie der PlaneSensor
- Unterschied: die Maus-Position wird auf einen Zylinder gemapt
- Ausgabe-Feld: rotation_changed
- Häufiger Verwendung: zur Implementierung von Drehknöpfen
- Gibt sehr viele weitere Parameter-Felder

- **SphereSensor:**

- you get the idea by now ...

- **KeySensor:**

- Interface zum Keyboard
- Ausgabe-Felder:

```
SFInt32  actionKeyPress  
SFInt32  actionKeyRelease  
SFString keyPress  
SFString keyRelease
```

- **StringSensor:**

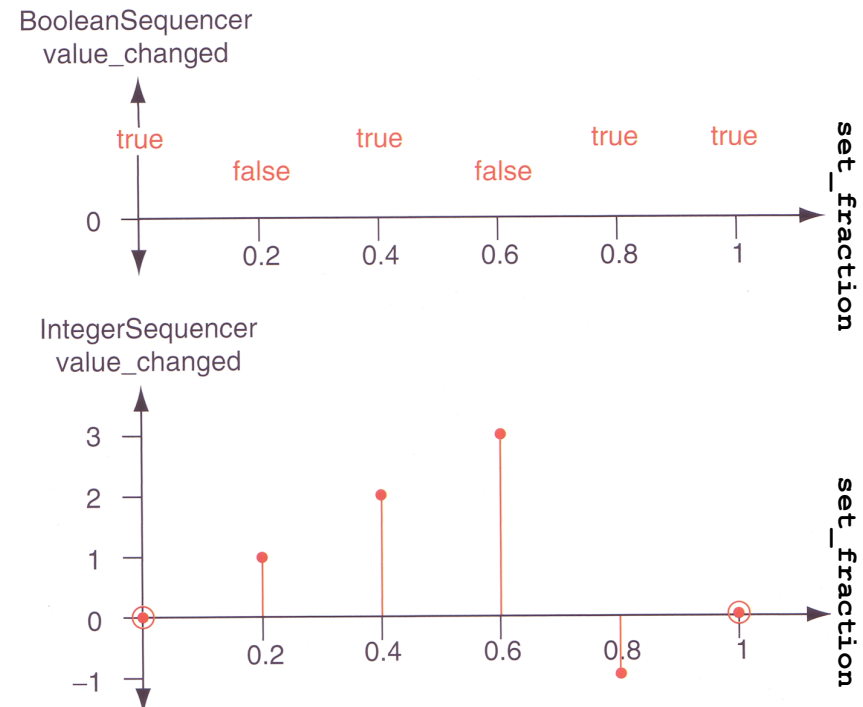
- String-basiertes Interface zum Keyboard
- Erlaubt die Eingabe kompletter Strings

- Diese sind soz. "glue nodes"
 - Z.B.: SFBool → SFTime, SFFloat → SFVec3f

- Sequencer-Knoten:

- Diskrete Variante der Interpolator-Knoten

- Ausgabe-Feld **value_changed** nimmt nur Werte aus dem Feld **keyValue** an
- Keine Interpolation
- Das Ausgabe-Feld ändert seinen Wert nur, wenn der Wert im Feld **set_fraction** von einem Interval [**key[i], key[i+1]**) in ein anderes Interval [**key[j], key[j+1]**) wechselt



- Varianten: **BooleanSequencer**, **IntegerSequencer**, **FloatSequencer(?)**

■ Trigger-Knoten

- **BooleanTrigger:** konvertiert SFTime → SFBool
 - Erzeugt immer einen True-Event, wenn ein Event eingeht
- **IntegerTrigger:** konvertiert SFBool → SFInt32
 - Der ausgegebene Wert kann an einem weiteren Feld spezifiziert werden
- **TimeTrigger:** konvertiert SFBool → SFTime
 - Erzeugt die aktuelle System-Zeit am Ausgabe-Feld, wenn ein Event eingeht

■ BooleanFilter:

```
SFBool eventIn  set_boolean
SFBool eventOut inputFalse
SFBool eventOut inputNegate
```

- `inputFalse` liefert **True**-Event, wenn `set_boolean` **False**-Event empfängt
- `inputNegate` liefert den negierten Event von `set_boolean`

■ BooleanToggle:

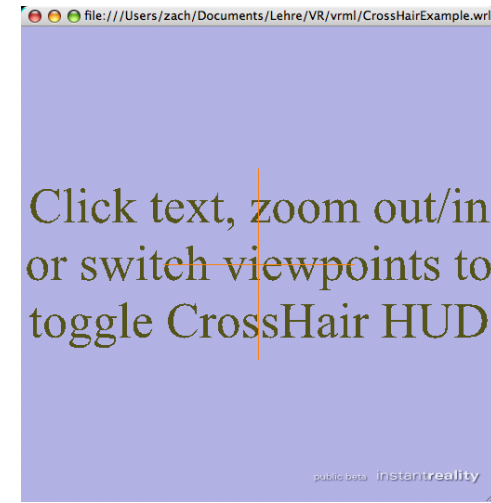
- "Toggle switch" = Kippschalter

```
SFBool eventIn      set_boolean
SFBool exposedField toggle
```

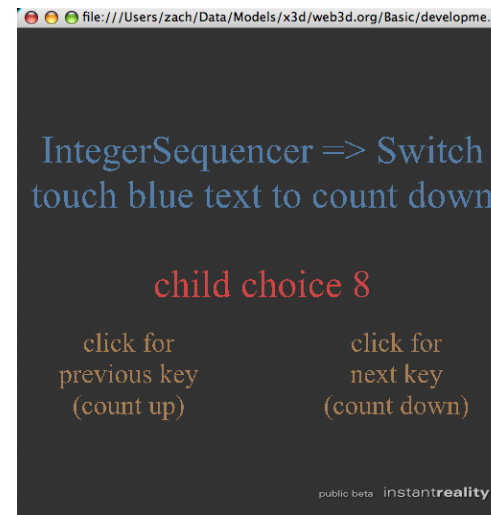
- Das Feld `toggle` kippt jedesmal, wenn ein **True**-Event empfangen wird

- Beispiel für
 - Geometrie, die fest bleibt relativ zum Viewpoint
 - **Switch** (und **PROTO**)
 - **ProximitySensor**, **TouchSensor**,
 - **BooleanToggle**, **-Trigger**, **-Filter**

- Beispiel für
 - **IntegerSequencer** und **Switch**



CrossHair.wrl



IntegerSequencer.wrl

- Erlauben die Implementierung von komplexen Verhalten in einer sog. "Skript"-Sprache
 - Der Standard läßt die konkrete Sprache offen; er definiert statt dessen ein sog. *Scene Access Interface (SAI)*
 - Früher *EAI (External Access Interface)*
 - Typisch sind Java und/oder Javascript (aka ECMAScript)
 - Für diese gibt es dann sog. Bindings, d.h., ein konkretes API, das das SAI impl.
 - InstantReality kann beides, FreeWRL ?
- IMHO ein Design-Bug im Standard: "Browsers are not required to support any specific language"

- Deklaration / Syntax:

```
Script {  
  exposedField MFString url  
  field SFBool directOutput FALSE  
  field SFBool mustEvaluate FALSE  
  # And any number of:  
  eventIn fieldType fieldName  
  exposedField fieldType fieldName initialValue  
  eventOut fieldType fieldName  
  field fieldType fieldName initialValue  
}
```

- Zur Erinnerung die Access Types:

VRML97	Bedeutung
eventIn	nur Eingabe
eventOut	nur Ausgabe
field	nur Daten
exposedField	Kombination

- Das `url`-Feld:

- Link auf einen Javascript-File:

```
url [ "MyBehavior.js"  
      "http://www.my.site/MyBehavior.js" ]
```

- Link auf einen Java-File:

```
url [ "MyBehavior.class"  
      "http://www.my.site/MyBehavior.class" ]
```

- Javascript-Source-Code:

```
url [ "javascript:  
      var x, y, z;  
      function initialize( timeStamp )  
      {  
          ...  
      }  
      "  
      ]
```

- Zu jedem **eventIn**-Feld gibt es eine Funktion mit demselben Namen
 - Heißt **Event-Handler**
 - Wird irgendwann innerhalb der Event-Kaskade aufgerufen, falls Event eintrifft
 - Parameter: Wert des Events (Kopie der Daten) & Timestamp
- Zu jedem **eventOut**-Feld gibt es eine implizit vordefinierte Variable mit dem Namen **xxx_changed** und passendem Typ
 - Schreiben der Variable = Generieren eines Events
 - Event wird nur 1x pro Time-Stamp erzeugt!
- Zu jedem **eventIn**-Feld gibt es eine implizit vordefinierte Variable mit dem Namen **set_xxx**

```
DEF SomeNode Transform
{
  translation 0 0 0
  children [ ... ]
}

Script
{
  field SFNode tnode USE SomeNode
  eventIn SFVec3f pos
  directOutput TRUE
  url [ "javascript:
      function pos(value, timestamp)
      {
        tnode.set_translation = value;
      }
    " ]
}
```

- Das Feld **directOutput** muß man auf **TRUE** setzen, falls der Script-Knoten andere Knoten direkt manipuliert; falls **directOutput= FALSE**, dann darf der Script-Knoten den Rest der Szene nur über Routes modifizieren!
- Innerhalb einer Event-Kaskade bekommen alle Funktionsaufrufe denselben Timestamp
- Innerhalb eines Frames können die Event-Handler mehrfach aufgerufen werden
- Spezielle Funktionen (hier in Javascript):
 - **eventsProcessed ()** : wird am Ende einer Event-Kaskade aufgerufen
 - **prepareEvents ()** : wird vor dem Route-Processing aufgerufen
 - Z.B. zur Abfrage von externen Geräten
 - **initialize ()** : wird direkt nach dem Laden der Szene aufgerufen

- Alle Knoten haben 2 (bislang verschwiegene) Felder

```
inputOnly MFNode addChilden  
inputOnly MFNode removeChildren
```

- Das Execution-Model mit Skript-Knoten:

```
1. Update Camera (based on currently active Viewpoint node)  
2. Evaluate Sensor nodes and  
   all script-nodes' prepareEvents() function  
   (→ initial events)  
3. for all initial events:  
4.   process the event cascade, i.e., route events  
5.   if any script node was visited during routing:  
6.     evaluate its eventsProcessed() function  
7. Render scene; swap rendering buffers  
8. Browser's clock ← system clock (one "clock tick")
```

- Output-Events eines Skripts bekommen denselben Timestamp wie dessen Input-Events
- Zugriff auf Elemente eines Vektors (z.B. **SFVec3f**) :
 $v[0], v[1], v[2]$
- Zugriff auf Elemente eines **MF**-Feldes (Arrays): **values [0], ...**
 - Beispiel: **MFRotation values** → **values [0] [3]** = Winkel der 1. Rot

- Zur Laufzeit generierte Geometrie:



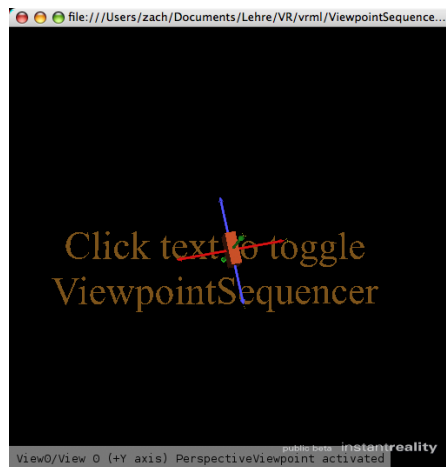
<examples/CreateVrmlFromStringRandomBoxes.wrl>

- Beispiel für

- "printf"-Ausgabe auf der Konsole des Browsers
- Javascript-Knoten
 - Zugriff auf andere Knoten der Szene mittels

```
field SFNode AliasNode USE SceneNode
```

- Billboard
- Funktionalität: schaltet verschiedene Viewpoints durch



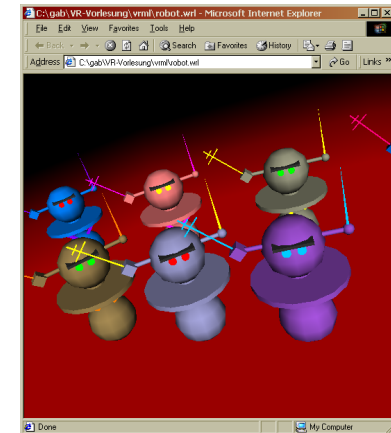
<examples/ViewpointSequencer.wrl>

Ein Bug in der X3D-Spezifikation

- Felder mit einem Namen zzz, die den Access-Type exposedField haben, haben implizit den Namen zzz_changed, wenn sie als Ausgabe-Feld verwendet werden, und den Namen set_zzz, wenn sie als Eingabe-Feld verwendet werden
- Problem mit inputOutput-Feldern in Script-Knoten:
 - Es wird ein Event-Handler (= Funktion) zzz() definiert ...
 - ... und eine Variable zzz
 - Ist in Javascript nicht erlaubt, da Funktionen und Variablen im selben Namespace leben!

- Definition neuer Knotenarten; faßt zusammen:
 - Knoten (Shapes, Sensors, Interpolators, etc.)
 - Script-Knoten
 - Routes
- Beispiel:

examples/robot.wrl



```

PROTO Robot
[ field SFCOLOR eyeColor 1.0 0.0 0.0
  ...
]{
  Shape { appearance Appearance {
    material Material {
      diffuseColor IS eyeColor
    } }
  ...
}
    
```

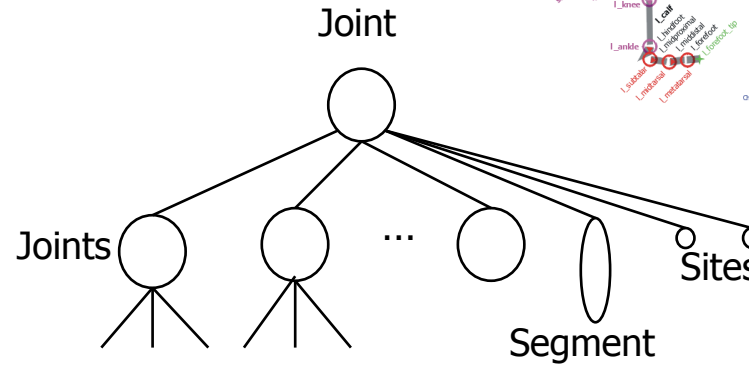
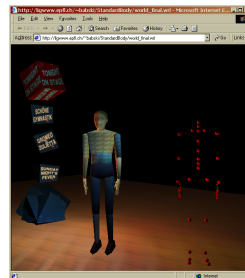
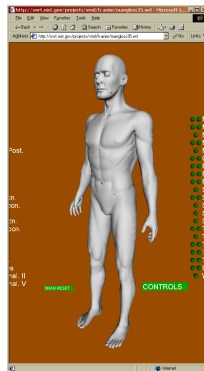
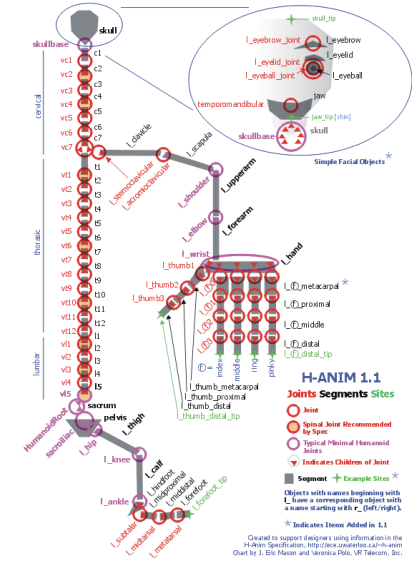
Name der neuen Knoten-Klasse

Interface (Felder & Events)

Body (Implementierung)

Zuweisung des Interfaces

- In X3D:
 - Es kommen laufend neue Knoten hinzu, z.B. für Shader
- H-Anim:
 - Standard zur Spezifizierung von "humanoiden" Figuren in VRML97/X3D
 - Joints = Baumstruktur
 - Segments = Geometrie
 - Sites = "Handles" (ausgezeichnete Punkte im Koord.system des Joints)



Tips & Tricks zum Entwickeln mit X3D/VRML

- Editor:
 - Man wird früher oder später X3D-Code "von Hand" editieren müssen
 - ASCII-Editor verwenden, der Syntax-Highlighting für VRML/X3D hat!
 - ... und der Klammer-Matching beherrscht!
- X3D-Edit ?

- Im wesentlichen nur "printf"-Debugging möglich
- Beispiel: ein Debug-Knoten

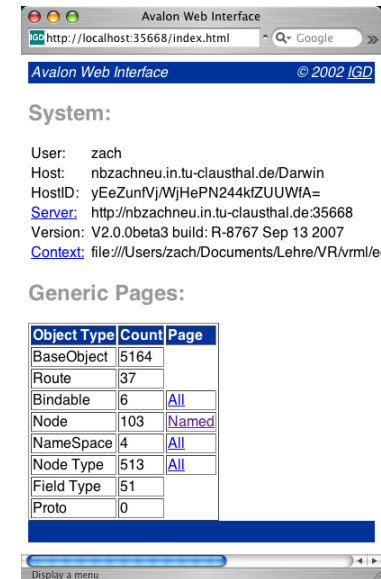
```
DEF Debug Script {
  eventIn MFVec3f set_coord
  eventIn SFFloat set_float
  url [ "javascript:
    function set_coord( value, timestamp )
    {
      print( 'Debug: coord = ' + value + '\n' );
    }
    function set_float( value, timestamp )
    {
      print( 'Debug: float = ' + value + '\n' );
    }
  " ]
}
```

Beispiel in examples/wiggle_with_debug.wrl

- Alternative in InstantReality:
 - Mittels des Nicht-Standard-Knotens **Logger**

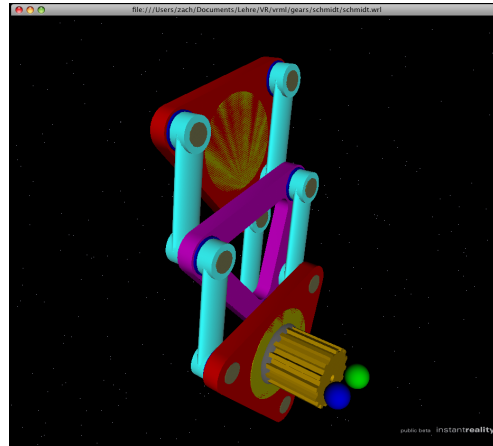
```
DEF Log Logger
{
  level 3      # 0 - ..
  logFile ""   # default = console
}
ROUTE Clock.fraction_changed TO Log.write
```

- Extrem praktisches Feature: Der X3D-Browser von InstantReality erlaubt es, zur Laufzeit den Szenengraphen zu beobachten und sogar Felder zu verändern!
- Anleitung:
 - In InstantPlayer: Help → Web Interface Scenegraph
 - Ein Browser-Fenster öffnet sich
 - In der Tabelle klicken: "Named" → "scene (Scene)"
- Demo:
 - Szene: [eg2001competition/ah2k/ah2k.wrl](http://localhost:35668/eg2001competition/ah2k/ah2k.wrl)
 - Links: Named → scene (Scene) → DEF spinClock TimeSensor → enabled TRUE/FALSE

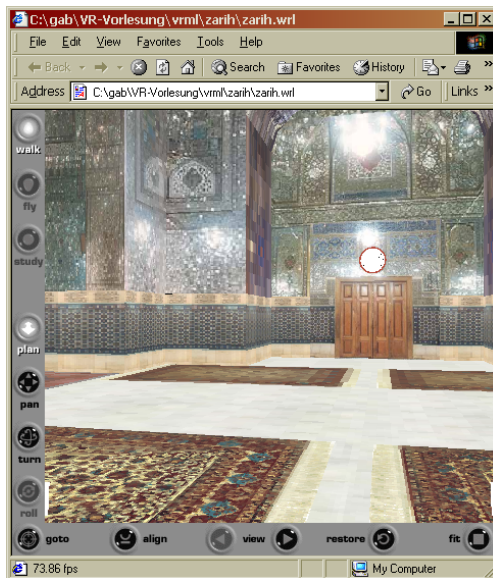


Demos

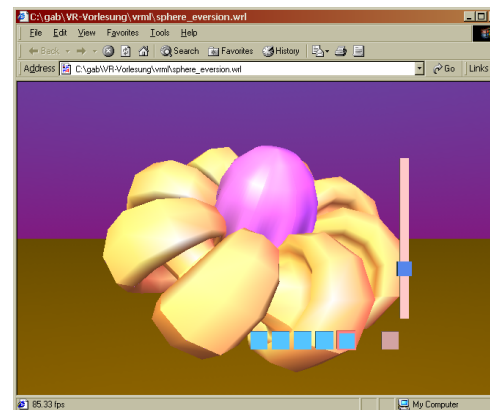
Veranschaulichung von komplizierten Kinematiken (hier: *Schmidt Offset Coupling*)



Spiele
(Quelle: Eurographics 2001 competition)



Cultural heritage
(Quelle: www.aqrazavi.org)

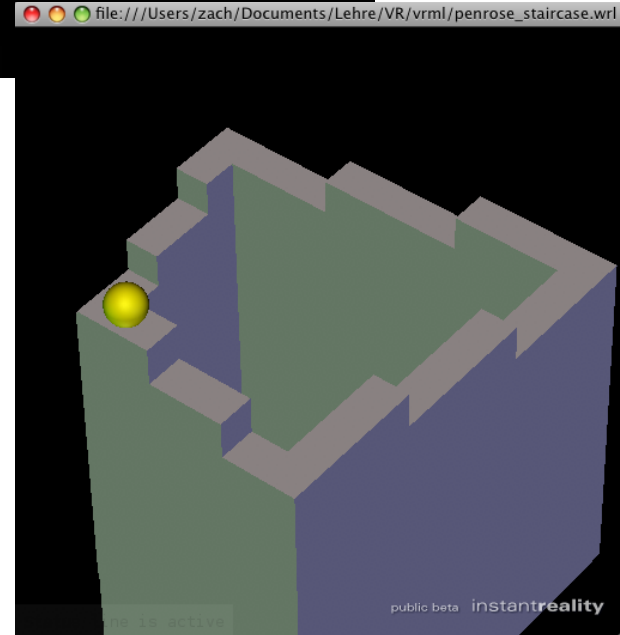
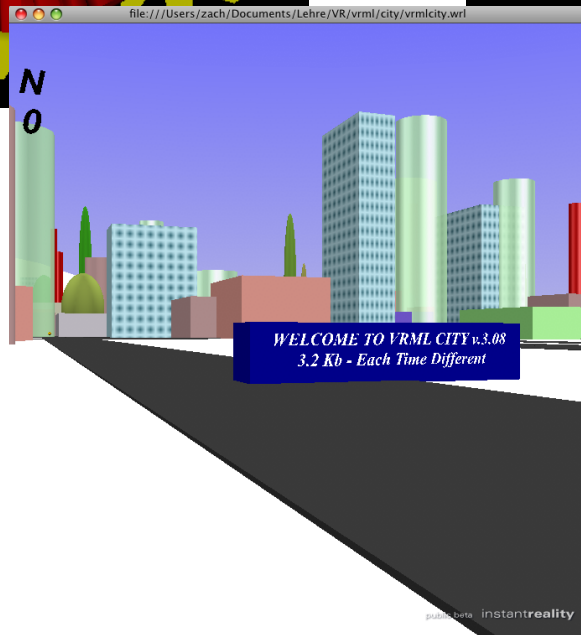
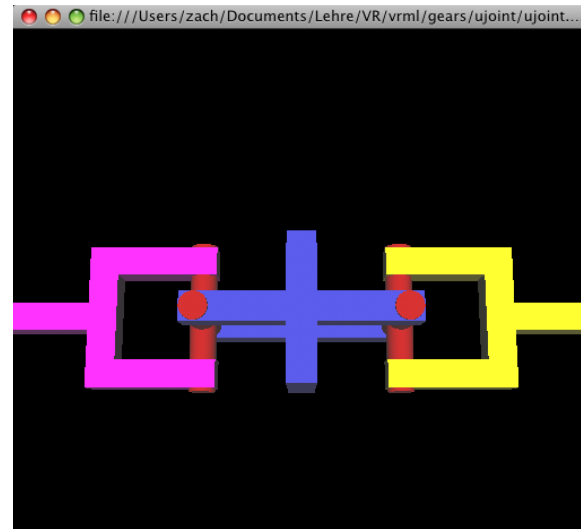
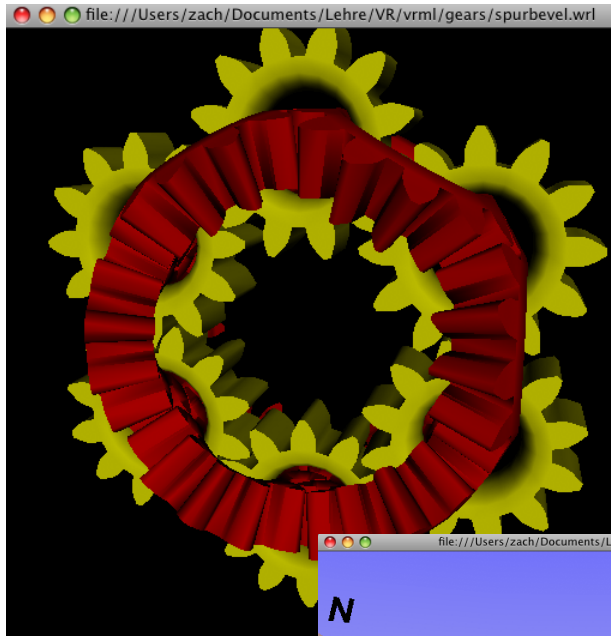


Edutainment (Wissenschaft?),
Wissenstransfer
Bsp.: *sphere eversion*

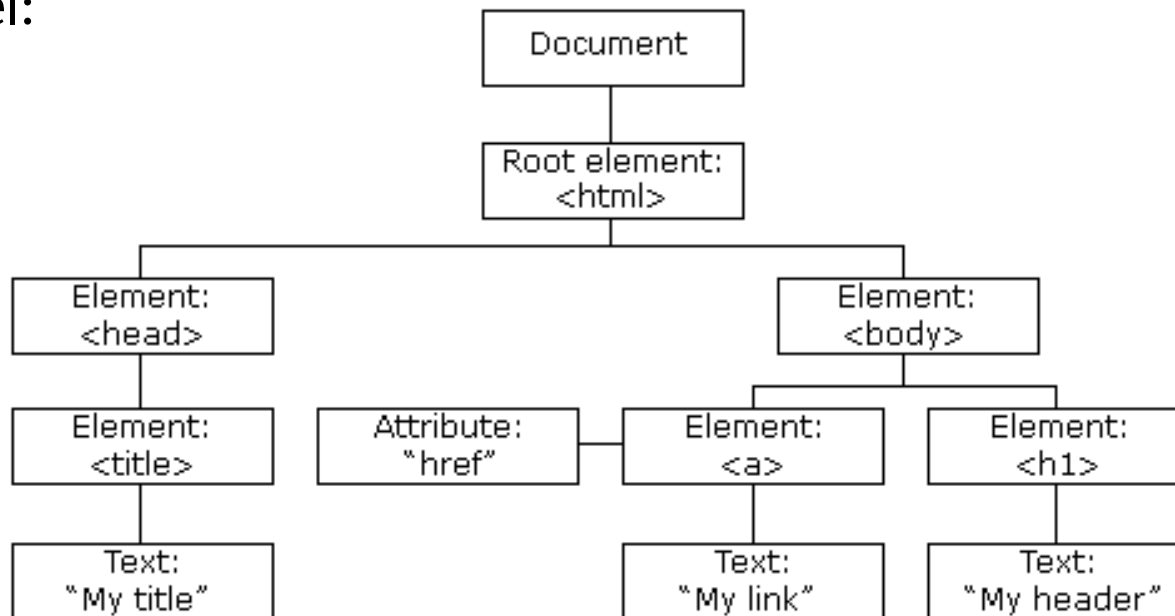
Sphere Eversion (Video)



<http://www.youtube.com/watch?v=BVVfs4zKrgk>



- Aktuelle Bestrebung des W3C, X3D als festen Bestandteil von HTML5 zu integrieren
- DOM = Document Object Model = Spezifikation ...
 1. wie ein HTML/XML-Dokument als Baum repräsentiert wird; und
 2. wie man auf die Elemente des Baumes zugreift
- Beispiel:



```
...
<body>
  <h1>X3D DOM Events</h1>
  <x3d xmlns="http://www.web3d.org/specifications/x3d-3.0.xsd">
    <Scene>
      <Transform>
        <Shape>
          <Box size="4 4 4" />
        </Shape>
        <TouchSensor id="ts" DEF="ts" />
      </Transform>
    </Scene>
  </x3d>
  <script type="text/javascript">
    // The namespace URIs
    var xhtml_ns = "http://www.w3.org/1999/xhtml";
    var x3d_ns = "http://www.web3d.org/specifications/x3d-3.0.xsd";
    // Get elements using namespaces
    var h1=document.getElementsByTagNameNS( xhtml_ns, "h1" );
    var x3d =document.getElementsByTagNameNS( x3d_ns, "x3d" )[0];
    var ts = x3d.getElementsByTagName( "TouchSensor" )[0];
    ts.addEventListener( "touchTime",
                        function() {alert("clicked"); },
                        false );

  </script>
</body>
```


- Demo:
demos/x3dom/fanOutRoute.xhtml

- Weiterführender Link:
<http://www.x3dom.org/>
 - Beta-Versionen, Doku, FAQs,

