



Einführung in VRML

- VRML heisst Virtual Reality Modeling Language, ein Teil von “Web3D”/”X3D”.
- VRML ist einfach ein 3D Datenaustausch-Format: Es besitzt Sprachelemente für die heute in 3D-Applikationen üblichen Elemente wie Objekte, Licht, Texturen...
- VRML ist ähnlich zu HTML: Dadurch kann man es einfach im Internet als Dokument transportieren und in einem geeigneten “Browser” anzeigen lassen.



VRML + HTML + Scripting/Java

- VRML in HTML einbetten:
Dies geht mit Hilfe des `<EMBED>` oder `<OBJECT>` HTML Tags. Nicht jeder Browser muss das unterstützen.
- Script-Code in VRML eingebettet:
Standard-Feature von VRML 2.0; benutzt einen so genannten “Script Node”.
- Java Applet kommuniziert mit VRML
Browser: Erweiterung von VRML 2.0, genannt “EAI” oder “External Authoring Interface”.



Überblick über VRML (1)

- “Scene Graph” Struktur:
Eine VRML Datei ist hierarchisch aufgebaut. Objekte werden “Nodes” (Knoten) genannt.
- Nodes:
Es gibt ca. 54 verschiedene **Typen von Nodes**, z.B. für geometrische Grundformen, Erscheinungs-Eigenschaften, Klänge usw. Nodes speichern ihre Daten in Felder, von denen es 20 verschiedene Typen gibt.



Überblick über VRML (2)

- **Event Architektur:**
VRML 2.0 kennt einen “Event” oder Nachrichten-Mechanismus, über den Nodes in einem Scene Graphen miteinander kommunizieren können. Jeder Knotentyp definiert individuelle Events und Event-Typen, die Knoten diese Typs senden und empfangen können. Route-Statements definieren die Verbindungen zwischen Event-Quellen und -Senken.
- **Sensoren:**
Grund-Elemente für die User-Interaktion und Animation. Animationen werden über “TimeSensors” gesteuert, die Events generieren und damit die Animation steuern.



Überblick über VRML (3)

- **Script Nodes:**
Können zwischen Event-Quellen und -Senken geschaltet werden, um das Verhalten z.B. einer Animation zu beeinflussen.
- **Interpolator Nodes:**
Werden benutzt, um (getriggert durch Events eines TimeNodes) Objekte auf bestimmten Bahnen zu bewegen.
- **Inline Node / EXTERNPROTO:**
Einbindung anderer VRML Welten (irgendwo aus dem WWW) als “Subroutine”.



Los gehts mit VRML

- Buch: VRML 97 bei Addison Wesley (1998)
- Tutorial im Web:
<http://web3d.vapourtech.com/tutorials/vrml97/index.html>
- File-Struktur eines .wrl Files:
 - Header: `#VRML V2.0 utf8`
 - Scene Graph: Besteht aus Nodes, die hierarchisch angeordnet die Geometrie beschreiben, plus Nodes für Event-Erzeugung und Routing.
 - Prototypes: Dadurch kann man Nodes-Typen erweitern.
 - Event Routing.



Syntax in VRML-Files

- VRML unterscheidet Groß- und Kleinschreibung: **Sphere != sphere**
- Kommentare beginnen mit #
Alles **hinter #** in der Zeile ist Kommentar
- Whitespace sind *Space, Tab, Komma, Linefeed, Carriage Return*: Sie werden ausserhalb von Strings **ignoriert**.
- Namen dürfen keine Sonderzeichen oder Whitespace enthalten.



Beispiel-VRML-File

```
#VRML V2.0 utf8

Transform {
  translation 2 0 3
  children [
    Shape {
      geometry Sphere { radius 1.5 }
    } # Ende von Shape
  ] # Ende von children
} # Ende von Transform
```



Erklärung zum Beispiel

- # kennzeichnet Kommentare.
- Transform { } ist ein Gruppenknoten und schliesst einen Scene Graphen ein. Hinter der schliessenden Klammer kann man zu besserer Lesbarkeit auf das zugehörige Schlüsselwort hinweisen.
- Translation positioniert den Gruppenknoten im Raum. Ausprobieren!
- Children[] ist ein Feld, das die Kindknoten des Gruppenknotens sammelt, hier nur ein einziges: Eine Kugel mit Radius 1.5



Absolute und relative Position

- Wenn man **mehrere “transform{” Knoten nacheinander** aufführt, die jeweils ein anderes “translation” Statement enthalten, wird jeder dieser Knoten absolut positioniert.
- Man kann auch mehrere solche **Knoten ineinander verschachteln**, indem man nach dem “shape{” Statement den nächsten “transform{” Knoten einfügt.
- Merke: Das ist nicht dasselbe, der **Kindknoten erbt die Position des Elternknotens**, und “translation” ist relativ dazu!

Beispiel relative Translation

```
#VRML V2.0 utf8

Transform { translation 2 0 3

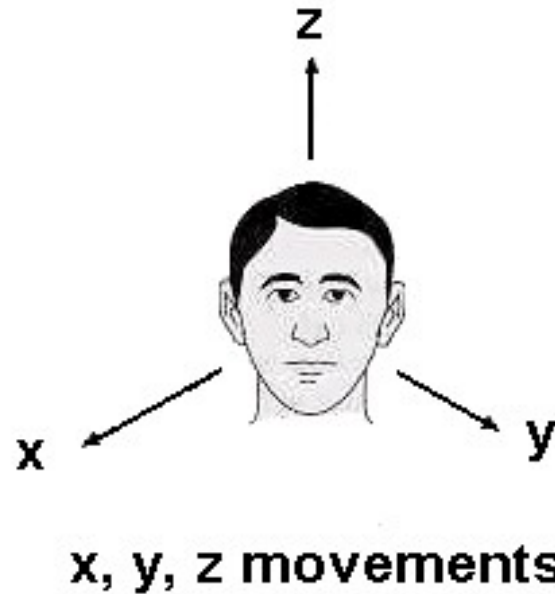
  children [

    Shape { geometry Sphere { radius 1.5 } } # Shape

    Transform {
      translation 1 0 1
      children [
        Shape {
          geometry Cone { bottomRadius 1 height 2}
        } # Shape Cone
      ] # children Nr. 2
    } # Transform Nr. 2

  ] # children
} # Transform
```

Rotations-Richtungen



roll



pitch



yaw

Drehen von Objekten

- Zum Drehen benutzt man das Feld “rotation” des Knotens “transform{”.
- rotation nimmt Parameter vom Typ SFRotation: `yaw pitch roll value`
- Das heisst: Drehung um die X-, Y- und Z-Achse. Und der Drehwinkel im Bogenmass.
- Beispiel: `1 0 0 3.1415`
bedeutet Drehung um X-Achse, etwa 180°



Shapes...

- **Box**

- size `<x-size> <y-size> <z-size>`

- **Sphere**

- radius `<value>`

- **Cone**

- bottomRadius `<value>`
- Height `<value>`
- bottom `<FALSE|TRUE>`

- **Cylinder:** `radius, height, top`



Farbe ins Spiel bringen

- Im Kindknoten Shape ist ein Feld “appearance” zu setzen.
- Ein weiterer Knoten “material” kennt Farben:

```
Shape {  
    appearance Appearance {  
        material Material  
            { diffuseColor 1 0 0 } # Rot  
    }  
    geometry Sphere {}  
}
```

Beispiel Farben

```
Shape {  
    appearance Appearance {  
        material Material {  
            emissiveColor 0 0.8 0  
            transparency 0.5  
        }  
    }  
    geometry Box {  
    }  
}
```



Spezielle Farbwerte

● Weiss:	1	1	1
● Schwarz:	0	0	0
● Rot:	1	0	0
● Grün:	0	1	0
● Blau:	0	0	1
● Grau:	0.5	0.5	0.5



Mehr zu material{ }

- `diffuseColor`
Die normale Farbe des Objektes
- `specularColor`
Die Farbe der Lichtreflexe
- `emissiveColor`
Die Farbe, in der das Objekt selbst leuchtet
- `ambientIntensity`
Die Menge reflektierten Umgebungs-Lichtes
- `shininess`
Wie stark das Objekt reflektiert
- `transparency`
Transparenz des Objektes



Texturen (statt material{ })

- Muster (Texturen) ermöglichen, realistische Oberflächen darzustellen
- Beispiel:

```
Appearance {  
    texture ImageTexture {  
        url "brick.jpg"  
        repeatT TRUE      # vertikal  
        repeatsS TRUE     # horizontal  
    }  
}
```



MPEGs als Texturen

- MovieTexture bringt ein MPEG1 File auf die Oberfläche eines Objektes, zus. Felder:
 - speed
Geschwindigkeits-Faktor (1=Normal, 2=Doppelt usw.).
 - loop
TRUE wenn Wiederholung, FALSE wenn nicht.
 - startTime
Start-Zeitpunkt (Sekunden seit 1.1.1970 ;-)
 - stopTime
Stop-Zeitpunkt (dto.)
- **Wer es zum Laufen kriegt, soll sich melden...**



Text ist ebenfalls möglich...

```
geometry Text {  
    string ["Hello", "World"]  
    fontStyle FontStyle {  
        size 0.8  
        justify "MIDDLE"  
    }  
    maxExtent 5  
    length [3, 3]  
}
```



Vor-Übungen zur Belegarbeit

- Setzen Sie Ihre einfachen Welten fort, die aus den Grundgeometrien “Sphere”, “Cone”, “Box” und “Cylinder” bestehen.
- Nehmen Sie Elemente zur Wiederverwendung hinzu!
- Experimentieren Sie mit **Text** und ***Texture Nodes**!
- Informieren Sie sich über Events & Routen!



Sprachelemente für Abstraktionen

- Grössere Projekte müssen wie üblich bearbeitet werden:
 - Anforderungsanalyse / Spezifikation,
 - Strukturierung in Abstraktions-Elemente,
 - Implementierung der Elemente,
 - Implementierung der Struktur.
- Einfache Variante: DEF / USE
- Höhere Form: PROTO u. EXTERNPROTO

Wiederverwendung PROTO

```
PROTO VBox [  
    field SFCOLOR boxColour 1 0 0  
]  
  
{  
    Shape {  
        appearance Appearance {  
            material Material {  
                diffuseColor IS boxColour  
            }  
        }  
        geometry Box {  
        }  
    }  
}
```

```
--> VBox { boxColour 0 1 0 }
```

Wiederverwendung EXTERN...

- PROTO Definition ist in einem anderen File

```
EXTERNPROTO VBox [  
    field SFCColor boxColour  
]  
  
[  
    "proto.wrl"          # Alternativen 1 + 2  
    "http://www.mydomain.com/protos/proto.wrl"  
]
```

- Wenn mehrere Prototypen in einem .wrl File enthalten sind, benutzt man Targets:

```
"proto.wrl#VBox"
```



Beispiel zu EXTERN PROTO

- Regale mit sich wiederholenden Elementen
- Sehr große Anzahl Elemente
- Relativ große Welt



DEF und USE

- Mit DEF gibt man einem Knoten einen Namen, um ihn später in demselben VRML File wieder zu verwenden:

```
DEF mein_knoten Transform { ... }
```

- Mit USE kann man den Namen an anderer Stelle wieder benutzen, um einen gleichartigen Knoten einzufügen oder den Knoten zu referenzieren (z.B. in ROUTE Knoten, siehe weiter hinten):

```
ROUTE mein_knoten.collideTime TO SOUND2.startTime
```



Details zu PROTO

- Mit PROTO definiert man eigene Knoten.
- Eine PROTO Anweisung besteht aus Name, `[Interface]` und `{Implementierung}`.
- Der erste Knoten in der Implementierung definiert den “Typ” des neu definierten Knotens, weitere Knoten sind nicht referenzierbar und erscheinen auch nicht.
- USE / DEF funktioniert nicht über PROTO-“Grenzen” hinweg.



Events und ihr Einsatz (1)

- “Sensor” Knoten können Nachrichten absetzen, wenn Ereignisse eintreten.
- Verschiedene Knoten können derartige Nachrichten empfangen.
- StartTime und stopTime sind solche Nachrichten oder “Events”.
- Man kann z.B. einen Timer-Knoten definieren, der in bestimmten Intervallen Events erzeugt.

Events und ihr Einsatz (2)

- Eine Event-Verbindung definiert man durch ein ROUTE Statement.
- Damit werden ausgehende Events eines Knotens zu “Eingängen” eines anderen Knotens geleitet.
- Damit das geht, müssen die Knoten mit DEF benannt werden, z.B.:

```
DEF TIMER TimeSensor {  
    cycleInterval 5.0  
    loop TRUE  
}
```

Events und ihr Einsatz (3)

- Der Empfänger sieht z.B. so aus:

```
Sound {  
    minFront 10, minBack 10  
    maxFront 50, maxBack 50  
    source DEF SOUND AudioClip {  
        url "PICARD.WAV"  
    }  
}
```

- Dazu die Verbindung:

```
ROUTE TIMER.cycleTime TO SOUND.startTime
```

Code-Beispiel

```
DEF TIMER TimeSensor {  
    cycleInterval 5.0  
    loop TRUE  
}  
  
Sound {  
    minFront 10    minBack 10    maxFront 50    maxBack 50  
    source DEF SOUND AudioClip {  
        url "PICARD.WAV"  
    }  
}  
  
Shape { ... }  
  
ROUTE TIMER.cycleTime TO SOUND.startTime
```



Das nächste Projekt (Beleg)

- Ziel: Komplexe(re) virtuelle Welt
- Vorgehensweise:
 - Thema wählen,
 - Req.-Definition / Spezifikation,
 - Abstrahieren / Strukturieren der Bausteine,
 - Implementierung.
- Dokumentation der Implementierung mit Begründung der Design-Entscheidungen.