VRML Primer and Tutorial

Daniel K. Schneider[1]
and Sylvere Martin-Michiellot[2]
TECFA[3], Faculte de Psychologie et des sciences de l'education,
University of Geneva,
Email: Daniel.Schneider@tecfa.unige.ch

March 18, 1998

[1]http://tecfa.unige.ch/tecfa-people/schneider.html
[2]http://tecfa.unige.ch/tecfa-people/sylvere.html
[3]http://tecfa.unige.ch/

DRAFT Version 1.1a - use at your own risk !!
This evolving document is an introduction to VRML 97.
See section 8.6.1 for the version history. We will work on it again in the next few weeks (test-version).
DON'T bookmark any pages inside this tree.
Node names are generated dynamically and will change !

This manual is available over the WWW at:
http://tecfa.unige.ch/guides/vrml/vrmlman/vrmlman.html.
Ideally you should also load the on-line version while you are working through this printed version, because all URLs that appear as footnotes here (including the examples) are clickable in your browser.
Most examples can be found in the Index. See also our Examples directory at http://tecfa.unige.ch/guides/vrml/examples/[1].

[1] http://tecfa.unige.ch/guides/vrml/examples/

Contents

List of Tables

List of Figures

Chapter 1

Starting with VRML

Unless you already know some VRML, you probably should go through this in linear fashion and build your own examples while you are trying to digest this.

Style is rather example based, although we include frequently pointers to the =>VRML 2.0 specification[1] that we have mirrored at Tecfa. They are signaled with an "=>". (Note that some of those specification files are rather lengthy so use a disk cache with your browser).

1.1 Introduction to the VRML standard and 3D Graphics

VRML stands for "Virtual Reality Modeling Language". It allows to specify dynamic 3D scenes through which users can navigate with the help of a VRML browser. VRML scenes can be distributed over the World-Wide Web and browsed with special VRML browsers, most of which are plug-ins for Netscape or Internet Explorer. VRML is well integrated with the WWW, e.g. VRML scenes can be connected with other VRML scenes (as well as other WWW formats) via URLS.

The previous standard was VRML 1 and allowd to specify static scenes only. Forget about it! The current standard is VRML 97, which in its draft phase was known as VRML 2. The VRML 97 Specification[2] contains many changes to the VRML 1 language. It adds audio, interactive objects, behavior and scripting among other things. In this sense, VRML 97 is far more complex than VRML 1.

A future version (or layer of VRML) will add "multi-user shared experience", i.e. some kind of standardized interactive multi-user technology.

1.1.1 What is VRML?

The Virtual Reality Modeling Language (VRML) can been seen as a 3-D visual extension of the WWW. People can navigate through 3-D space and click on objects representing URLs (including other VRML worlds). Often, VRML is pronounced like "Vermal", not "V-R-M-L".

As Mark Pesce [Pesce, 1995, p. 16] points out, the WWW had two fundamental dimensions: *connectivity* (the http protocol) and *interface* (i.e. the rendering of content, especially HTML and embedded URLS). VRML inserts itself seamlessly in the Web's connectivity. VRML browsers can access other VRML files via an URL. They can access any other format that then is passed to another application (e.g. an HTML browser or a HTML window). On the other hand HTML browsers can be configured to fire up VRML helper applications (or plug-ins). HTTP servers, finally, can be configured to tell the client that a VRML (*.wrl) document is transferred.

A short word on its history: The major impulse for VRML can be traced back to a "birds of the feature sessions" on "Virtual Reality Markup Languages" at the First International Conference

[1] http://tecfa.unige.ch/guides/vrml/vrml97/spec/index.html
[2] http://tecfa.unige.ch/guides/vrml/vrml97/spec/

on the World-Wide-Web[3], May 25-27, 1994 at CERN in Geneva. It's conceptual origins are older, e.g. (a) Science Fiction literature (e.g. [Gibson, 1994], [Stephenson, 1992]), (b) Mark Pesce's, P. Kennard's and Toni Parisi's "Labyrinth" system ([Pesce et al., 1994]) and proposal for a 3-D navigation and representation scheme and (c) more generally 3-D computer graphics (including VR). Based upon SGI's "Open Inventor" format, a almost final draft for VRML 1.0 was presented at the second WWW conference[4] in fall 94 in Chicago. On April 3, 1995 SGI presented WebSpace, the first publicly available VRML browser. So all in all it took about a year to set standards and make the first browser available. Since VRML is a relatively simple format building upon a well defined standard, very quickly a number of modeling tools and convertors also became available.

In the next sections we will look at simple static VRML scences. These are built with VRML's *symbolic* description language. Note different VRML browsers all have a different user interface (e.g. for navigation and object examination). They also render things a bit differently. Most will also give you a "quality" choice (e.g. faster renderning and lower quality vs. slower rendering but better quality. For now, let's just assume that the user can move himself though 3D space by moving a **camera** through the space (and therefore what he sees on the display is what sees "his" camera).

1.1.2 VRML resources

On-line resources: There are several good resources about VRML on the Net. Since links move a lot, no pointers here, see our VRML pointers[5] instead. You will find most everything you want (including links to other indexes).

Books

- Jed Hartman's and Josie Wernecke's "VRML 2.0 Handbook, Building Moving Worlds on the Web" ([Hartman et Wernecke, 1996] is my recommended book for learning VRML 2. It includes discussion of a complete example.

- Ames', Nadeau's and Moreland's "The VRML 2.0 Sourcebook" ([Ames et al., 1996a]). Useful if you want to learn the details. RECOMMENDED if you are interested in the graphics part.

- Carey's and Bells's "The Annotated VRML 2.0 Reference Manual" ([Carey et Bell, 1997]). If you need a reference manual on paper.

- Lea, Matsuda, and Miyashita's "Java for 3D and VRML Worlds" ([Lea et al., 1996]). For VRML and JAVA script nodes.

- Roehl's et al. "Late Night VRML 2.0" ([Roehl et al., 1997]) for advanced VRML and Java. Recommended if you plan to go far and want to buy just one single book.

Note that all of those books do have outdated parts.

Some texts about Virtual Environments

1. Kevin Hugues' "From Webspace to Cyberspace" ([Hughes, 1995]) is good conceptual reading. Version 1.1 is available from Tecfa (locally) in PDF form.

2. See also our Kaspar's Cyberspace Pointers[6] for things on-line.

[3] http://www.elsevier.nl/cgi-bin/ID/WWW94
[4] http://www.ncsa.uiuc.edu/SDG/IT94/IT94Info-old.html
[5] http://tecfa.unige.ch/guides/vrml/pointers.html
[6] http://tecfa.unige.ch/guides/cspace-pointers.html

Specifications

1. The "VRML 97"[7] VRML specifications page at the VRML Consortium. A copy is available at Tecfa[8] (downloaded Jan 98)

1.1.3 An introduction to 3D graphics

Representation: This is (roughly) how an object is built up:

1. In 3D Graphics, an object is first defined by its edges (points) in a three dimensional x-y-z space.

2. When those points a are linked together by lines we get a *wireframe* rendering of an object.

3. After the frame has been created, a *surface* or skin is applied to the object. The surface can have many qualities: color, texture, shininess, reflectivity, etc.

4. Finally, objects are either lit or emit light. Most objects have been lit by a light source and must be shaded. Shading is the most computer intensive task (see section 1.1.5).

   All surfaces can be represented as a set of polygons (that are perfectly flat). Complex Polygons are always split up into triangles by the rendering machine.

   Polygons have only **one** side, the so-called "normal" or outside. Therefore, in VRML even flat objects (such as a sheet of paper) are always represented (as very flat) cubes. Note that a cube is composed of 12 polygons (2 triangles for each side) with their "normals" outside.

Position and Orientation: *Positions* in space are given in x-y-z coordinates:

- Width (x axis) or left(-) / right(+)
- Height (y axis) or down(-) / up(+)
- Depth (z axis) or far(-) / close (+)

You can picture the coordinate system with the "right hand rule": "x" is your thumb, "y" the index finger, and "z" the middle finger.

Try to remember the image in figure 1.2.

Orientation of an object is defined by "yaw", "pitch" and "roll" (Imagine what a ship can do):

- *Yaw* is left-right orientation. Imagine turning your head or your body around to look at something. "The ship can't hold its track."
- *Pitch* is backwards-forwards up/down orientation. "The ship goes straight into big waves."
- *Roll* is left-right up/down orientation. "The ship is hit on the side by big waves."

Therefore, in 3-D graphics there are **six degrees of freedom**: 3 positions (x,y,z) plus yaw (around the y axis), pitch (around the x axis) and roll (around the z axis)

[7] http://www.vrml.org/Specifications/
[8] http://tecfa.unige.ch/guides/vrml/vrml97/spec/

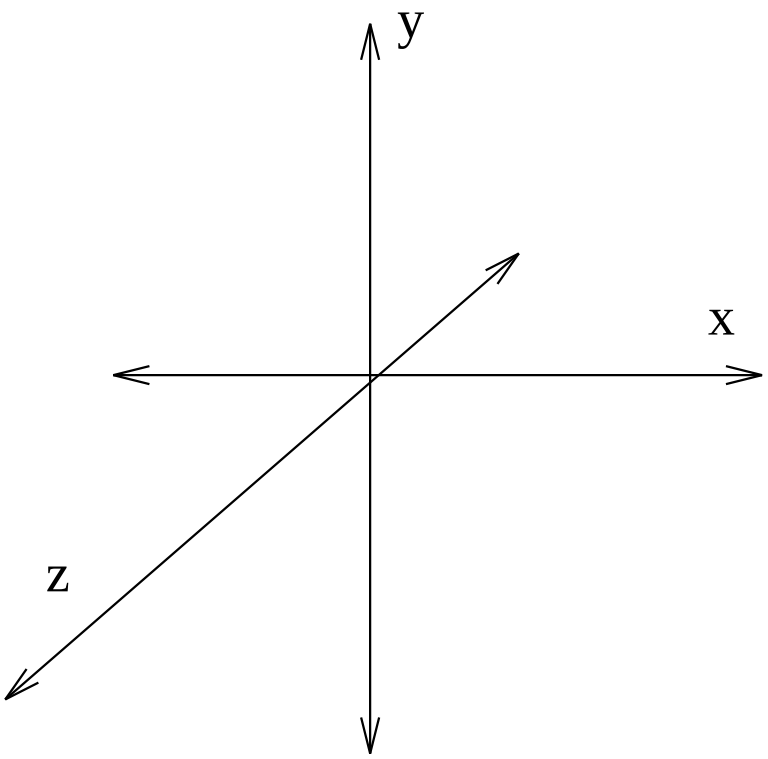

Figure 1.1: The 3 axis of 3-D graphics

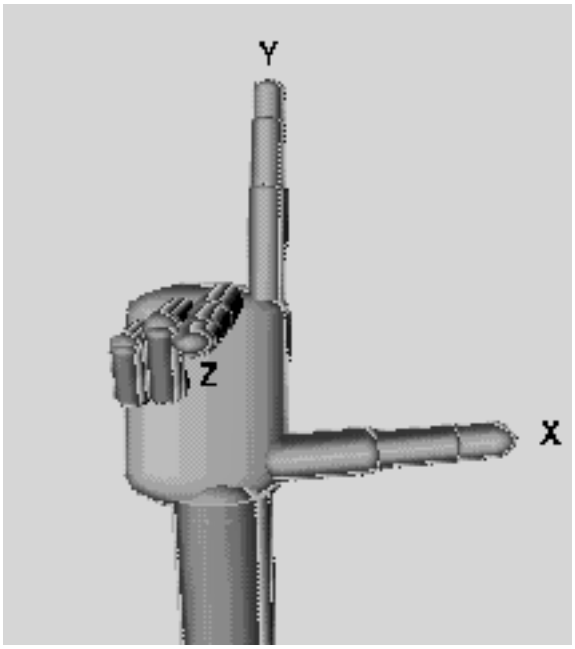

Figure 1.2: The right hand rule

Lights: There are different sorts of lights and by definition VRML browsers have some ambient light source:

1. Point Lights
   (a) Full point lights
   (b) Parallel point lights
2. Directional Lights have:
   - A location (x-y-z axis)
   - An orientation (yaw-pitch-roll)

   Spotlights have an umbra (i.e. a focus expressed in terms of width and how it widens).

1.1.4 Producing VRML

There are 3 (major) ways for producing VRML:

1. Code VRML by hand. If you like to hand code VRML (which is a good choice for learning how to build truly interactive worlds), consider getting a VRML assisting editor like emacs.
2. Use a VRML supporting Modeler. There are basically two types of tools you need:
   (a) A object creation tool
   (b) A space (or "walk thru") creation tool such as Cosmos' (ex-Paragraph's) Home Space Builder (the non-profit versions are cheap) or Virtus' Walkthrough Pro.
   - See e.g. the Tools section in our VRML Pointers[9].
3. Use a Filter to transform other 3D Formats into VRML.

1.1.5 Browsers and how to use them

Wich browser ? Right now (Feb 98) there are good enough VRML browsers for the Win/NT platforms and decent enough ones for Mac and SGI/Irix. See our VRML pointers[10] page for recommendations and section 8.3 on page 86 for a few hints about using and developping for current browsers.

Navigation: VRML browsers don't have all the same functionalities and they don't have the same interface. Have a look at the documentation, before you start feeling dizzy. The most common navigation means are:

1. Walk (6 degrees of freedom)
2. "Plane" Walk: restricted walk in in x-y or x-y-z axis, heads up/down in y-z axis, etc.
3. Flight: start/stop flying and accelerate/decelerate.
4. "Point at" (or "Seek"). Jumping to an object, or selecting on with a harpoon and get closer in several steps
5. "Examine" mode is special. It can be used to examine (rotate and zoom) either the whole scene or a selected object (or both). An example is Webspace's Examine Mode/ Rotation Globe.

Also, in our opinion browsers ideally should have a "map view" Builder, i.e. show the user where he is on a map (2D) or within wire-frame map (3D) of the whole scene.

[9] http://tecfa.unige.ch/guides/vrml/pointers.html#TOOLS
[10] http://tecfa.unige.ch/guides/vrml/pointers.html

Help I am Lost: Don't worry, most VRML scenes can bring you back to the entry point. Search for something like: "Restore Viewpoint", "Entry View" etc. under "View(s)" menus. Else, reload the page and next time, don't walk too fast !

Quality (Speed) of Viewing: Note: Details and names may be wrong !!!

Quality of rendering and speed have to with the way in which objects are rendered. "Wireframe" rendering is very fast, Phong Shading can be quite slow depending on the complexity of the object.

1. Wireframe (fastest)
2. Solid (Flat?) Shading: The whole object gets the same shading depending on its angle (fast).
3. Smooth (Gouraud) Shading (slower): Each polygon is shaded differently
4. Phong Shading (very slow): Polygons corners are shaded too.
5. Ray Tracing (impossibly slow on usual end-user's machines): Every point has a different shading.

Note that some browsers (e.g. VrWave) allow you to specify the display type both for viewing (camera stopped) and moving (you move or turn objects around).

1.2 Basic static VRML

When you play around with your own little VRML files, make sure they *really* get reloaded right after you change them. To make sure reload them holding down the SHIFT key on your machine when you click on reload in Netscape. **If you read this on paper**, note that URLs in the examples need to be completed by "http://tecfa.unige.ch/guides/vrml/"

1.2.1 Getting Started

The structure of a WRL File: VRML (*.wrl) files have 3 basic elements:

1. A header which tells the browser that the file is VRML and which version also. A header line is mandatory.
2. Comments are preceded by a `#`.
3. **Nodes**: Most everything else are nodes. Nodes generally contain:
   (a) The type of node (required). Nodes always are in Capital letters.
   (b) A set of curly braces `{.....}` (required)
   (c) A number of fields, all or some of which are optional. Note that there is no mandatory ordering of fields.
   (d) Fields with that can have multiple values require braces `[ ...]`. Fields always start with lowerCase letters.

Here is a typical VRML file with a single node (don't worry if you don't understand it):

```
# VRML V2.0 utf8
# A sample file with a single stupid node

Transform {
   translation 0 2 0
   children [
      Shape {
```

```
        geometry Sphere {}
     }
    ]
}
```

VRML II has about 9 major node types. We will shortly list the features of some important nodes as explained in detail in the Concepts[11] and the =>Node Reference[12] sections in the VRML 97 specification[13]. However, don't worry too much about the exact meaning of these features yet. The purpose of this list is just giving you an overview of some of "what's there". Once you master some basics you then should consider going back to reading the the Concepts Section of VRML 97 specification in whole !

1. Grouping nodes: =>Grouping nodes are used to create hierarchical transformation graphs. Grouping nodes have a children field that contains a list of nodes which are the transformation descendants of the group. Each grouping node defines a coordinate space for its children. This coordinate space is relative to the parent node's coordinate space–that is, transformations accumulate down the scene graph hierarchy

2. Special groups: The =>Inline is a grouping node that reads its children data from a location in the World Wide Web. The =>LOD specifies various levels of detail or complexity for a given object, and provides hints for browsers to automatically choose the appropriate version of the object based on the distance from the user. The =>Switch grouping node traverses zero or one of the nodes specified in the choice field

3. Common nodes: Used for things like light, sound, scripts, etc. See the =>Node Reference Chapter in the specs.

4. Sensors: =>Sensor nodes generate events. Geometric sensor nodes (ProximitySensor, VisibilitySensor, TouchSensor, CylinderSensor, PlaneSensor, SphereSensor and the Collision group) generate events based on user actions, such as a mouse click or navigating close to a particular object. TimeSensor nodes generate events as time passes.

5. Geometry nodes: =>Geometry nodes must be contained by a Shape node in order to be visible to the user. The Shape node contains exactly one geometry node in its geometry field.

6. Appearance: The Appearance node specifies the visual properties of geometry (see above) by defining the material and texture nodes.

7. Interpolators: =>Interpolator nodes are designed for linear keyframed animation.

8. Bindable Nodes: =>Bindable Nodes have the unique behavior that only one of each type can be active (i.e. affecting the user's experience) at any point in time.

VRML 1.0 contained 6 major (and 36 minor node) types, i.e:

1. Shape Nodes (such as cubes, text, sphere)

2. Geometry and Material Nodes (such as texture and font style)

3. Transformation Nodes (such as rotation and translation)

4. Camera Nodes

5. Lightning Nodes

[11] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html
[12] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html
[13] http://tecfa.unige.ch/guides/vrml/vrml97/spec/index.html

6. Group Nodes

Now be careful! **VRML is a caseSensitive Language**. Respect that or you will get strange errors!

VRML versions are easy to identify: Just look at the first line in the file You can translate most of old VRML I files with the vrml1to2 program[14] from Sony.

Example 1.2.1 *A simple VRML 2.0 file*

VRML:	*../examples/basics/basics-1.wrl*
Source:	*../examples/basics/basics-1.text*

You don't need to worry about details yet, it will just give you an idea of the structure of a VRML file

```
#VRML V2.0 utf8

#      ^^^^^  this is the MANDATORY header line
# The VRML Primer
# 1997 Daniel K. Schneider, TECFA
# No Copyright, steal this!
#

# Draw a blue cylinder - a first node specifying an object
# Radius and height can be floating numbers
Shape {
   geometry Cylinder {
      radius 0.1
      height 3.0
      }
   appearance Appearance {
      material Material { diffuseColor 0 0 1  }
   }
}

Transform {
   # Move the pen up - a second node specifying a translation
   # and a sphere
   translation 0 2 0
   children [
      Shape {
         geometry Sphere {}
      }
     ]
}
```

Ordering is quite simple: VRML is a declarative language and all declaration are executed in the order they appear. Basically speaking, VRML goes "do this", "do this", etc. However, VRML nodes can be nested at any level of detail (and **should** be as you will learn later). The usual **scoping** principles apply. E.g. if you insert a Transform node within a Transform node (you will see that later) transformations applied in the child node are applied within or before the transformations in the parent node. E.g. within a low level node you can rotate some objects and then place those objects somewhere else with a higher level node.

Now all you need is a VRML manual and you can start producing VRML files or write programs that generate VRML. Or can you continue reading this for a little while.

[14] http://tecfa.unige.ch/guides/vrml/vrml1to2E.html

1.2.2 Basic Geometry (Shapes)

Per definition, all basic predefined shapes in VRML **are drawn around the origin** and have **a standard size**. Size is measured in units (roughly meters in reference to the real world). E.g. a cube has a standard size of 2 by 2 by 2 meters. Note that sizes are given **with floating point numbers**, e.g. 2.0 or .01 or 0.2. **Degrees** are given in *radians*.

Table 1.1: Degrees to radians conversion

30:	0.5236
60:	1.0472
90:	1.5708
180:	3.1416
270:	4.7124

You can also use our Javascript Radian Converver

Some objects (i.e. Cylinders and Cones) have named parts which you can avoid drawing (e.g. the bottom of a cylinder). Per default, all parts are drawn.

Cubes: You can look now at this most simple cube[15].

```
#VRML V2.0 utf8
#A simple box (was called cube in VRML I)

Shape {
        geometry Box {}
}
```

The box (cube) above might appear as a flat rectangle in your browser, turn it around to be sure it's a 3D object. Also it does not have any color so it will appear as without any shading (in most or all browsers). If you can't see it in your browser skip this example. Per default, all objects are drawn **off the center** of the current origin **in all directions**. E.g. in the above case we draw around the [0 0 0] point: 1 left/right, 1 up/down and 1 forward/backwards leading to a 2 by 2 cube. Just remember this right now: *You don't draw towards right, up and forward, but around the origin, i.e. left-to-right, down-to-up, backwards-to-forwards of the origin (the current coordinates of the "pen").*

Example 1.2.2 *A simple floor*

VRML:	../examples/basics/basics-shapes-2.wrl
Source:	../examples/basics/basics-shapes-2.text
VRML Full:	../examples/basics/basics-shapes-2V.wrl
Source Full:	../examples/basics/basics-shapes-2V.text

The box here represents something like a floor. If you can't see it in your browser, click on the "Full" version. As you will learn later, you have to define appearance and position the user someplace he can see something when the scene loads.

```
#VRML V2.0 utf8
#A floor, i.e. a thin and large box

Shape {
```

[15] ../examples/basics/basics-shapes-1.wrl

```
        geometry Box {
                size 20 0.1 30
        }
}
```

Spheres: or balls:

```
#VRML V2.0 utf8
#A large sphere
      Shape {
         geometry Sphere {
               radius 2}
      }
```

Cylinders: Cylinders are more complex. Here is a simple and a more complex example.

```
#VRML V2.0 utf8
#A long stick
Shape {
   geometry Cylinder {
      radius 0.1
      height 3.0
   }
```

Example 1.2.3 *A simple cup*

VRML:	../examples/basics/basics-shapes-5.wrl
Source:	../examples/basics/basics-shapes-5.text
VRML Full:	../examples/basics/basics-shapes-5V.wrl
Source Full:	../examples/basics/basics-shapes-5V.text

In the cup example we tell VRML which parts we need. By definition, VRML renders all parts of an object. If you want to display just some of them you must specify which ones like in the following example. Note: this replaces the parts field in VRML 1 that took as arguments a list of parts in parenthesis separated by vertical bars, i.e. `(...|...|...)`.

```
#VRML V2.0 utf8
#A round large cup
Shape {
   geometry Cylinder{
      height 0.5
      radius 0.5
      bottom TRUE
      side TRUE
      top FALSE
   }
}
```

Also note that you wouldn't even display a simple cup this way. You'd have to add some color and position either the cup or the default view somewhere else (you will learn that later).

Cones have a height and bottomRadius field and "Sides" and "Bottom" parts

1.2.3 Positioning Shapes

In order to position objects in a VRML scene (unless you want to move the user's viewpoint) you have to use the =>Transform[16] Node. We will introduce some of its features here.

Example 1.2.4 *A simple translation*

VRML:	../examples/basics/basics-1.wrl
Source:	../examples/basics/basics-1.text

Let's have a look again at the first VRML scene shown in this tutorial (and *ignore* the contents of the appearance field right now). The Sphere is placed on top of a stick (the cylinder) because we told VRML to move the pen 2.5 up on the y axis. Since the stick is drawn from -1.5 to + 1.5 along the y axis we draw the Sphere (which has a radius of 1) starting with an origin of 1 further up the y axis.

```
#VRML V2.0 utf8
# The VRML Primer
# 1997 Daniel K. Schneider, TECFA
# No Copyright, steal this!
#

# Draw a blue cylinder - a first node specifying an object
# Watch out: radius and height are floating numbers !
Shape {
   geometry Cylinder {
      radius 0.1
      height 3.0}
   appearance Appearance {
      material Material { diffuseColor 0.1 0.1 0.9 }
   }
}

Transform {
   # Move the pen up - a second node specifying a translation
   # and a red sphere
   translation 0 2.5 0
   children [
      Shape {
         geometry Sphere { radius 1 }
         appearance Appearance {
            material Material { diffuseColor 1 0 0 }
         }
      }
     ]
}
```

To move to another point in the *same frame of reference* the **Transform** node has a **translation field** which allows to move the Pen (e.g. the following objects in the group) in some x-y-z direction.

Within a Transform node, objects can be embedded (combined) and addressed as a whole. Concretly, this means that a translation will affect all the renderings within the same Transform node) This is why you must use the "children" field to indicate the nodes which are affected by a transformation. (You will see more of that later).

You can also draw overlapping shapes like in the following example:

[16] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Transform

Example 1.2.5 *Overlapping objects*

VRML:	../examples/basics/basics-shapes-10.wrl
Source:	../examples/basics/basics-shapes-10.text
Full VRML:	../examples/basics/basics-shapes-10V.wrl
Full Source:	../examples/basics/basics-shapes-10V.text

```
#VRML V2.0 utf8
# Overlapping shapes

Shape {
        geometry Box {
                size 3 0.1 0.1
        }
}
Shape {
        geometry Box {
                size 0.1 3 0.1
        }
}
Shape {
        geometry Box {
                size 0.1 0.1 3
        }
}
Shape {
        geometry Sphere { radius 0.5 }
}
```

Note that rendering overlapping node is less efficient than drawing ajacent nodes. E.g. if you want to draw a forest with some balls on top of sticks make sure that the ball does not overlap the stick (altough the visible result would be the same). Also, if you think that this object looks too dull on the screen, apply some Material like we did in the Full VRML[17] scene. We will look into this in section 1.2.5).

1.2.4 Rotations

Rotations can be around the x, y and z axis (yaw, roll and pitch). The rotation field takes 4 arguments: The axis and the degree in radians. A 180 degree rotation is 3.14159 (Pi). Here is a simple example of a rotation around the z axis:

Example 1.2.6 *Simple Rotation example*

VRML:	../examples/inter/inter-rotation-1.wrl
Source:	../examples/inter/inter-rotation-1.text
Full VRML:	../examples/inter/inter-rotation-1V.wrl
Full Source:	../examples/inter/inter-rotation-1V.text

The cone is rotated 3.14 radians around z by giving the following arguments to the rotation field of the Transform node.

```
        rotation 0 0 1  3.14
```

Note that the Boxes represent the x,y,z axis.

[17] ../examples/basics/basics-shapes-10V.wrl

```
#VRML V2.0 utf8

Shape {
        geometry Box {
                size 0.1 10 0.1
        }
}
Shape {
        geometry Box {
                size 10 0.1 0.1
        }
}
Shape {
        geometry Box {
                size 0.1 0.1 10
        }
}
Transform {
        rotation 0 0 1  3.14
        children [
                Shape {
                        geometry Cone {}
                }
        ]
}
```

Rotation is not that intuitive, but if you play around with a few examples you might find the grasp for it. When you define a rotation you first of all define the **angle** of rotation in radians (the 4th number in the field). Then you specify around which axis to rotate. Imagine that the the three axis parameters define a **diagonal line** within a cube. Rotation will happen around this diagonal ! [drawing needed here]. Rotation has the same effects as translation. It "reorients" the pen like any other "transform".

PlayTime: You can intuitively learn more about size, translation and rotation by playing with the Tiny 3D Applet/Java1.02[18]. Note that you need a EAI capable browser (most will do). If your browser understands Java 1.1 widgets (e.g. patched Netscape 4) you can try Tiny 3D Applet/Java1.1[19].

1.2.5 Defining appearance

Usually you **must** use an =>Appearance[20] node when drawing objects. Else they will appear blank, or white, that is dull (or if your headlights are turned off you won't even see a thing at all. Here you will learn how to use simple colors and how to "dress" objects with bitmap files.

Simple Materials Here is an simple example representing Earth and Sun. The Sun is behind to the right. If you can't see it, put your client in flip mode and turn the scene around. You will learn later how to define correct viewpoints. (Well if you are in a hurry look at the Full Vrml example).

Example 1.2.7 *Simple Materials*

[18] ../examples/eai/tiny3D-2-java1.0.2/tiny3D-2.html
[19] ../examples/eai/tiny3D-2/tiny3D-2.html
[20] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Appearance

VRML:	../examples/basics/basics-materials-1.wrl
Source:	../examples/basics/basics-materials-1.text
Full VRML:	../examples/basics/basics-materials-1V.wrl
Full Source:	../examples/basics/basics-materials-1V.text

```
#VRML V2.0 utf8
# Different materials for Earth and Sun
# Sun turns around Earth :)

# The Earth in blue
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 0 0 1
      }
   }
   geometry Sphere {
      radius 1
   }
}

#The Sun yellow and shining
Transform {
   translation 2 1 -2
   children [
      Shape {
         appearance Appearance {
            material Material {
               emissiveColor 1 1 0
               shininess 1
            }
         }
         geometry Sphere {
            radius 1
         }
      }
     ]
}
```

What is new here are the appearance and the material node. =>Appearance[21] nodes define the visual properties (skin) of a polygon. The =>material[22] nodes specify surface material properties for associated geometry nodes and are used by the VRML lighting equations during rendering. You can select from several types of color. Colors themselves are given as three RGB numbers in the range of [0,1]. E.g. "diffuseColor 0 0 1" gives a diffuse blue.

VRML's lightning model[23] is quite complex and we will only make a few short comments about the fields of the Material node here.

```
Material {
  exposedField SFFloat ambientIntensity  0.2
  exposedField SFColor diffuseColor      0.8 0.8 0.8
  exposedField SFColor emissiveColor     0 0 0
  exposedField SFFloat shininess         0.2
```

[21] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Appearance
[22] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Material
[23] /guides/vrml/vrml97/spec/part1/concepts.html#4.14

```
   exposedField SFColor specularColor     0 0 0
   exposedField SFFloat transparency      0
}
```

Those fields define both the color of an objects and the way it reflects light from the defined light sources. But please note that lightning effects are also (and mainly) function of the lights you put into the scene.

- The diffuseColor field is the most commonly used. It will (like most real world objects) reflect light depending on the angle of the surface. The object appears brighter (more lit) when its surface is directly exposed to light as you would expect.
- The *emissiveColor* field defines "glowing objects". E.g. you would use this to build a visible lamp.
- The *specularColor* field defines the color extra reflection has when the angle from the light is close to the angle you are looking at. It is used together with the *shininess* field. Mostly you would use white color. You can experiment this effect in real life by holding a (new apple) or a photograph between you and a window (or a lamp).
- The *transparency* field specifies how much light is transmitted through the object with 1.0 being completely transparent, and 0.0 completely opaque.

The Color Applet: You should now play a little bit with this applet. It will give you a good feeling for VRML colors under simple lightning conditions. In addition it will generate a complete VRML scene for your own usage.

Color Appplet:	eai/color/

Image Textures There are various of ways of using textures in an =>Appearance[24] node. Here, we will see how to use =>ImageTexture[25] node. VRML 2 browsers are required to support JPEG and PNG file formats. GIF support is only an option, so use JPEGs or PNGs if ever possible !

Here is a very simple node that dresses a cube with a bitmap file.

```
Shape {
   appearance Appearance {
      texture ImageTexture {
         url "tecfa/mendelsohn.gif"
      }
   }
   geometry Box { }
}
```

Per default each side of an object will be painted with an image (e.g. cubes will have 6 images, spheres one, and cylinders two).

Here is a small VRML file that shows some Tecfa people using a few few basics geometric shapes.

Example 1.2.8 *Image texture example*

VRML:	../examples/basics/basics-persons-1.wrl
Source:	../examples/basics/basics-persons-1.text

Note how you can repeat textures easily by using a =>TextureTransform[26]. The "floor" in the example above has been created using this technique. The *scale* field tells how many times to repeat a bitmap in the **s** and **t** direction. (Note: s and t is more or less like a standard x, y of a plane). Look at the code below:

[24] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Appearance
[25] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#ImageTexture
[26] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#TextureTransform

```
Transform {
   translation -4 -2 2
   children [
      Shape {
         appearance Appearance {
            texture ImageTexture {
               repeatS TRUE
               repeatT TRUE
               url "tecfa/sylvere.gif"
            }
            textureTransform TextureTransform {
               scale 10 10
            }
         }
         geometry Box {size 10 0.1 10 }
      }
     ]
}
```

There is much more to texturing with Bitmaps, e.g. you can combine colors and textures (read the specifications or another tutorial).

1.2.6 Anchors and Teleportals

Anchors =>Anchors[27] are group nodes, i.e. they have to be wrapped around the object(s) that represent the anchor. Here is an example that leads to our WWW home page if you click on the Sphere.

Example 1.2.9 *Anchors*

VRML:	../examples/basics/basics-anchors-1.wrl
Source:	../examples/basics/basics-anchors-1.text

```
#VRML V2.0 utf8

Transform {
   rotation 1 0 0 0.4
   children Shape {
      appearance Appearance {
         material Material {
            diffuseColor 0.2 0.1 0.6
         } }
      geometry Box {
         size 20 0.1 10
      }  } }

Anchor {
   url         "http://tecfa.unige.ch/"
   description "A link to the Tecfa Home page"
   children [
      Shape {
         appearance Appearance {
```

[27] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Anchor

```
            material Material {
               diffuseColor 0 0 1
            }
         }
         geometry Sphere {}
      }
     ]
}
```

Teleportals Teleportals are doors into other VRML worlds (scenes). They are done with WWW Anchors. Split up your scene into several ones when rendering, downloading, memory usage, etc. grows too high. But don't, if you want to avoid loading when people walk frequently back and forth.

1.2.7 Text

This example draws 2 texts near the origin. Per default, text is aligned left and bottom (i.e. without translation starts at 0 0 0.

Example 1.2.10 *Simple Text*

VRML:	../examples/basics/basics-text-2.wrl
Source:	../examples/basics/basics-text-2.text

```
#VRML V2.0 utf8

Shape {
   geometry Text {
      string "Le chemin vers l'enfer"
      fontStyle FontStyle {
         size 2
      }   }
}

Transform {
   translation 0 1.5 0
   children [
      Shape {
         appearance Appearance {
            material Material {
               diffuseColor 1 1 0
            }
         }
         geometry Text {
            string [
                    "Le chemin vers",
                    "l'autre enfer"
                   ]
            fontStyle FontStyle {
               size 2
            }    }       }
     ]
}
```

1.2.8 Exercises

Exercice 1.2.1 *Play with the Tiny-3d-2 Applet.*

It will allow you to create simple VRML geometry, color, size and move around it. It also allows you to generate VRML code that you can paste into your editor.

Java 1.1.x (Netscape 4x PATCHED):	eai/tiny3D-2/tiny3D-2.html
Java 1.0.x (Netscape 3x,4x):	eai/tiny3D-2-java1.0.2/tiny3D-2.html

Exercice 1.2.2 *You now might consider building a 3-D information page that points to HTML files.*

You can look at some (simple) pages produced by our students[28].

1.2.9 Moving On

You should at least glance at the next chapter (2) in order to know how you can reuse nodes, build your own nodes, inline other scences etc.

In any case, you then should read chapter 3 on page 35. Section 3.1 is absolutly *mandatory* reading.

If you think that you deserve a break go and check out the worlds at SGI's VRML gallery which can be found at http://vrml.sgi.com/worlds/.

[28] http://tecfa.unige.ch/vrml/tecfa-index.html

Chapter 2

Modularization and Abstraction

Before you start building any more serious example, you have to learn about DEF (naming objects so that you can reuse them) and PROTO (building your own node type).

2.1 Multiple instances of the same object

An important feature of VRML (and any "programming language") is being able to reuse an object by giving it a name and making copies of it. In the example below you can see that each node has been named with the **DEF** node. Most nodes can be named, not just geometry nodes! Even if you don't plan reusing objects, it is good practise to name at least major nodes because it implictly adds documentation to your file and you never know if later you want to reuse an object in bigger scenes and/or modify it's contents with a script. An example of "names everywhere" is below:

```
#VRML V2.0 utf8
DEF ROOTNODE Transform {
     children [
         DEF SOL Transform {
             children Shape {
                 appearance Appearance {
                     material Material {
                         diffuseColor 1 1 0
                         shininess 1
                       }
                   }
                 geometry Sphere {
                     radius 5
                   }
               }
           }
         DEF TERRA Transform {
             translation 10 10 10
             children [
                 DEF EARTH Transform {
                     children Shape {
                         appearance Appearance {
                             material Material {
                                 diffuseColor 0 1 0
                               }
                           }
```

```
                        geometry Sphere {
                          }
                      }
                  }
                  DEF LUNA Transform {
                      translation 1 1 1
                      children Shape {
                          geometry Sphere {
                              radius 0.5
                            }
                        }
                    }
                ]
            }
          DEF JUPITER Transform {
              children [
                  # Jupiter and it's 14 moons in here
                ]
            }
      ]
  }
```

A good and slightly more complex example is the solar system, look at the source form Pesce's book (translated into VRML II). In addition it has anchors to other URLs (see section 1.2.6).

Now let's see how we can reuse a named object. This example shows for instance how to do the spikes of a wheel (taken from [Ames et al., 1996b, page 90]).

Example 2.1.1 *Rotation: Spikes of a Wheel*

VRML:	../examples/inter/inter-rotation-2.wrl
Source:	../examples/inter/inter-rotation-2.text

Note the usage of **DEF** and **USE** which will allow you to use multiple instances of the same Object. In order to better understand what (where) we have drawn we also *inlined* a drawing of the x (red color), y (green) and z (blue) axis.

You can make use of the coordinates.wrl[1] file yourself if you are not sure what your translations and rotations do (inlining is explained in section 2.2).

```
#VRML V2.0 utf8

# Draws 6 spikes using 3 cylinders
# we give a name to our first cylinder so that we can reuse it again
# Check the DEF and USE commands

DEF AXE Shape {
   geometry Cylinder {
      radius 0.5
      height 5
   }
}
Transform {
   rotation 0 0 1  1.0472
   children USE AXE
```

[1] ../common/coordinates.wrl

```
}
Transform {
   rotation 0 0 1  2.0944
   children USE AXE
}

# Import the coordinates axes (ignore the code right now)
Inline {
   url "coordinates.wrl"
   bboxSize 10 10 10
   bboxCenter 0 0 0
}
```

The next example translates and scales this wheel, i.e. it squeezes it together (half size on y and z axis and a quarter size on the x axis).

Example 2.1.2 *Rotation: Spikes of a Wheel II*

VRML:	../examples/inter/inter-rotation-3.wrl
Source:	../examples/inter/inter-rotation-3.text

```
#VRML V2.0 utf8

Transform {
   children [
      DEF TUBE Shape {
         appearance Appearance {
            material Material {
               diffuseColor 0.6 0.3 0.1
            } }
         geometry Cylinder {
            radius 0.5
            height 5
         }
      }
      Transform {
         rotation 0 0 1  1.0472
         children USE TUBE
      }
      Transform {
         rotation 0 0 1  2.0944
         children USE TUBE
      }
     ]
}
Transform {
   translation 0 5 0
   scale       0.25 0.5 0.5
   children USE TUBE
}
Transform {
   translation 0 5 0
   scale       0.25 0.5 0.5
   rotation    0 0 1  1.0472
```

```
   children USE TUBE
}
Transform {
   translation 0 5 0
   scale       0.25 0.5 0.5
   rotation    0 0 1  2.0944
   children USE TUBE
}

# Import the coordinates axes (ignore the code)
Inline {
   url "coordinates.wrl"
   bboxSize 10 10 10
   bboxCenter 0 0 0
}
```

2.2 Inlining other VRML files

VRML code can be inlined. The **Inline** node has three fields:

1. The **name** defines the URL
2. The **bboxSize** gives the size of the scene to inline. It defines the *bounding box*, i.e. a rectangle that encloses all the geometry of the scene to be loaded. It's purpose to optimize loading. By default the size value is (-1,-1,-1) and must be calculated by the browser.
3. The **bboxCenter** field defines the center of the bounding box, i.e. **where to insert the objects**.

Here is a simple example that shows how to reuse several times the same object defined in an other file.

Example 2.2.1 *Inlining other VRML files*

VRML:	../examples/inter/inter-inline-2.wrl
Source:	../examples/inter/inter-inline-2.text

Alternatively you can also look at the source code of the naming/rotation examples in section 2.1 above which imports an x, y and z axis.

```
#VRML V2.0 utf8

Inline {
   url      "inter-inline-1.wrl"
   bboxSize 0.5 2.5 0.5
}
Transform {
   translation 3 0 0
   children [
      Inline {
         url      "inter-inline-1.wrl"
         bboxSize 0.5 2.5 0.5
      }
     ]
}
```

Note: insted of a translation one could use the bboxCenter field.

You can also look at our version of the Castle Example created by Mike McCue at Paper Software, Inc. (Note that this example has disappeared from the Web, if you know where the original is, please mail me.)

Example 2.2.2 *The Castle Example*

VRML:	../examples/castle/castle.wrl

In addition it shows how to define multiple cameras and how to use GIFs for texturing. Note, that this example needs some polishing. In order to look at the files just cd to the ../examples/castle/castle.wrl directory and click on the *.text files. They are identical with the *.wrl files.

2.3 Prototyping

Once you build more complex objects you want to reuse smaller elements in different ways. So, learning how to define =>PROTO node[2] is important before you really start to produce a lot of VRML code. PROTOs define a new **node name** with custom **fields** you can specify. In other words, PROTOs allow you to *extend the language* by adding your own nodes. In this respect, PROTOs are much more powerful than simple DEFs/USEs that only allow to apply transformations to reused objects.

The structure of a **simple** PROTO is the following:

```
PROTO prototypename [ field fieldtypename name defaultValue
                      field fieldtypename name defaultValue
                      ... ]
                    {node}
```

Let's look shortly at the definition of fields. Here is an example taken from below:

```
field SFColor legColor  .8 .4 .7
```

field is just a keyword you must use. *SFColor* is a valid VRML "field type name" (see section 8.4 on page 87. *legColor* is the name that the field will have and ".8 .4 .7" are it's default values. If this is unclear, go through the next two examples and see how they are used.

Example 2.3.1 *Three Stools*

VRML:	../examples/inter/inter-proto-1.wrl
Source:	../examples/inter/inter-proto-1.text

Here is an example of three stools. It is taken from the =>VRML II Specification[3].

It defines a "TwoColorStool" node with two public fields named "legColor" and "seatColor" that one can modify when making instances. Note that there are default colors (pink legs and a yellow seat)

```
#VRML V2.0 utf8

PROTO TwoColorStool [ field SFColor legColor  .8 .4 .7
                      field SFColor seatColor .6 .6 .1 ]
{
   Transform {
      children [
         Transform {   # stool seat
```

[2] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#4.8
[3] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/examples.html

```
            translation 0 0.6 0
            children
            Shape {
               appearance Appearance {
                  material Material { diffuseColor IS seatColor }
               }
               geometry Box { size 1.2 0.2 1.2 }
            }
         }

         Transform {   # first stool leg
            translation -.5 0 -.5
            children
            DEF Leg Shape {
               appearance Appearance {
                  material Material { diffuseColor IS legColor }
               }
               geometry Cylinder { height 1 radius .1 }
            }
         }
         Transform {   # another stool leg
            translation .5 0 -.5
            children USE Leg
         }
         Transform {   # another stool leg
            translation -.5 0 .5
            children USE Leg
         }
         Transform {   # another stool leg
            translation .5 0 .5
            children USE Leg
         }
        ] # End of root Transform's children
   } # End of root Transform
} # End of prototype

# The prototype is now defined. Although it contains a number of nodes,
# only the legColor and seatColor fields are public.

# Instead of using the default legColor and seatColor,
# this instance of the stool has red legs and a green seat:

TwoColorStool {
   legColor 1 0 0 seatColor 0 1 0
}

# The chair to the right has yellow legs and a bright blue seat:

Transform {
   translation 2 0 0
   children [
      TwoColorStool {
         legColor 1 1 0 seatColor 0 1 1
      }
```

```
      ]
}

# The chair to the left uses the default colors

Transform {
   translation -2 0 0
   children [
      TwoColorStool {
      }
     ]
}

NavigationInfo { type "EXAMINE"}      # Use the Examine viewer
```

You can look at an another (locally produced) example that uses embedded protos: Patrick[4] Jermann's MOO Ants.

Example 2.3.2 *Moo Ants PROTO Example*

VRML:	../examples/inter/inter-proto-ant.wrl
Source:	../examples/inter/inter-proto-ant.text

This example uses PROTOS within PROTOS, a facility which can add additional modularity to your VRML worlds.

2.3.1 Exercises

Exercice 2.3.1 *A forest without Snow White*

Build a little forest on a flat plane

- Build a proto made of a cylinder (the trunk) and a cone
- Build a proto made of a cylinder and a ball
- Both protos should allow you to define their position and their color.

[4] http://tecfa.unige.ch/%7Ejermann/

Chapter 3

Good VRML Style and Performance

In this chapter you will learn (a little bit) about good VRML style and performance. You should at least consult the following section that will give you a checklist of minimal requirements that are necessary in a VRML you will put up for others to look at.

3.1 Your first "real" VRML file

So far you have learned that VRML is a text file format for describing 3-D shapes and interactive environments on the WEB. It basically contains a set of **Nodes** that contain fields and values. As the examples shown so far demonstrate, you could do "interesting" VRMLized web "pages", but you need to learn a few more things.

Navigation Information VRML allows you to specify Navigation Information with the =>NavigationInfo[1] Node. Currently most VRML 2.0 browsers will support the **type** field which allows you to specify the navigation paradigm to use. You can choose among "WALK", "EXAMINE", "FLY" and "NONE". In the following example we tell the browser to be in "examine mode", which is useful for displaying 3-D data or single objects.

```
NavigationInfo { type "EXAMINE"}      # Use the Examine viewer
```

Note: be careful with this. Some browsers (e.g. Community Place) will not allow the user to switch modes once you defined one in your file. E.g. when you put the browser in "examination" (or flip) mode, the user won't be able to turn back into "walking" mode. In this case, use:

```
NavigationInfo { type [ "EXAMINE", "ANY" ] }  # Examine viewer but allow change
```

Viewpoints Looking at some examples you wrote, you probably noticed that your "scene" (or yourself) is not at the "right place".

The most simple thing you can do is to define a single default =>Viewpoint[2] that places the user at the right distance from the object. Per default, browsers are supposed to put you at (0 0 10) which is fine for viewing small objects. In the following example we place the viewpoint 12 "meters" in front, 3 "meters" up, and 2 "meters" to the right.

```
Viewpoint { position 2 3 12 }
```

[1] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#NavigationInfo
[2] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Viewpoint

The Viewpoint node is more complex than that. You can also give an orientation to the view ("look into a direction"). It works the same way as rotation does (see section 1.2.4).

Once you have defined a few viewpoints (and DEFed them with a name), it is important to fill in the description field if you want people to be able to use the "viewpoints" menu in their browser. Here is an example:

```
DEF SunView Viewpoint {
   position 2 0 -10
   description "VERY close to the SUN"
   orientation 1 0 0 0.4}
```

Next, you can use viewpoints to organize "tour guides". The next example shows how to organize a simple guided tour.

Example 3.1.1 *A simple ugly tour guide*

VRML:	../examples/inter/inter-tour-guide.wrl
Source:	../examples/inter/inter-tour-guide.text

In particular, look at the following details:

- how we use the different Viewpoint fields (position, description, jump, orientation and fieldofView)
- The anchor construct is used to jump around. Each DEFed Viewpoint can be accessed with "internal #xxx" tags. Viewpoints are the VRML equivalent of HTML "name" tags.

Information about your World Instead of putting information about your world in comments at the beginning of the file, you better should use the =>WorldInfo[3] node. Here is an example:

```
WorldInfo {
  info  ["A sample home page", "steal it if you like", "Version 1 - Feb 97"]
  title "Ka's kewl home page"
}
```

The browser may display the contents of the title field!

Finally: Don't forget to color your objects, without color one can hardly make out anything.

3.2 Levels of Details

In order to make scenes more efficiently rendered you can display the same object at different levels of detail with the =>LOD node[4]. The principle is very simple: If you look at an object from a certain distance it does not appear the same, e.g. a house can be just a cube. So instead of building up the full house as soon as the wrl file is loaded we just display a simple cube for this house (if and when the "user is far away"). Only when he get's closer we start rendering the object in fuller details.

The structure of a LOD with three levels of details must be something like the following example. In the "center" field you have to specify the center of your objects (which is at 0 0 0 in this case). The "range field" tells the browser what level of detail to display in function of the distance of the camera (your point of view). The first number in the list refers to the first level (the most detailed one), the next to next detail level and so on.

[3] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#WorldInfo
[4] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#LOD

```
LOD {
    center      0.0 0.0 0.0
    range       [10, 15]
    level       [
                 # LEVEL III: Detail
                 Inline {
                    url "inter-persons-2.wrl"
                 }

                 # LEVEL II: The bunch as a whole
                 Anchor {
                       ..... }

                 # LEVEL III: ....
                 Shape { .... }
```

Note, that you completely can hide away an object by specifying an empty node as the last level with something like:

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

Also, remember that inlined worlds are only loaded on need (at least in Cosmo Win Beta3a). Both features are useful for building larger scenes.

This example shows Tecfa People at three levels of details: (1) as an uninteresting box if you look at it from far away, (2) as a group and as (3) individuals if you get closer. Just walk FORWARD to see the effect !

Example 3.2.1 *Tecfa People LOD*

VRML:	../examples/inter/inter-LOD-1.wrl
Source:	../examples/inter/inter-LOD-1.wrl

Of course, this example needs some tuning. First of all, distances should be bigger to have a smoother "getting closer" effect and most of all different levels of detail ought to have more similar shapes and appearance. For a better example see page 52 in ([Hartman et Wernecke, 1996] or visit directly the Tenochtitlan Web Site[5].

But as the above examples shows, LOD nodes can be used not just levels of details, but different functionality, e.g. the mid-level of detail is an anchor to a group WWW page of people, whereas at the detail level you get anchors to some person's home pages.

3.3 Good VRML style and performance

Now you should have most elements for writing simple VRML scenes. For the sake of mastering complexity (and also efficiency) it is important to master the following techniques:

1. Grouping: A lot of nodes do =>Grouping. Grouping so far encountered was mostly done with the =>Transform node. If you just want to put a series of objects into a group (without any transformation, just use the =>Group node. It is important to group objects together that can be viewed from the major angles that users are likely to use (culling effect). This way your browser won't need to render hidden objects.

2. Naming: Each object can be given a name with "DEF". In this case it can reused with "USE" (see =>Instancing) and look again at the examples in section 2.1 on page 27.

[5] http://vrml.sgi.com/handbook/MasterLayoutPC.html

3. Once your objects get more complex and you want to parameterize you should master the =>PROTO node. Note that you can do much more than we have introduced in this section above.

4. Efficiency is major concern (at least today). Grouping and using instances (DEFs) already will improve the rendering of a scene. In addition when building more complex scenes make use of the following techniques:

   - =>LOD nodes. See section 3.2 again if needed.
   - Overlapping: Drawing objects that overlap are more expensive to render than adjacent objects. At least when creating complex objects, remember this. Per default, all objects are drawn **off the center** of the current origin **in all directions**. E.g. if we draw a simple cube around the [0 0 0] point: We draw 1 to the left and 1 to the right, 1 up/down and 1 forward/backwards leading to a 2 by 2 cube.
   - Use =>Inlines (preferrably with bounding boxes). This will allow people to start navigating while details are still being loaded. It works like the "IMG height=xx width=yy" tag in HTML.
   - For people who don't like to "waste" their time with navigation you might want to build tour guides as shown above. Note that later you will learn how to make "jumps" more interesting with scripting techniques.

Further reading: Chapter 10 of [Hartman et Wernecke, 1996].

Chapter 4

Mixing HTML with VRML

In this chapter you will learn a few more static VRML tricks. Alternatively you can start learning some interactive VRML and directly go to chapter 5 on page 49.

4.1 Introduction

There exist several kinds of possible links between HTML and VRML. The most frequently used are the following:

HTML-VRML links:

- `<A HREF="../examples/basics/basics-shapes-10V.wrl">My world</A>` will replace your html page by a VRML scene. (a VRML page)
- `<A HREF="../examples/basics/basics-shapes-10V.wrl TARGET="_blank">My world</A>` will open a *new browser window*. This is recommended way to bring (non-embedded) VRML scenes things to newbie Internet users. Others should know how to open another browser window.
- You can also combine HTML and VRML in various frames. See the next section for details.

Embeding a VRML browser in a HTML Page: Use the `<embed> tag`, e.g. something like

```
<EMBED align=center
 src="../examples/basics/basics-shapes-10V.wrl" border=0
 width=180 height=180>
```

in order to embed a VRML plug-in window *within* a HTML page. If you don't need to display large VRML scenes, this is the recommended way to do it (rather then using frames).

VRML-HTML links: See section 1.2.6 on page 24. Note that can you pass parameters like `target=` if you want for example the user to open up another HTML frame when clicking on a VRML object.

You can also use the Browser object in a Script node (make sure to have a recent browser !) for more sophisticated stuff:

```
Browser.loadURL(url, parameter);
          e.g.
Browser.loadURL("publicity.html", "target=_new");
```

If you want to launch a new window, use '`target=_new`' for the parameter. If you want to target a frame named explanation, use 'target=explanation' for the parameter.

JAVA, Javascript, VRML, HTML links: Of course you can do more crazy stuff (more to follow maybe one day) but just be aware that "Javascript" inside VRML is NOT Netscape's (or whatever) language. see section 5.3 on page 58 for some more information.

4.2 Mixing HTML and VRML Frames

By using frames you can create interesting mixed VRML/HTML documents like the Tenochtitlan Web Site[1] we already mentioned. However make sure that a simple `<embed>ed` VRML browser window won't do the trick, because simple HTML pages are easier to bookmark. [Note: we have to look into combining the `<embed>` tag with frames].

You can learn about frames in several places, for Netscape you can check out the THE Netscape Frames Tutorial[2]. Here we just present two simple examples.

4.2.1 VRML scenes library

Image that you'd like to build a libary of the VRML scenes you built. You would like to (a) display your files, (b) let the user display the scene in a different frame and (c) and let him look at the code in a still different frame.

```
*******************************************
*                          *              *
* Source Code Frame:       *              *
* src=frame-1-source.html  *              *
* name=source              *  Contents Frame:*
*                          *              *
*                          *  src=..      *
**************************  name=contents  *
*                          *              *
* VRML Frame:              *              *
* src=frame-1-vrml.html    *              *
* name=vrml                *              *
*                          *              *
*                          *              *
*******************************************
```

Please have a look at the example to "see it for real".

Example 4.2.1 *VRML in a Frame*

HTML:	../examples/frames/frame-1.html

Here is the contents of the Main page (frame-1.html):

```
<HTML>
<HEAD>
<TITLE>Embedded VRML Frame ( 7-Mar-1997)</TITLE>
<!-- Changed by: D.K.S.,  7-Mar-1997 -->
</HEAD>
<FRAMESET COLS = "75%, *">
  <FRAMESET ROWS = "40%, *">
     <FRAME SRC="frame-1-source.html" name="source" scrolling="auto"
      Frameborder="yes" border="3">
```

[1] http://vrml.sgi.com/handbook/MasterLayoutPC.html
[2] http://www.newbie.net/frames

```
    <FRAME SRC="frame-1-vrml.html" name="vrml" scrolling="auto"
     Frameborder="yes" border="3">
  </FRAMESET>
  <FRAME SRC="frame-1-contents.html" name="contents" scrolling="auto"
   Frameborder="yes" border="3">

</FRAMESET>
</HTML>
```

The contents of the files `frame-1-source.html` and `frame-1-vrml.html` does not matter since they will be replaced anyhow by either code listing or your VRML plug-in.

Now, all you need to do in order to display a link in `frame-1-contens.html` in the right frame is to use a "TARGET" field in the "A HREF" tag as in the following example:

```
<HTML>
<HEAD>
<TITLE>Contents ( 7-Mar-1997)</TITLE>
<!-- Changed by: D.K.S.,  7-Mar-1997 -->

</HEAD>
<BODY>
<H1>Contents</H1>

<a href="anim-move-1.text" target="source">anim-move-1.text</a><br>
<a href="anim-move-1.wrl" target="vrml">anim-move-1.wrl</a><p>

..... and so on

<a href="anim-touch-3.text" target="source">anim-touch-3.text</a><br>
<a href="anim-touch-3.wrl" target="vrml">anim-touch-3.wrl</a><p>
<hr>
Use the BACK button of your www browser to get back from where you came
in the <A HREF="/guides/vrml/vrmlman/vrmlman.html">VRML Primer</A>.
<hr>
</BODY>
</HTML>
```

4.2.2 VRML to HTML in a frame

You can use the same kind of technology to display a HTML page in a frame when you click on a VRML anchor. (see section 1.2.6).

Example 4.2.2 *VRML in a Frame II*

HTML:	../examples/frames/frame-2.html

The definition of the frame is almost the same:

```
<HTML>
<HEAD>
<TITLE>Embedded VRML Frame ( 7-Mar-1997)</TITLE>
</HEAD>
<FRAMESET COLS = "75%, *">
  <FRAMESET ROWS = "50%, *">
     <FRAME SRC="frame-2-contents.html" name="contents" scrolling="auto"
      Frameborder="yes" border="3">
```

```
    <FRAME SRC="frame-2-vrml.wrl" name="vrml" scrolling="auto"
     Frameborder="yes" border="3">
  </FRAMESET>
    <FRAME SRC="frame-2-explanation.html" name="explanation" scrolling="auto"
     Frameborder="yes" border="3">
</FRAMESET>
</HTML>
```

Now let's see how we tell the VRML plug-in to display links in another frame:

```
# LEVEL II: The bunch as a whole
Anchor {
   url "http://tecfa.unige.ch/tecfa-people/tecfa-people.html"
   description "A link to Tecfa People"
   parameter "target=contents"
   children [
      Transform { .......
```

The code for this example[3] is exactly the same as for the LOD example[4] in section 3.2 on page 36, **except** for the **parameter field**.

4.3 VRML code generation

There are (in theory) three situations:

1. Generate a VRML file (or frame content) from outside VRML.
2. Generate VRML from inside VRML.
3. Generate code from outside into an existing scene. See chapter 6 on page 65 which will introduce you to the External Authoring Interface (EAI).

In this section we will only cover the first item somewhat. Make sure not to confuse Vrml Code generation with Javascript with scripting interfactive VRML. This section is absolutly optional reading, you don't need anything of this to move on with the following chapters.

4.3.1 VRML Code Generation with Javascript

Generating a VRML scene with Javascript is rather easy once you know how to deal with Netscape Frames using Javascript. [You also need a browser that works, some older browsers will not allow to it].

Generating VRML can be useful in various applications, e.g. you can write a VRML generator for your students or show mathematical functions (or other data) in 3D. The next example has been very strongly inspired from Yasuyuki Suzuki[5]'s Object Creation[6] example.

Example 4.3.1 *VRML code generation with Javascript*

Contents:	Contains 2 frames. A html/javascript frame allows to create simple objects
HTML:	generate-js-1-frame.html
Frame Source:	generate-js-1-frame.text
HTML/JS Source:	generate-js-1.text

Look at the full example before you move on please ! Below we will discuss a slightly simplified version. Anyhow, in order to understand this example you rather must learn more about Javascript

[3] ../examples/frames/frame-2-vrml.text
[4] ../examples/inter/inter-LOD-1.text
[5] http://meg.me.aoyama.ac.jp/%7Eyasuyuki/
[6] http://meg.me.aoyama.ac.jp/%7Eyasuyuki/javascript/vrml/vrml.html

than anything about VRML. (See Netscape's OnLine documentation[7] for developpers. At Tecfa we have a copy of the JavaScript Guide[8]).

Let's have a quick look at the frameset in the file generate-js-1-frame.html (if you don't understand this, please read section 4.2 on page 40).

```
<FRAMESET COLS="*, 70%">
<FRAME SRC="generate-js-1.html" NAME="menu_frame"
       BORDER="0">
<FRAME SRC="javascript:top.vrml();" NAME="vrml_frame"
       NORESIZE SCROLLING="no">
</FRAMESET>
```

Note that we also generate dynamically an inital object in the frame by directly calling `javascript:top.vrml()`. We could have left this page blank or inserted some canned VRML `*.wrl` page. Just remember that the frame to the left is called `menu_frame` and the frame to the right `vrml_frame`.

The user enters size and color information in the left (HTML/Javascript) frame and then can click on "Create Box". This will render the object in the VRML frame to the right.

This is a simple HTML Form that will collect information from the user. Note the following things:

1. The "form" tag has just a name field (no method field needed)

2. The input "button" will trigger an onClick event when pressed that will launch the "makeBox" script. This script will then poll the form for values the user entered (see below).

```
<FORM NAME="cont">
<STRONG>Example Objects:</STRONG><p>
<STRONG>Custom Box:</STRONG><p>
 Width : <INPUT TYPE="text" NAME="w" VALUE="1" SIZE="5"><br>
 Height : <INPUT TYPE="text" NAME="h" VALUE="1" SIZE="5"><br>
 Depth : <INPUT TYPE="text" NAME="d" VALUE="1" SIZE="5"><p>
 Colors (be sure to enter values between 0 and 1!<br>
 Red  : <INPUT TYPE="text" NAME="red" VALUE="1" SIZE="5"><br>
 Green : <INPUT TYPE="text" NAME="green" VALUE="0.2" SIZE="5"><br>
 Blue  : <INPUT TYPE="text" NAME="blue" VALUE="0" SIZE="5"><p>

<INPUT TYPE="button" NAME="make" VALUE="Create Box" onClick="makeBOX();"><P>
<HR>
</FORM>
```

Here are the most important elements of the (simplified) source code. The function makeBOX will draw a box of a given size in given colors. It will access the values entered in the form below and write lines to the VRML frame.

In this example (like in the original) we poll the values the user entered from the form, e.g.:

```
w = top.menu_frame.document.cont.w.value;
```

will assign the value for the width of the box to a temporary variable "w". There are several ways of doing this as you can note. Also note that polling the form is one way of doing it. Alternatively we could have passed the values to this function within the `INPUT` tag of the form. Finally be aware that we do not any value checking. It's up to the user to enter good values.

[7] http://developer.netscape.com/library/documentation/
[8] http://tecfa.unige.ch/guides/java/javascript/

```
var MIMEType = 'x-world/x-vrml';

function makeBOX() {
     var w, h, d, red, green, blue;
// Note: all these ways of accessing values work
     w = top.menu_frame.document.cont.w.value;
     h = top.menu_frame.document.forms[0].h.value;
     d = document.forms[0].d.value;
     r = document.forms[0].red.value;
     g = document.forms[0].green.value;
     b = document.forms[0].blue.value;

     with (top.vrml_frame.document) {
        open(MIMEType);
        writeln('#VRML V2.0 utf8');
        writeln('Shape \{');
        writeln('  appearance Appearance \{');
        writeln('         material Material \{ ');
        writeln('            emissiveColor ' + r + ' ' + g + ' ' + b);
        writeln('                          \} \}');
        writeln('  geometry Box \{');
        writeln('     size ' + w + ' ' + h + ' ' + d)
        writeln('\} \}');
        close();
     }
}
```

The only other thing you need to know is how to write to the VRML frame:

1. We first get some handling on the VRML frame's document window object. The `with` statement makes referring to objects properties and methods simpler (you don't have to type the full path each time). In the following expression `top` refers to the "top" node in the Javascript hierarchy (?), `vrml_frame` is the name we gave to the frame where the plugin will appear and `document` refers to it's contents.

   ```
   with (top.vrml_frame.document) {
   ```

2. Next we "open" output to the vrml frame with `'x-world/x-vrml'`.

   ```
   var MIMEType = 'x-world/x-vrml';
   .....
   open(MIMEType);
   ```

   We defined the variable `MIMEType` here since we do it more than once.

3. Next you simply write out the VRML code (including a valid VRML header first).

4. Then you close the output to the frame.

Note how we can simply "string-add" "canned" VRML instructions with variables in the code. Javascript is very convenient in this respect.

4.4 Adding and Removing Kids with Javascript

Warning, needs some cleaning probably (has been left alone since Spring 97 and may crash your Browser .. it does crash my Irix CP 1.02 but works with Win95/Cosmo 2). **Before you start**

reading please work out chapter 5 on page 49 that will teach a bit more about JavaScript and VRML.

In this example we show how to add and remove DEFed objects from children properties of Transform/Group Nodes.

Example 4.4.1 *Add/Remove DEFed objects I*

Contents:	add/remove children
VRML:	generate/generate-js-5.wrl
Source:	generate/generate-js-5.text

The following code defines a node that contains child we want to reuse and that contains a TouchSensor.

```
DEF RedBoxSwitch Transform {
   translation -5 10 0
   children [
      DEF RedBox Shape {
         appearance Appearance {
            material Material { diffuseColor 1 0 0 }
         }
         geometry Box {}
      }
      DEF MakeRedBox TouchSensor {}
      ]
}
```

Next we define a place where we want to put a cousin of the DEFed object. Note the empty `children [ ]` of the `BoxHolder` Node.

```
DEF Scene Group {
   children [
      ...................
      DEF BoxHolder Transform {
         translation 0 1 0
         children [ ]
      }
      ...................
     ] }
```

A little script will be activated by the touchsensor, because of this route statement:

```
  ROUTE MakeRedBox.isActive TO GenBoxS.isActive
```

and then activate in turn two routes that remove/add the node.

```
ROUTE GenBoxS.child TO RedBoxSwitch.removeChildren
ROUTE GenBoxS.child TO BoxHolder.addChildren
```

Here is the Code of the script

```
DEF GenBoxS Script {
   eventIn SFBool isActive
   eventOut MFNode child
   field MFNode node USE RedBox
   url ["javascript:
                function isActive(value) {
                        if(value) child = node;
```

```
                }
        "]
}
```

The next example is slightly more complicated, because it will allow you to "reset" the scene. Note that it breaks with Cosmo and works with WorldView.... maybe my fault. [Will have to work on this when I have time].

Example 4.4.2 *Add/Remove DEFed objects (and reset) II*

Contents:	add/remove Children
VRML:	generate-js-6.wrl
Cosmo:	generate-js-6C.wrl
Source:	generate-js-6.text

Compared to the above example we need to add a few things: We want to add a cousin to the place where we have removed a childe after a reset. So a typical node now looks like this (it includes the `RedBoxHolder`) Group from which we will remove/add a child:

```
DEF RedBoxSwitch Transform {
   translation -5 10 0
   children [
      DEF RedBoxHolder Group {
         children [
            DEF RedBox Shape {
               appearance Appearance {
                  material Material { diffuseColor 1 0 0 }
               }
               geometry Box {}
            }
           ] }
      DEF MakeRedBox TouchSensor {}
     ]
}
```

Below is the script that will reset the scene. Note how we bind objects in the script in order to do some direct output (instead of using lots of routes):

```
DEF ResetS Script {
   eventIn SFBool isActive
   eventOut MFNode child
   directOutput TRUE
   field SFNode redBallHolder USE RedSphereHolder
   field SFNode redBall USE RedSphere
   field SFNode redBoxHolder USE RedBoxHolder
   field SFNode redBox USE RedBox
   field SFNode blueBallHolder USE BlueSphereHolder
   field SFNode blueBall USE BlueSphere
   field SFNode boxHolder USE BoxHolder
   field SFNode sphereHolder USE SphereHolder

   url "javascript:
               function isActive(value) {
                  redBallHolder.addChildren = redBall;
```

```
                    blueBallHolder.addChildren = blueBall;
                    redBoxHolder.addChildren = redBox;
                    boxHolder.removeChildren = redBox;
                    sphereHolder.removeChildren = redBall;
                    sphereHolder.removeChildren = blueBall;
                }
        "
}

ROUTE Reset.isActive TO ResetS.isActive
```

`Reset` is a simple touchsensor (a yellow ball) in the scene.

Chapter 5

Introduction to moving, interactive VRML

DISCLAIMER: I don't really understand a lot about VRML and even less about interactive VRML. **In addition, some code is simply disgusting (blatant lack of abstraction and some Voodoo code lines). Will clean this up progressively.**

This chapter will graducally build up. So far, just consider that there is an order of complexity in doing/learning VRML:

1. Understanding Events and ROUTE
2. Doing simple routes from sensors to objects
3. Adding Interpolators
4. Pluging some Javascript (or VRMLScript) in between
5. Doing things with JAVA
6. Connecting up to a multi-user server

I intend to look into all of these at some point. Right now this chapter covers *some* of 1-4 (not enough!).

5.1 The idea

Interaction with a vrml world can take many forms, but the basic principle is the same. We can place various sensors in the world that monitor some user action (like loading the file, clicking on some object, getting close to some object) and that will trigger some animation.

The logic of programming interactive worlds is more or less the same as in traditional GUI programming. You will have to define what =>Events[1] the browser should generate who should handle them and how they should be handled. The difference is that all (?) VRML nodes can generate and receive events. =>ROUTEs[2] will define how different VRML are bound together (i.e. affect each other).

In order to understand events, you have to rethink the concept of a VRML node. You might want to read the entry =>"Nodes, Fields and Events"[3] in the specification also. Typically, a VRML node is composed of a set of fields. Some of these, i.e. the exposed fields can be altered by incoming events. Each VRML node also has a set of incoming and outcoming events defined and

[1] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#Events
[2] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#Routes
[3] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#Nodes,Fields,andEvents

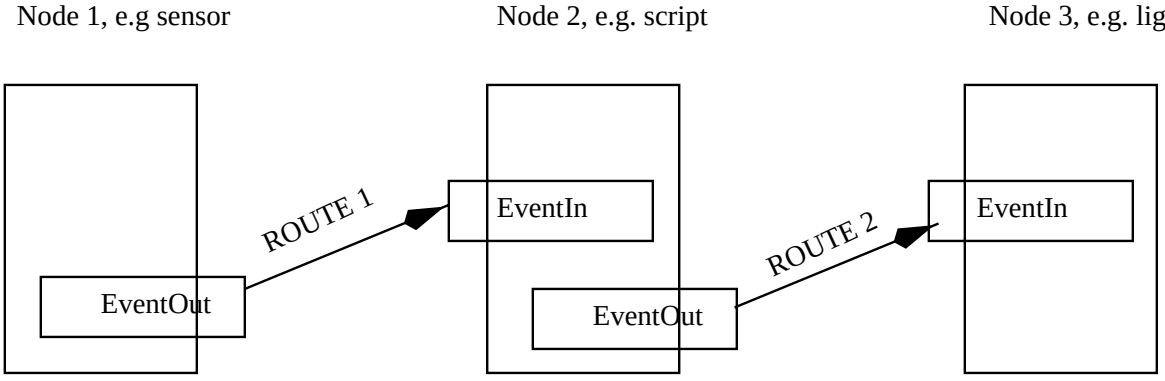

Figure 5.1: the route/events model in VRML

you can hook together nodes by defining a route that links together any outcoming and incoming event of the same time. In the next sections you will learn how to to this.

In some cases, user triggered events are first routed to a script node. Scripts in VRML are just a kind of VRML nodes, but nodes that can compute and that can make changes to any kind of exposed VRML fields. Figure 5.1 describes such a setup.

At a more abstract level we can describe a typical "Animation Event Path" as described in chapter 9 [Hartman et Wernecke, 1996], a book you might want to buy if you want a better introduction. More sophisticated animations are triggered off by some sensor, (e.g. the user clicks on an object), the event generated by the browser is routed to a script that will trigger a timer. The timer will trigger a engine (e.g. a position interpolator for moving objects) which in turn affect a VRML object (e.g. the position field of a transform node).

In the next section (5.2) will first teach you how to do animation without scripting and we will introduce the other elements of the typical animation path

5.2 Introduction to Events, Routes, Sensors and Interpolators

The basis for animation is to dynamically change the values of a given object's fields. Lets imagine that you want to rotate an object and you that have a geometry node embedded in a Transform node like in the following example:

```
DEF Tecfa Transform {
   rotation  0 1 0  0
   children [
      Inline { url "inter-persons-2.wrl" }
     ]
}
```

In order to rotate the object, you want to able to change the value of the rotation field.

Events: All exposed fields of a VRML node can be changed via incoming events. E.g. if you look at the definition of the Transform node, you can see that you can for example scale or rotate or translate and object, and even add or remove nodes from its children.

```
Transform {
  eventIn      MFNode     addChildren
  eventIn      MFNode     removeChildren
  exposedField SFVec3f    center          0 0 0
```

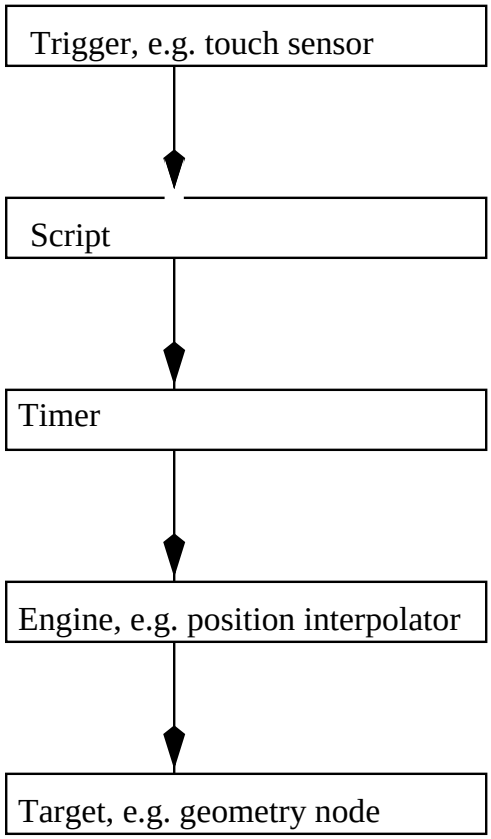

Figure 5.2: A typical Animation Event Path in VRML

```
  exposedField MFNode      children         []
  exposedField SFRotation  rotation         0 0 1  0
  exposedField SFVec3f     scale            1 1 1
  exposedField SFRotation  scaleOrientation 0 0 1  0
  exposedField SFVec3f     translation      0 0 0
  field        SFVec3f     bboxCenter       0 0 0
  field        SFVec3f     bboxSize         -1 -1 -1
}
```

In addition to fields, most VRML nodes have so-called =>Events[4] slots which are distinct from the "fields" you already encountered. There are two kinds of events:

1. "eventIN", i.e. incoming events that are messages sent by other nodes to change some state (field) within the receiving node.
2. "eventOUT", i.e. outgoining events used to send messages (events) to destination nodes. "eventOUT" slots are less frequent (mostly sensor nodes).

In addition to specific "eventIN" and "eventOUT" definitions, all the **exposedFields** implicitly define events too:

1. You would use e.g. **LIGHT.set_on** to change the value of the "on" field in a PointLight node named "LIGHT"
2. and **LIGHT.on_changed** to propagate the state of the changed "on" field to other nodes (via a ROUTE statement).

See below for examples.

Routes: In order to connect a node generating a message (event) with an other node receiving a message you must use a "ROUTE" statement. A node that produces events of a **given type** can be routed to a node that receives events of the **same type** with the following syntax:

```
ROUTE NodeName.eventOutName TO NodeName.eventInName
```

Remember this: you can only connect fields of nodes that are *exactly* of the same type, else you will have to write a script in between as you will learn later.

Here are two examples of such a route statement:

```
ROUTE LIGHT_CONTROL.isActive TO LIGHT.set_on
ROUTE Random.feeling_changed TO Hasard.set_consequence
```

Since you can ONLY route messages that are exactly of the same data type, it might be a good idea now to start mastering all of VRML's data types (see section 8.4 on page 87).

Sensors: Usually Events are triggered by **sensors**. A variety of =>Sensor Nodes[5] allow you to detect what and when something happens. We will discuss a few of them in the next sections. Now let's look at a simple TouchSensor example. The =>Touch Sensor[6] node is defined as follows:

```
TouchSensor
  exposedField SFBool  enabled TRUE
  eventOut     SFVec3f hitNormal_changed
  eventOut     SFVec3f hitPoint_changed
  eventOut     SFVec2f hitTexCoord_changed
  eventOut     SFBool  isActive
  eventOut     SFBool  isOver
  eventOut     SFTime  touchTime
```

[4] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#Events
[5] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#SensorNodes
[6] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#TouchSensor

The TouchSensor generates events as the pointing device "passes over" any geometry nodes that are descendants of the TouchSensor's parent group. In the code below we will make use of the "isActive" field which will change it's value from FALSE to TRUE whenever the user points to it (e.g. moves the mouse "over" it *and* holds down a button). As soon as the state of this field changes, the contents of the field (TRUE or FALSE) is *ROUTE*ed to a PointLight node.

A simple sensor switches light example Before reading further on, you might want to have a look at the example.

Example 5.2.1 *A Simple Touchsensor Effect*

VRML:	../examples/anim/anim-touch-1.wrl
Source:	../examples/anim/anim-touch-1.text

Click on the "switch" and the blue sphere will be lit while you hold down the mouse. Holding down the mouse over the sensor means that the *LIGHT_CONTROL.isActive* is TRUE, releasing the mouse makes it FALSE again.

Here are the important parts of the code:

```
# A TouchSensor
#(in the same group as a geometry object so that it can be seen)
#
Transform {
   translation 0.5 1 5
   children [
      DEF LIGHT_CONTROL TouchSensor {
      }
      Inline { url ["light-switch.wrl"] }
     ]
}

# The light, off when the file is loaded
#
DEF LIGHT PointLight {
   on   FALSE
   location     -3 2 2
}

# The object to be lit in here:
#
 ......

ROUTE LIGHT_CONTROL.isActive TO LIGHT.set_on
```

As you can see in the following definition of the =>PointLight[7] node, it has an **exposedField** (i.e. accessible field) called "on". All the ROUTE statement above does, is to define a connection between the sensor's *isActive* event to the PointLight's *set_on* event (i.e. to the value of the *on* field of PointLight. As in other GUIs it's the browser that will handle the routing of user actions (events) to sensors, i.e. they will become *active*. Below we give the complete definition of PointLight and you can see that there is indeed a *exposedField* "on" and therefore implictely a *set_on* event defined. [Yes I know, a section on lights is missing in this manual.]

[7] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#PointLight

```
PointLight {
  exposedField SFFloat ambientIntensity  0
  exposedField SFVec3f attenuation       1 0 0
  exposedField SFColor color             1 1 1
  exposedField SFFloat intensity         1
  exposedField SFVec3f location          0 0 0
  exposedField SFBool  on                TRUE
  exposedField SFFloat radius            100
}
```

Of course you can also make light switches that will turn on some light permanently but (if I am not wrong) you have to do this with some scripting (see section 5.3.1 on page 59 that does exactly this). But before we look into scripting, let's learn a few more things about events.

5.2.1 TimeSensors and Rotations with Interpolators

[Warning, this section is difficult reading. Have to rewrite this at some point. Study the examples while you digest the text]

In this section you will meet first an other sensor, the TimeSensor which can be used for example to produce "ticks" that will get an animation going for a certain time and with a certain speed. Then we introduce the concept of an interpolator, i.e. a node that can generate numbers to producde an animation path.

Time Sensors: The =>TimeSensor[8] is a kind of =>Time Dependant Node[9]. TimeSensors generate events as time passes. TimeSensors can be used to drive continuous simulations and animations, periodic activities (e.g., one per minute), and/or single occurrence events such as an alarm clock. Each Time Dependant Node contains the exposedFields: *startTime*, *stopTime* and *loop* and the eventOut *isActive*. Time dependant nodes are **inactive** until **startTime** is reached. Once active (isActive = TRUE), they will excute for 0 or more cycles until **stopTime** is reached or **loop** becomes FALSE. Note that they will ignore *stopTIME* if **stopTime** < **startTime**. To sum it up:

- A TimeSensor is like a stop watch: There is a start time, a stop time and it cycles (you fix the amount too).
- It generates events while it is running. These events can be routed to change fields of other nodes.

Here is the full definition of TimeSensor

```
TimeSensor {
  exposedField SFTime   cycleInterval 1
  exposedField SFBool   enabled       TRUE
  exposedField SFBool   loop          FALSE
  exposedField SFTime   startTime     0
  exposedField SFTime   stopTime      0
  eventOut     SFTime   cycleTime
  eventOut     SFFloat  fraction_changed
  eventOut     SFBool   isActive
  eventOut     SFTime   time
}
```

[8] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#TimeSensor
[9] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#4.6.9

Note that VRML time is not human readable. =>SFTime[10] are the number of seconds since Jan 1 1970. In the example that follows we will simply use it to get an animation going as soon as the user loads a world. I.e. the node below will just run eternally as soon as you load the file.

```
DEF TIME TimeSensor {
   cycleInterval        20
   startTime    0
   stopTime     -1
   loop         TRUE
}
```

When a TimeSensor becomes active it will generate an *isActive = TRUE* event and begin generating *time, fraction_changed,* and *cycleTime* events, which may be routed to other nodes to drive animation or simulated behaviors. The two most important output events are:

- **isActive** which will output TRUE at timer start and FALSE at timer stop
- **fraction_changed** which will output values from 0.0 to 1.0 during a cycle. At the start of each cycle its value is 0.0

We will discuss more about those events in the example further down.

We route the *fraction_changed* event to a so called OrientationInterpolator whose time-sensor triggered output is then sent to the object we are rotating. In order to guarantee a smooth animation, it will compute vaious orientation points as you will learn below.

The Orientation Interpolator Let's look at the =>OrientationInterpolators[11] which is one kind of VRML's =>Interpolators[12]. In short, they allow you do simple animations (e.g.rotate objects, move objects around, change colors of things). An interpolator will generate values based on "clues" defined by a set of **key**s (received from the *set_faction* eventIn) and a set of corresponding **keyValue**s. In other words, it will automatically compute *intermediate positions* for each time_fraction. You only have to specify a few to get it going like in the example below. Key Values are of a different type for each Interpolator. Here is the definition for the OrientationInterpolator:

```
OrientationInterpolator {
  eventIn      SFFloat    set_fraction
  exposedField MFFloat    key           []
  exposedField MFRotation keyValue         []
  eventOut     SFRotation value_changed
}
```

A simple Time Sensor/Orientation Interpolator example You now can first look at the example before reading on.

Example 5.2.2 *A simple Timesensor Example*

VRML:	../examples/anim/anim-rotate-1.wrl
Source:	../examples/anim/anim-rotate-1.text

Below, you can see both the definition of the Orientation Interpolator and the ROUTE statements of the example world.

```
DEF Tecfa Transform {
```

[10] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/fieldsRef.html#SFTime
[11] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#OrientationInterpolator
[12] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/concepts.html#4.6.8

```
   rotation  0 1 0  0
   children [ ..... ]
}

DEF TIME TimeSensor {
   cycleInterval        20
   ........
}


DEF OI_Tecfa OrientationInterpolator {
   key          [ 0 0.5 1 ]
   keyValue     [ 0 1 0  0, 0 1 0  3.1416, 0 1 0  6.2832]
}

ROUTE TIME.fraction_changed TO OI_Tecfa.set_fraction
ROUTE OI_Tecfa.value_changed TO Tecfa.rotation
```

The *keyValue* field is of type MFRotation, i.e. a comma separated set of orientation values that correspond to each element in the *key* field. In our example you can see that we rotate around the y axis (0 1 0) using orientations 0, 3.1416 and 6.2832. VRML now uses those positions to interpolate, i.e. to generate rotation values in between which are sent out with *value_changed* eventOUT each time the node receives a *set_fraction* eventIN.

The *set_fraction* event is of type SFFloat in the range of [0,1] and usually routed to from a TimeSensor's **fraction_changed** event. The fraction_changed event is an indicater in the range of [0,1] that tells how much of a complete cycle has elapsed so far. The duration of a cycle itself is defined in seconds by the **cycleInterval** field. E.g. in our example, the interval is 20 seconds.

```
DEF TIME TimeSensor {
   cycleInterval        20
    ......
}
```

Indeed, if you look again at this simple rotation example[13] you can see that a full rotation lasts about 20 seconds. Rotation speed is constant, because we told the interpolator so with the key/keyValues.

To sum it up Interpolators:

- the **set_fraction** input event will set the current fraction along the key path
- the **value_changed** output event will output the position according to the keyValue path each time the fraction is set.

The next example shows some more irregular rotation (randomly put together).

Example 5.2.3 *A simple Timesensor Example II*

VRML:	../examples/anim/anim-rotate-2.wrl
Source:	../examples/anim/anim-rotate-2.text

Of course, eternal rotations can be annoying to the user and they certainly waste resources. Since (as far as I understand it) there is no way to start and stop animation at a given time after the scene has been loaded (time is in seconds starting some obscure date) you'd have to write a touchsensor that activates the rotation and do some Javascript for that.

[13] ../examples/anim/anim-rotate-1.wrl

5.2.2 More on Time Sensors and Interpolators

Let's see how to "configure" a TimeSensor according to your needs:

1. Continously running:

```
loop TRUE
stopTime < startTime
```

2. Runs one cycle and then stops:

```
loop FALSE
stopTime < startTime
```

3. Runs until stopped, or after cycle is over:

```
loop TRUE or FALSE
stopTime >= startTime
```

The next example inspired from David Frerichs' exellent Creating Integrated Web Content with VRML 2.0 Tutorial[14] In this example you can see something sliding forth and back once you click on the little red switch.

Example 5.2.4 *A TouchSensor activating a Shuttle*

VRML:	../examples/anim/anim-shuttle-1.wrl
Source:	../examples/anim/anim-shuttle-1.text

Let's have a look at things that are new: When we load the file, the TimeSensor (called TimeSource here) has the **loop = FALSE**, so there will be no eventOUTs looping:

```
DEF TimeSource TimeSensor {
    cycleInterval 10
    loop FALSE
 }
```

Then we use a TouchSensor node (see section 5.2 on 50) to activate the loop. More precisely, the *Starter.isActive* event (Touchsensor) is routed to *TimeSource.set_loop* (Timesensor). As soon as you click on the switch the loop will be turned on (and remain so). Here is the code of the switch and it's route:

```
DEF Switch Transform {
    translation 2 -2 5
    children [
         DEF Starter TouchSensor { }
         Inline { url ["light-switch2.wrl"] }
         ]
}
ROUTE Starter.isActive TO TimeSource.set_loop
```

Finally, in order to move the object forth and back we use a =>PositionInterpolator[15] that works more or less the same way as the OrientationInterpolator encountered in the previous section. The keyValue fields are the kinds of values (SFVec3f) that one uses in translation fields. Here is the other half of the code:

[14] http://reality.sgi.com/frerichs_esd/course/index.html
[15] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#PositionInterpolator

```
DEF TECFA Transform {
   translation 0 0 -50
   children [
      DEF TimeSource TimeSensor {
         cycleInterval 10
         loop FALSE
      }

      DEF Animation PositionInterpolator {
         key   [ 0, .50, 1.0 ]
         keyValue [ 0 0 -40, 0 0 6, 0 0 -40]
      }

      Inline { url "inter-persons-2.wrl" }
     ]
}
ROUTE TimeSource.fraction_changed TO Animation.set_fraction
ROUTE Animation.value_changed TO TECFA.translation
```

5.3 Introduction to Scripting with Javascript

Many things you want to do are too complex for the built-in sensors and interpolators. In this case you can write nodes that are program scripts, that will:

- accept event inputs (eventIn)
- generate event outputs (eventOut)
- read and write any exposed fields of DEFed other nodes

Here is the formal definition of a =>Script[16] node:

```
Script {
  exposedField MFString url           []
  field        SFBool   directOutput  FALSE
  field        SFBool   mustEvaluate  FALSE
And any number of:
  eventIn      eventTypeName eventName
  field        fieldTypeName fieldName initialValue
  eventOut     eventTypeName eventName
}
```

As you will learn in the examples in the next sections, the "url" fields contains either a pointer to a script file or a full script.

The VRML standard leaves it open to the browser manufactures what language they will support for scripting. However, if they implement scripting with either JAVA or JavaScript, conformance to the spectifcations is required!

In this tutorial we don't teach you how to program with JavaScript or even to program in the abstract. See our JavaScript Page[17] for the most important pointers that will help you to get started. There are 2-3 good on-line tutorials and several good books. Scripting with JavaScript will be based on the ECMAScript[18] scripting language with extensions that are *required* for the VRML plug-in. I.e. it does not contain typical Netscape objects. This means for example that you can't open popup windows and such!

[16] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Script
[17] http://tecfa.unige.ch/guides/js/pointers.html
[18] /guides/vrml/vrml97/spec/part1/javascript.html

WARNING: If you use the Cosmo 1.02 on Win95 for any reason (you should not) or Cosmo 1.02 on Irix (you have to), you have either to replace:

```
url "javascript: ...
      by
url "vrmlscript: ...
```

The better solution is just to duplicate the code within the url field as in the following example. Your browser will pick whatever he likes best.

```
url [
  "javascript: ......

       ",
  "vrmlscript: ......

       "]
```

From the little I have looked at the documentation, vrmlscript looks more or less like javascript, i.e. it implements a subset that contains most functionalities you need to get started. My examples all work the same (except for the name of the language). Compare yourself:

- VrmlScript Reference [19]
- JavaScript Reference [20]

5.3.1 Touchsensors, ROUTE and Javascript

Example 5.3.1 *The Light-on problem again*

VRML:	../examples/anim/anim-touch-2.wrl
Source:	../examples/anim/anim-touch-2.text
Contents:	JavaScript, TouchSensor

The "simple touchsensor effect" example discussed in section 5.2 directly wired the state of a TouchSensor to the state of a PointLight. This meant that the light went on when you were holding down the mouse button, but it went off as soon as you released the button. Now let's see how to make a switch that that will keep the object lit with JavaScript. (Note that I don't know yet if I could do without, but that's another problem).

The trick is to build a JavaScript node in between that will receive the input from the TouchSensor and then simply switch the PointLight on. When the user releases the mouse button, the Script allready propagated the information and the light will stay lit.

The definition of the TouchSensor exactly looks the same:

```
# The touchsensor for putting the Light ON
#
Transform {
   translation -0.5 1 5
   children [
      DEF LightONSwitch TouchSensor {
      }
      Inline { url ["light-switch.wrl"] }
     ]
}
```

[19] http://tecfa.unige.ch/guides/vrml/vrml2/spec/part1/vrmlscript.html
[20] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/javascript.html

Now if you look at the routing statements below you can see that the event LightONSwitch.isActive (the user points on the LightONSwitch node) is routed to eventIn of a node called "lightONScript". This node is an "engine" written in JavaScript that will compute this "In" event and produce an "Out" event that in turn is wired to the LIGHT node (i.e. the PointLight).

```
# link activation of lightON to eventIn of lightONScript
#
ROUTE LightONSwitch.isActive TO lightONScript.lightONIsActive
#
# link "output" of lightONScript to LIGHT
#
ROUTE lightONScript.lightIsON TO LIGHT.set_on
```

Let's look at the JavaScript node called lightONScript now: We define a =>Script[21] node with two **mandatory** slots: "eventIn" and "eventOut". For each of those slots we must define its **type** (in our case we use SFBool since both the TouchSensor's and the light's fields we are interested in are either on or off, i.e. TRUE or FALSE). The second argument are the **event names** that must be unique identifiers within the scope of the node.

```
DEF lightONScript Script {
   eventIn SFBool lightONIsActive
   eventOut SFBool lightIsON

   url "javascript:
                function lightONIsActive(active) {
                        lightIsON = TRUE;
                }"
}
```

In order to compute the eventOUT from the eventIn we must assign a function to the "url" field. That function has the *same* name as *eventIn* and assigns a value to the variable of *eventOut* [hmmmm this does not sound very clean]. Now where do we put the function ? Either we insert the function as **string** in the url field or we use an URL that points to a script. (Note, that you can write several functions if needed).

As you can see, the script is fairly simple. It takes one argument (the value of eventIn, which we don't use here) and sets the value of the eventOUT.

In order to change a value of a node, you don't actually need to ROUTE the eventOUT. Instead you can directly set a value within the Scripting node. Here is how you do it:

```
# Clicking on the green Light Switch (ON)
#
DEF lightONScript Script {
   eventIn SFBool lightONIsActive
   field SFNode node USE LIGHT
   directOutput TRUE
   url "javascript:
                function lightONIsActive(active) {
                        node.set_on = TRUE;
                }"
}

# link activation of lightON to evenIn of lightONScript
ROUTE LightONSwitch.isActive TO lightONScript.lightONIsActive
```

In the line

[21] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Script

```
field SFNode node USE LIGHT
```

we bind the LIGHT node to "node" and in the function we can simply send an event to the "set_on" eventIn of the LIGHT on with

```
node.set_on = TRUE;
```

Example 5.3.2 *The Light-on problem without eventOUT*

VRML:	../examples/anim/anim-touch-3.wrl
Source:	../examples/anim/anim-touch-3.text

The script can access any exposedField, eventIn or eventOUT of any node to which it has a pointer in a "field". Note that eventIN of the passed node (in our example that is LIGHT) can be only to the left side (meaning you can set a value) and eventOUT of the passed node can only be on the right side (meaning you can read it). This needs some example .... hold on.

Here is another example that activates something. Instead of turning on/off a light, we change the ViewPoint. It has been strongly inspired from Markus Roskothen's Touch Me[22] example. You can see an other example of using a TouchSensor with JavaScript.

Example 5.3.3 *A viewpoint changing TouchSensor Script*

VRML:	../examples/anim/anim-move-1.wrl
Source:	../examples/anim/anim-move-1.text

5.3.2 Dealing with state

Let's assume that you that feel that two switches (ON and OFF) for turning on and off the light is too much.

Within a Script Node you can define fields that will serve as state variables like you would encounter in object-oriented programming. In other words you can remember things, e.g. you can store objects, colors, user activities and more. The only restriction is that those "variables" must be VRML data (field) types.

Example 5.3.4 *The Light-on problem with state*

VRML:	../examples/anim/anim-touch-state.wrl
Source:	../examples/anim/anim-touch-state.text
Contents:	JavaScript state, TouchSensor,

Here is the code of the Script Node that solves the light switch problem:

```
DEF lightScript Script {
   eventIn SFBool lightIsActive
   eventOut SFBool fixLightState
   field SFBool state FALSE

   # Only do something in the function
   # if lightIsActive = TRUE (the active param)
   # because as soon as the user releases the mouse
   # button a FALSE is generated again.
   url ["javascript:
        function lightIsActive(active) {
           if (active) {
              if (state == FALSE) {
                 state = TRUE;
```

[22] http://www.wrvuniverse.com/tutorials/touchme.html

```
                fixLightState = TRUE;
            }
            else {
                state = FALSE;
                fixLightState = FALSE;
            }
        }
    }"
   ]
}
```

Our state variable that will remember if the light is on or off is declared as a boolean (note that name "state" is our choice):

```
field SFBool state FALSE
```

Now if you read the specifications of the =>Touch Sensor[23] node again you will learn that when the user clicks on it, it generates an isActive Event==TRUE. However when the user releases the mouse button, an isActive Event==FALSE is generated.

We are not interested in the fact that the user releases a button, so we ignore it. When we pass the value of the eventIn Field lightIsActive to the function we only do something if it is TRUE (as we said before):

```
if (active) { .... }
```

Within the "big" if clause we then look at the value of our state variable. If `state = FALSE` we know that the light is off and we will turn it on and remember it by toggling the state variable. On the contrary we will turn the light on and remember that too.

Now of course we also should change the color of the light button. There are many ways to do it, e.g. change the color of the little cube or use a switch node to insert another one. We leave that exercice to the reader.

Well, now did we really had to make use of state variable ? The answer is *NO*. The PointLight node itself perfectly knows if it is on or off and we could have asked it each time the script node receives a "lightIsActive" Event. The next example exactly does this. However to show the usefulness of state variables we now count the number of times a user has switched on the light. After 10 trials we will turn on another light that will be lit permanently with the hope that this will discourage the person from playing too much with light switches.

Example 5.3.5 *The Light-on problem with VRML state)*

VRML:	../examples/anim/anim-touch-state2.wrl
Source:	../examples/anim/anim-touch-state2.text
Contents:	JavaScript, TouchSensor, VRML state

In order to do this we "USE" again handlers to VRML nodes somewhere in the scene as shown in the two following lines of code:

```
field SFNode node USE LIGHT
field SFNode node2 USE LIGHT2
```

Instead of asking a state variable if the light is on we then go and ask the light itself:

```
if (node.on == FALSE) {....}
```

Finally we light the second light with the following instruction:

[23] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#TouchSensor

```
        if (n > 10) {
           node2.set_on = TRUE;
           }
```

Note that you can't turn it off again. Below you can study the whole function:

```
DEF lightScript Script {
   eventIn SFBool lightIsActive
   eventOut SFBool fixLightState
   field SFNode node USE LIGHT
   field SFNode node2 USE LIGHT2
   field SFInt32 n 0
   directOutput TRUE

   # Only do something in the function
   # if lightIsActive = TRUE (the active param)
   # because as soon as the user releases the mouse
   # button a FALSE is generated again.
   url ["javascript:

        }",
        "vrmlscript:
        function lightIsActive(active) {
           if (active) {
              if (node.on == FALSE) {
                 n = n+1;
                 node.set_on = TRUE;
                 if (n > 10) {
                   node2.set_on = TRUE;
                   }
              }
              else {
                 node.set_on = FALSE;
              }
           }
        }",
       ]
}
```

All this (I hope) did give you an idea about the "dynamics" potential of VRML. It wouldn't take much more to program a little bug that would crawl into the scene and remove the light switch for example after somebody played too much with it.

5.3.3 Touchsensors, ROUTE, Switch and Javascript

Let's assume again that you that feel that two switches (ON and OFF) for turning on and off the light is too much. You also want one switch that shows it's "state" and does both. Additionally you did not grasp how to deal with state (like I did when I built this example in spring 97).

In order to so you need to learn the =>Switch[24] node. It allows to define several definitions of the same conceptual object (a bit like the LOD node), one of which is displayed at a given time.

```
Switch {
```

[24] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Switch

```
  exposedField    SFInt32 whichChoice -1
  exposedField    MFNode  choice      []
}
```

The `WhichChoice` field specifies the index of the list of nodes in `choice` that will be displayed. Note that the first element in the list is `0`. If the index is less than 0 or bigger than than number of nodes, nothing is chosen.

The only tricky stuff (assumed that you understood the previous examples) is how to deal with the fact that all nodes under a Switch continue to receive and send events (i.e.routes) regardless of the value of whichChoice. So maybe what I did here is more or less illegal. I let you study the examples by yourself. In any case this example shows *a particularly dumb* way of programming a on/off switch but it also introduced the important switch node.

Example 5.3.6 *The Light-on problem switch the dumb way(a)*

VRML:	../examples/anim/anim-touch-4.wrl
Source:	../examples/anim/anim-touch-4.text
Contents:	JavaScript, TouchSensor, Switch Node

Example 5.3.7 *The Light-on problem switch the dumb way (b)*

VRML:	../examples/anim/anim-touch-5.wrl
Source:	../examples/anim/anim-touch-5.text
Contents:	JavaScript, TouchSensor, Switch Node

5.3.4 Scripting with TimeSensors

If you don't know the basics about =>TimeSensors[25], go and read section 5.2.1 on 54 again.

You can manipulate the following eventIns (among others):

- **set_loop** whether it should loop over cycles
- **set_enabled** whether the time sensor is enabled.

In the next simple example we will show how to start and stop some animation. You can see something sliding forth and back once you click on the little green switch and you stop it with the red switch.

Example 5.3.8 *Activating and Stopping a Shuttle*

VRML:	../examples/anim/anim-shuttle-2.wrl
Source:	../examples/anim/anim-shuttle-2.text

This example used pretty much the same interface mechanism as the light examples. However it represents an additional element in the interaction, i.e. an animation element.

[25] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#TimeSensor

Chapter 6

The External Authoring Interface

Under construction, the whole thing is rather *rough draft* like, though the examples will work with several kinds of browsers.

Examples for this chapter (and others) can be found here: http://tecfa.unige.ch/guides/vrml/examples/eai/[1] However, other examples in this tree (not used in this manual) might be broken sometimes.

This section assumes that you are a little bit familiar with Java programming. However, since our students learn JAVA with VRML we also discuss how to do certain things with Java. See section 8.5 on page 90 for a few comments about Java and setting up your environment.

Oh do you want to know, why I picked up EAI programming before doing more complicated other VRML stuff. Well we bought an SGI O2 machine for doing VRML. Since SGI does not support the Java Script node for their own hardware, nor all those fine JavaScript examples featured on their site, there was only the EAI left to play with. Fortunately the EAI is a very fine thing for a number of applications. So I am **very** angry at SGI for not supporting their own hardware (and will not replace my old Sun by an O2 but by an Ultra 5). But I am grateful to the SGI folks who did produce the EAI. I hope that some of them must whine too at Cosmo's "management" decisions. (That is on Jan 98).

6.1 What can we use the EAI for ?

Let's quote from "VRML Is- Java Does- And the EAI Can Help" ([Kimen, 1997]): "[e]ssentially, to Java, the EAI is a set of classes with methods that can be called to control the VRML world. To VRML the EAI is just another mechanism that can send and receive events, just like the rest of VRML." In more general terms, the EAI is an interface specification that allows an external program to communicate with a VRML scene. So in principle the EAI is neither tied to Java nor to a Java applet on the same HTML page, it is just in practise (for now at least). Read Chris Marrin[2]'s "Anatomy of a VRML Browser" ([Marrin, 1996]) for details on the relationship between the EAI and the rest of VRML.

Again in [Kimen, 1997] Chris Marrin who is the main author of the EAI is quoted with an example that illustrates what the EAI does:

> For instance, If Java wants to change the color of a sphere in the VRML world, it would make a call to find the sphere by asking for it by name. Then it would make a call to find the color field of the sphere. Then it would make a call to change the color of that field. In VRML, you can name any node. That is the name the Java applet is asking for. Most nodes can be sent events. When the Java applet finds the color field and changes the color of that field, the VRML world receives an event to change the color, processes that event, and voila, the sphere changes color.

[1] http://tecfa.unige.ch/guides/vrml/examples/eai/
[2] http://www.marrin.com/

Note that VRML can also talk to the EAI "on its own initiative" as we shall learn later. Everything you can do within in script node (e.g. by using Java/VrmlScript or the the Java Scripting Application Interface) you can do with the EAI. However, in many situations it is better to use the Script Interface [more later].

Here are a few typical situations where you need or should use the EAI:

- Control of interactive multimedia presentations involving more than VRML.
- Visualizations needing a 2-D interface applet that controls the VRML scene in real time (e.g. users slides a button and something grows/shrinks/moves in the scene).
- Custom VRML navigation
- Multiuser Worlds with chat windows and other networking applications that need a user interface applet. (Note however that Sony's system[3] is based upon the JSAI, but you need a special browser)

6.2 The very first steps

In this section you will learn how to:

1. Set up an HTML page with an embedded VRML scene and a EAI Java Applet.
2. How get a handle from Java to the VRML browser.
3. How to print out messages to the Java console from the Applet (in order to trace errors, understand program flow etc.)

6.2.1 The application: HTML plus VRML plus a Java Applet

As we said before the EAI is designed to interface an external program with a VRML Scene. Typically you would first create an HTML page containing:

1. A VRML scene with a least a single DEFed Group or Transform Node
2. A Java Applet that controls the scene.

A typical template would look like this:

```
<HTML>
  <HEAD>
    <TITLE>Typical HTML EAI/VRML/JAVA file</TITLE>

  </HEAD>
  <BODY>
    <H1>Typical HTML EAI/VRML/JAVA file</H1>

    <!-- VRML plugged-in scene -->
    <embed src="vrml-scene.wrl" border=0 height="250" width="375">

    <!-- EAI Java applet -->
    <applet code="MyEAIapplet.class" mayscript height="200" width=500">
    </applet>

  </BODY>
```

[3] http://sonypic.com/vs/

```
</HTML>
```

Note, that you can copy this template[4] for your own use. Also make sure to include the `MAYSCRIPT` parameter in the `<applet>` tag. Finally, if you never have seen any EAI applet you can browse though our examples[5] if you wish.

The *Java* classes you need are all in the `vrml.external` package. Make sure that you *do not* include the `vrml`, `vrml.node` and `vrml.field` packages in the Java Classpath. These are reserved for the scripting interface and will *conflict.* [6] Therefore a typical applet skeleton would look like this:

```
import java.awt.*;
import java.applet.*;
import vrml.external.*;

public class MyEAIapplet extends Applet {
   .......
    public void init () {
     .....
    }
  }
```

See the next section (6.2.2) for a complete minimal example.

Writing simple Java applets Again, this section is not a fine introduction to Java programming. In case you are learning Java along with EAI just remember this:

- The name of the applet class ("MyEAIapplet" here) has to be the same as the file name ("MyEAIapplet.java"), else it will not be found.
- When the applet class is loaded into your Browser, it will first call the init() method to make any one-off initializations. There exists one per default, but you may as well redefine it to initialize things properly before the applets starts running. (e.g. getting references to VRML scene and painting Java widgets).
- Usually an applet also has method definitions for `start()`, `destroy()` and `stop()` as well as `action()` (user input handling).... more later.
- It is good practise to declare all global variables at the beginning of the code and give them *meaningful* self-documenting names.

6.2.2 Getting a reference to the VRML browser

In this section you will learn:

1. how to get a reference to the VRML browser from the Java applet,
2. how to print out messages to the Java console of your HTML browser (for debugging purposes).

For everything you want to do you need to establish communication with the browser. In order to do so you first declare a variable to hold an instance of the browser object in your Java applet:

[4] http://tecfa.unige.ch/vrml/templates/eai/eai-file.html.text
[5] http://tecfa.unige.ch/guides/vrml/examples/eai/
[6] Note however that all methods in the Browser Script Interface are available to an EAI application. The interface is the same as for Script node scripts. Any node reference obtained with the getNode( ) method can be used in these methods.

```
Browser browser;
```

We then can use the `getBrowser` method of the `Browser` class to get a browser reference:

```
browser = Browser.getBrowser(this);
```

Now let's look at a complete applet that does just about this and nothing much else:

Example 6.2.1 *EAI: The do nothing applet*

Contents:	getBrowser(),
HTML page:	eai/getbrowser/get-browser.html
JAVA code:	eai/getbrowser/get-browser.java
(dir):	(dir)

```
// Example applet illustrating getting a browser reference

import java.awt.*;
import java.applet.*;
import vrml.external.*;

public class MyEAIapplet extends Applet {

  Browser browser = null;

  public void init() {

    // Paint something to the applet so that you can see something :)
    add (new Label ("This is the Java Applet with a "));
    add (new Button ("No nothing button"));

    // get the Browser
    browser = Browser.getBrowser(this);

    // Test if we really got and print a message to the Java Console
    if (browser == null) {
      System.out.println("FATAL ERROR! no browser :( ");
    }
    System.out.println("Got the browser: "+browser);
    // Now we could do something with what we got
  }
}
```

Note that `getBrowser()` is a static method that ought to work with most modern browsers. In some older code that floats around or in tricky situations (e.g. finding an embedded something within a table I believe) you can find the following alternative:

```
import netscape.javascript.JSObject;
....
JSObject win = JSObject.getWindow(this);
JSObject doc = (JSObject) win.getMember("document");
JSObject embeds = (JSObject) doc.getMember("embeds");
browser = (Browser) embeds.getSlot(0);
```

We never used this Netscape/Lifewire specific method.

Feedback/tracing messages for the developer: If you start playing around with Java programming (and not only then) it is useful to print out messages to the Java Console (which the end user usually does not enable). The two following lines will exactly do that:

```
System.out.println("FATAL ERROR! no browser :( ");
System.out.println("Got the browser: "+browser);
```

Java Widgets: The example's last piece of code we need to discuss relates to the "user interface":

```
// Paint something to the applet so that you can see something :)
add (new Label ("This is the Java Applet with a "));
add (new Button ("No nothing button"));
```

Those two lines add a label and a button. Note that it is a good idea to start learning some basic widget programming if you haven't already done so. However we will explain somewhat the simple widgets we use in the other examples of this chapter.

Trouble with getBrowser(): Chances are there that the above will not work (even if you did install a recent VRML browser and did compile the classes as you should have). One reason might be that your applet needs more than one trial in order to get the browser reference. Instead of the code above you will have to to use something like this:

```
Browser browser = null;
    for (int count = 0; count < 10; count++) {
         browser = Browser.getBrowser (this);
         if (browser != null) break;
       try {Thread.sleep (200);}
         catch (InterruptedException ignored) {}
         System.out.println ("browser was null, trying again");     }
     if (browser == null) {
         throw new Error ("Failed to get the browser after 10 tries!");     }
     System.out.println ("Got browser\n");
```

This template[7] is also available on-line. Now let's start for real !

6.3 Essential EAI tricks

Now you already know how to get a reference (handle) to the browser. This is a necessary step in any situation. In the next subsections we will discuss how to do the following:

1. Getting a reference to a VRML node: Not necessary in any case, we will use this technique in most our introductory examples. Once you have a handle on a VRML node you can inspect and manipulate its Event fields or even do the same for the children nodes. (See section 6.3.1.)
2. Writing and reading values to the Event fields of a VRML node. (See sections 6.3.1, 6.3.2, 6.3.3).
3. Generating VRML from strings, replacing scenes, adding and removing nodes. (See sections 6.3.4 and 6.3.5)
4. Receiving Events from the Scene (call-back from VRML). (see section 6.3.6.)

Those basic operations are illustrated in the figure 6.1

[7] http://tecfa.unige.ch/vrml/templates/eai/GetBrowserApplet2.java

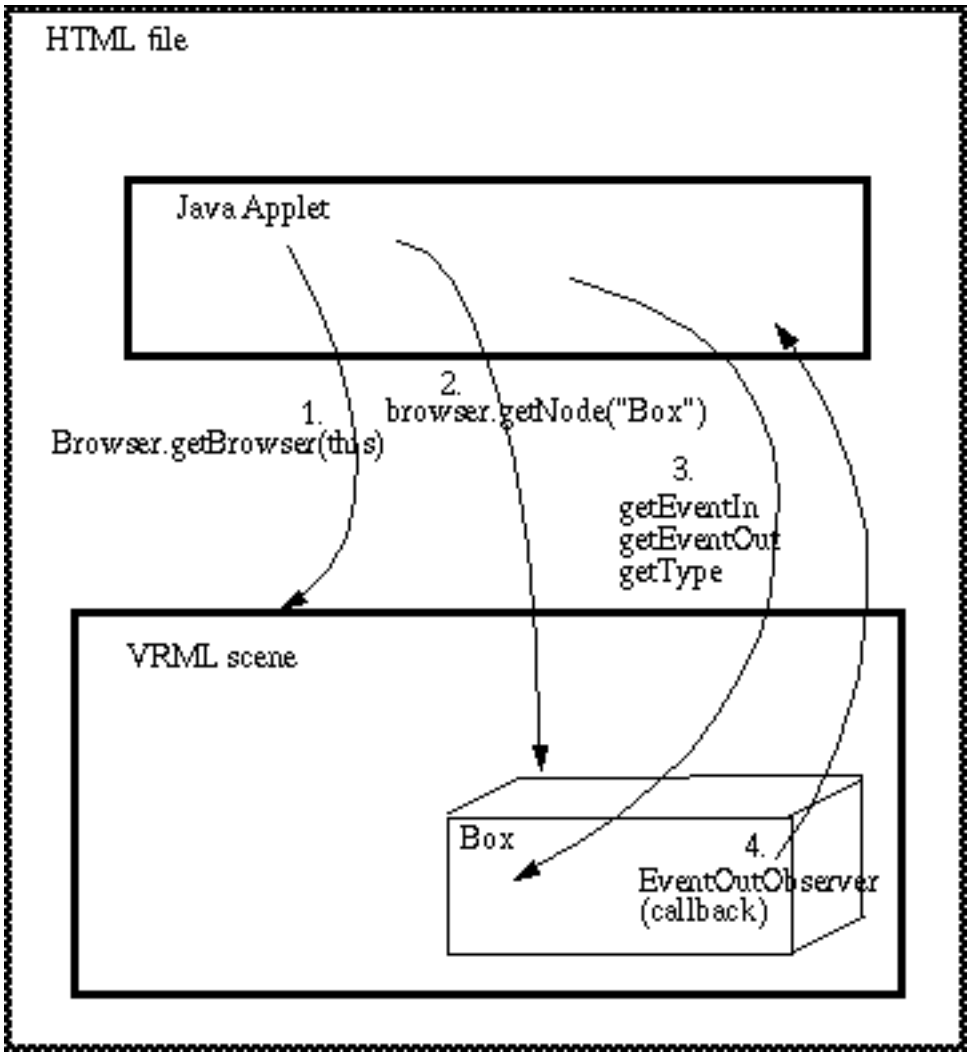

Figure 6.1: EAI: Basic JAVA methods

6.3.1 Getting a reference to a VRML node and writing Event values

For operations such as reading or sending information to a VRML node you first will need a *reference* to this node. In this subsection we will teach you first how to get a reference *to a named node that exists* in the VRML scene, a reference to its EventIn field we want to set and some elementary Java widget programming.

Let's look first at the following simple VRML scene. It will display a red ball. Now we want to be able to change its color with a few simple buttons from the Java applet.

```
DEF Ball Transform {
   children [
      Shape {
         appearance Appearance {
            material DEF MAT Material {
               diffuseColor 0.8 0.2 0.2
            } }
         geometry Sphere {}
      }
     ] }
```

In order to do so, we must first get a reference to the node which defines the color, i.e. `DEF MAT Material {...}`. Next, we need a reference to the `set_diffuseColor` field. Finally, we we must be able to set a value to this EventIn field. The following example just does this:

Example 6.3.1 *EAI: RGB Change Applet*

Contents:	getNode(), getEventIn()
HTML page:	eai/rgb-change/rgb-change.html
JAVA Code:	eai/rgb-change/RGBChange.java
Directory:	(dir)

Here is the relevant code in the Java applet for getting the node and field references:

- The `browser.getNode("MAT")` method will get a reference to the material node which we called "MAT" in the VRML scene.
- The `material.getEventIn("set_diffuseColor")` method will do the same for the eventIn field.

```
.....
Node material = null;
EventInSFColor diffuseColor = null;
......

 public void init() {
  ........
  // Get the material node...
  material = browser.getNode("MAT");
  // Get the diffuseColor EventIn
  diffuseColor = (EventInSFColor) material.getEventIn("set_diffuseColor");
  .......
}
```

Java programming comments: Java Newbies need some more explanation (others can skip this): The reference to the VRML browser is hold by a variable we called "browser". It has been initialized of type "Browser". Browser is a class that defines the "getNode" method allowing to retrieve named nodes from the VRML scene. In a similar way we assigned the result of

`browser.getNode("MAT")` to the `material` variable we declared as type *Node*. Let's just show you the definition of the "Node" class from the EAI Spec[8] (in Java code format):

```
public Node          getNode(String name)
     throws InvalidNodeException;
```

This means that getNode (a) wants as argument a node name (the string of a DEF), (b) returns an instance of a Node class (i.e. the reference to a node) unless it fails in which case it signals an InvalidNodeException. More later about this latter case.

Now the "Node" class defines a method called `getEvent` that will return an instance of the EventIn class. However you must cast (i.e. translate) this instance to the appropriate class you are interested in. The reason is that `getEventIn` can return different types of values. In our example we cast to `(EventInSFColor)` in the following line:

```
diffuseColor = (EventInSFColor) material.getEventIn("set_diffuseColor");
```

You have to do this everytime you deal with some event fields. The EAI Spec is pretty self explaining about those types. So far, we only have the handles on VRML things we want to manipulate. In order to obtain what we want we now must proceed with some ordinary Java coding. If you look at the Java code and the applet you can see that we painted three simple buttons:

```
// paint 3 simple Java buttons
 add(new Button("Red"));
 add(new Button("Green"));
 add(new Button("Blue"));
```

The Applet class which is imported from `java.applet.*` is a child of the java.awt.Component[9] class which defines a simple action[10] event method. This method is activated (i.e. called) with the Event and the Object triggered when the user clicks on a button. The programmer now has to deal with the event, i.e. implement an "action" method for this applet. Since the "action" method is depreciated in Java 1.1. we will not explain much here, but revise the code once most Netscape and VRML plug-in users will have versions that fully support Java 1.1.

Java Event handling and writing to the VRML scene: Let's just look at the VRML related part for dealing with the Blue button:

```
float[] color = new float[3];
  .......
public boolean action(Event event, Object what) {
  // handle Java user actions
  if (event.target instanceof Button)
    {
      Button b = (Button) event.target;
      ..........
      color[0] = 0.2f; color[1] = 0.2f; color[2] = 0.8f;
      diffuseColor.setValue(color);
      ......
    }
  return true;
 }
```

[8] http://www.vrml.org/WorkingGroups/vrml-eai/ExternalInterface.html
[9] /guides/java/jdk/docs/api/java.awt.Component.html
[10] /guides/java/jdk/docs/api/java.awt.Component.html#action(java.awt.Event, java.lang.Object)

In the prologue code of the class (see just above) we did declare a "color" variable of type array that can take take three floating point elements. All we have to do after identifying which button has been clicked on is the following:

1. assign values to the color array:
2. write the value to VRML scene with the `setValue()` method. This method is defined by the EAI's `EventInSFColor` class.

The "setValue" method of the "EventInSFColor" class wants an array argument i.e. the Java equivalent of the VRML SFColor triple. The `action()` method is called whenever the user does something and the first lines of codes will identify events generated by clicking on a button.

Let's summarize what you have learned:

1. Getting a node reference with Browser.getNode()
2. Getting a reference to an EventIN field with Node.getEventIn()
3. Setting a value of an event field with setValue()

6.3.2 Getting a reference to a VRML node and writing Event values II

Now let's make our Applet a bit more bullet proof. The `getNode` and `getEventIn` method will throw exceptions that we could handle, e.g. by giving some feedback to the user and disabling Java code that depends on references not found:

Example 6.3.2 *EAI: RGB Change Test Applet*

Contents:	getNode(), getEventIn(), Java: try/catch
HTML page:	eai/rgb-change/rgb-change-test.html
JAVA Code:	eai/rgb-change/RGBChangeTest.java
Directory:	(dir)

The following code will implement what we do when references are not found. Java's "try" statement allows us to define "catch" clauses that are called by the Java interpreter when something is wrong. (Recall the definition of the the getNode discussed in the previous section 6.3.1).

```
try {
  // Get the material node...
  material = browser.getNode("MAT");
  // Get the diffuseColor EventIn
  diffuseColor = (EventInSFColor) material.getEventIn("set_diffuseColor");
}
catch (InvalidNodeException ne) {
  add(new TextField("Failed to get node:" + ne));
  error = true;
}
catch (InvalidEventInException ee) {
  add(new TextField("Failed to get EventIn:" + ee));
  error = true;
}
```

Lines like:

```
  add(new TextField("Failed to get node:" + ne));
```

will paint a text field on the Java Applet with the error message we defined. (Note, that we could just have written something to the Java Console instead.) In addition, the catch statements will set the "error" variable to true. If true the action method will not try to handle any events, i.e. it will just exit. Below is the the code (in old Java 1.0.x style) that whill handle the event.

```
public boolean action(Event event, Object what) {
    if (error)
      {
        showStatus("Problems! Had an error during initialization");
        return true;  // Uh oh...
      }
     .......
```

Finally note the usage of the "showStatus" method which will print a message on the status bar (the bottom) of your WWW client window. If you now want to see how this works play with the following example which shows a Java Applet where "MAT" it misspelled as "MATS" in the code:

Example 6.3.3 *EAI: RGB Change Test Error Applet with an error*

Contents:	getNode(), getEventIn(), Java: try/catch
HTML page:	eai/rgb-change/rgb-change-err.html
JAVA Code:	eai/rgb-change/RGBChangeErr.java
Directory:	(dir)

6.3.3 Reading Events from the Scene

The next example deals with reading information from a VMRL scene. Note that this is not the same as receiving information (the equivalent of VRML's ROUTEing) which we shall introduce later.

The scenario of our next examples deals with a (very) simple "where I am" problem. The user can click on a button an it will tell him some information about the EntryView Point.

Example 6.3.4 *EAI: Get Node Info Demo*

Contents:	getEventOut(),
HTML page:	eai/get-node-info/get-node-info-s.html
JAVA code:	eai/get-node-info/GetNodeInfoS.java
Directory:	(dir)

Retrieving Information from a VRML scene is very much the same as writing information, something we learned in the previous section 6.3.1. We also need a handle to an Event field. In this case we will read from *EventOut* fields. Remember that: Event**Out** and not just the field. Therefore ordinary VRML exposed fields can be accessed by something like `<handle>.getEventOut("<field_name>_changed")` as in the lines below:

```
EventOutSFVec3f positionVP = null;
EventOutSFRotation orientationVP = null;
......
positionVP = (EventOutSFVec3f) entryVP.getEventOut("position_changed");
orientationVP = (EventOutSFRotation) entryVP.getEventOut("orientation_changed");
```

Once we got the handle we can read. In the action method of our example we get the position and the orientation of a Viewpoint with the following lines of code:

```
currentPosition = positionVP.getValue();
currentOrientation = orientationVP.getValue();
```

Writing into a Java text field: Again some comments for Java newbies: The following defines a text widget to which we can print text:

```
TextArea output = null;
.....
public void init() {
  ......
  output = new TextArea(5, 40);
  add(output);
  ......
 }

 public boolean action(Event event, Object what) {
  .......
      output.appendText("Entry ViewPoint is at" +
                        " x=" + currentPosition [0] +
                        " y=" + currentPosition [1] +
                        " z=" + currentPosition [2] + "\n");
   .....
}
```

Such a thing is definitively better than telling the user to open up the Java Console.

Error handling again: The next example does exactly the same, but includes error handling. Of new interest is only the "InvalidEventOutException":

```
public void initScene() {
  try {
  .......
    positionVP = (EventOutSFVec3f) entryVP.getEventOut("position_changed");
    orientationVP = (EventOutSFRotation) entryVP.getEventOut("orientation_changed");
  }
  .........
  catch (InvalidEventOutException ee) {
    System.out.println("InvalidEventOutException: " + ee);
    die("initScene: EventOut Field not found!");
    return;
  }   }
```

Also note that that a "die" method has been defined that is called when encountering an error. This adds some better modularization to the code.

Finally you can look at a minimal implementation of the "start", "stop" and "destroy" methods that are called by the Java interpreter.

Example 6.3.5 *EAI: Get Node Info Demo with Error Handling*

Contents:	getNode(), getEventOut(), Java: try/catch
HTML page:	eai/get-node-info/get-node-info.html
JAVA code:	eai/get-node-info/GetNodeInfo.java
Directory:	(dir)

6.3.4 Create Vrml from String

In this subsection you will learn how to create a VRML scene from a string and to replace the existing scene in the browser. Let's start with an empty world, i.e. all there is in the *.wrl file is a line that will get the plug-in going:

```
#VRML V2.0 utf8
```

The following example will paint a blue ball as soon as the applet starts up:

Example 6.3.6 *EAI: Creating and inserting a simple VRML tree*

Contents:	createVrmlFromString()
HTML page:	eai/create/create-ex0.html
JAVA code:	eai/create/CreateEx0.java
Directory:	(dir)

A Vrml scene (or tree) is represented in the Java Applet as as an array of Nodes which we will initialize as follows:

```
Node[] scene = null;
```

The Browser class' `createVrmlFromString` method will take a simple string containing ordinary VRML statements and then create an array of VRML nodes from it. In this example we just add simple strings together, but of course you also could produce strings from variables and methods to program some more useful applet. The code below will just create a few VRML objects.

```
scene = browser.createVrmlFromString(
"DEF Camera Viewpoint {\n" +
     "  position 0 0 5}\n" +
"DEF MySphere Transform {\n" +
     "   children [ \n" +
 ......
     "     ] \n" +
     " } \n"
     );
```

Once the scene array contains VRML nodes (represented as Java instances) the rest is very simple. We can replace the current (in this case empty) VRML scene by the nodes in the new scene array with the `replaceWorld` method.

```
browser.replaceWorld(scene);
```

Note that we also could have added these nodes to a children field of an existing scene without replacing everything else. With what you already learned you ought to be able to set a value to a "addChildren" field of a grouping node, but we will demonstrate it anyhow in section 6.3.5.

The next example is more complex, because it will separately create a series of VRML nodes and then assemble them together by setting respective EventIn fields. If you do not understand what it does you should read again the previous sections or just move on and come back later.

Example 6.3.7 *EAI: Creating and assembling a VRML tree*

Contents:	createVrmlFromString()
HTML page:	eai/create/create-ex.html
JAVA code:	eai/create/CreateEx.java
Directory:	(dir)

A more sophisticated example from which the above has been inspired can be found in the EAI specification. A local copy is below. You can study it yourself :)

Example 6.3.8 *EAI: Creating and assembling a VRML tree*

Contents:	createVrmlFromString()
HTML page:	eai/create-test/create-test.html
JAVA code:	eai/create-test/CreateTest.java
Directory:	(dir)

6.3.5 Create, Put and Remove together

In this subsection we pull together most of what we have learned so far. This applet will allow the user to add a (VrmlFromString created) ball to an exiting Group node and also to remove it. The Vrml we originally start with is very simple:

```
#VRML V2.0 utf8

DEF Camera Viewpoint {
    position 0 0 7
}
DEF ROOT Group {}
```

As you might remember from section 4.4 on page 44, nodes like =>Group[11] and Transform define the EventIn `addChildren` and the EventOut `removeChildren` allowing you to add and remove objects from children. We will make use of this feature in the next example.

Example 6.3.9 *EAI: Add and remove an object*

Contents:	
HTML page:	eai/add-remove/add-remove1.html
JAVA code:	eai/add-remove/AddRemove.java
Directory:	(dir)

If you understood the previous subsections there is not much explaining needed. Look first at these lines in the `init()` method:

```
// Get root node of the scene
 root = browser.getNode("ROOT");

 // Instantiate (get handle to) the EventIn objects
 addChildren = (EventInMFNode) root.getEventIn("addChildren");
 removeChildren = (EventInMFNode) root.getEventIn("removeChildren");
```

Note that `root.getEventIn("addChildren")` and `root.getEventIn("removeChildren")` produce `EventInMFNodes`.

In the `action()` method we then simply `setValue()` the node we want to remove or to add. Note that shape is an array that holds the result of the `createVrmlFromString()` operation.

```
if (b == addButton) {
    addChildren.setValue(shape);
  }
else if (b == removeButton) {
    removeChildren.setValue(shape);
  }
```

This example is just a simplified version of the one found in the EAI spec. The original adds some error checking and feedback and you can look at it in the example below. Also, if the example above won't work on your browser, the one below will print out the reasons for you :)

Example 6.3.10 *EAI: Add and remove test*

Contents:	
HTML page:	eai/add-remove-test/AddRemoveTest.html
JAVA code:	eai/add-remove-test/AddRemoveTest.java
Directory:	(dir)

[11] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#Group

6.3.6 Receiving Events from the Scene

The last element we consider basic and essential in the EAI Spec is receiving Events from the Scene which as bit analogous to the ROUTE statement used with animated VRML and scripting. It is not the same as reading something from a scene which was discussed for example in subsection 6.3.3.

In the example we will discuss a simple TouchSensor example. When the user clicks on the cube it will change color (something that usually would not need the EAI).

Example 6.3.11 *EAI: Monitor and act upon a touch*

Contents:	EventOutObserver
HTML page:	eai/monitor-test/monitor-touch.html
JAVA code:	eai/monitor-test/MonitorTouch.java
Directory:	(dir)

So what's new ? Our applet in order to receive call-backs from the VRML scene needs to implement an `EventOutObserver` Interface. It can be done like this:

```
public class MonitorTouch extends Applet implements EventOutObserver
```

Of course you also could define a subclass implementing `EventOutObserver` but it would be slightly more complicated. In any case, you must write a `callback` method for the `EventOutObserver` class in order to process events. This method takes three arguments: the EventOut, the time and some user definable data. Let's look at the code:

```
public void callback(EventOut who, double when, Object which) {

  // retrieve the state of isActive (see above the advise function)
  EventOutSFBool state = (EventOutSFBool) which;

  // only deal with the event if isActive was true, else the ball would flip back
  if (state.getValue() == true)  {
    System.out.println("callback(): EventOut=" + who
                       + " state =" + state.getValue());
    // Change the color and remember it
    if (colorState == 1) {
      diffuseColor.setValue(redColor);
      colorState = 0;
    }
    else {
      diffuseColor.setValue(greenColor);
      colorState = 1;
    }
  }
}
```

In our case we do not need to deal with the EventOut object nor with the time stamp. The which variable contains any kind of Java Object that we decided to pass over from the VRML scene. In this example we passed `isActive`, i.e. an `EventOutSFBool` object related to the TouchSensor. To set up the call-back we had to add the following lines to the `init()` method:

```
....
 // Get the Touch Sensor
 Node sensor = browser.getNode("TOUCH");
 // Get its touchTime EventOut
 isActive = (EventOutSFBool) sensor.getEventOut("isActive");
```

```
// Set up the callback
isActive.advise(this, isActive);
....
```

As you can see, EventOuts do have an advise method that is used to tell (advise) the class, that this EventOut will be used for a call-back. Let's go back to the `callback()` method listed above. If you remember how to deal with state and isActive Events from a TouchSensor (see section 5.3.2 on page 61) you know that isActive will both generate true and false events depending on whether the user clicks or releases a mouse button. Here again, we are just interesting in the "down" click:

```
EventOutSFBool state = (EventOutSFBool) which;
// only deal with the event if isActive was true ...
if (state.getValue() == true)  { .... }
```

The rest is fairly easy. If the ball is green (remembered within our Java code as `colorState`) we paint it red, else green. The following line should be meaningful to you, else go back to subsection 6.3.1.

```
diffuseColor.setValue(redColor);
```

Voila, that's it. We have written an invisible applet that does something we have been doing before with Javascript (or that could be done with the Java Script node).

Other examples: The following example will monitor the output of a TimeSensor Node and the rotation field of a spinning cube.

Example 6.3.12 *EAI: Monitor a Rotation*

Contents:	EventOutObserver
HTML page:	eai/monitor-test/monitor-test2.html
JAVA code:	eai/monitor-test/MonitorTest2.java
Directory:	(dir)

The VRML scene we are dealing with is slightly more complex than the previous ones. In case you forgot all your basic animated VRML you can go back reading section 5.2.1 on 54. It just links the pulse of a TimeSensor to an OrientationInterpolator which sets the rotation field of a Transform node containing a cube.

On the Java Side there is not much more than in the previous example. Note that the advise methods simply passes an Integer as a way to identify the Event we have to handle. There are also 2 buttons to stop and start the animation, some error handling code, and a long loop that makes sure that we really get the browser. [I have a feeling that one should put the EAI/Java inits into the start() method to be sure that the VRML scene has enough time to fully load ... to be tested].

Finally you can now look at the RGB Test Applet from the EAI Spec. It does about everything you have learned so far.

Example 6.3.13 *EAI: RGB Test Applet*

Contents:	getEventIn, getEventOut, EventOutObserver
HTML page:	eai/rgb/RGB-demo.html
JAVA code:	eai/rgb/RGBTest.java
Directory:	(dir)

In the next sections we will discuss some slightly more complex examples and introduce some more EAI stuff.

6.4 Extracting Information from a scene

So far you have learned that you can access any node in a existing VRML scene that has a DEFed name. You can actually do more. Since you can inspect exposed field values, you can also retrieve children nodes. We are going to demo this in the next example. It will retrieve all the Viewpoints from a Group node called "Viewpoints" in a VRML scene. The Java applet then generates navigation nodes inserted into the applet. Note that this example is still pretty useless. Useful things have a tendency to become quite complex.

Example 6.4.1 *EAI: Retrieving children nodes*

Contents:	getEventOut, children nodes
HTML page:	eai/navigate/navigate-dyn.html
JAVA code:	eai/navigate/NavigateDyn.java
Directory:	(dir)

```
DEF Viewpoints Group {
   children [
      DEF Entry Viewpoint {
         position       0 1 15
         description    "Entry"
      }

      DEF AcrossTheRoom Viewpoint {
    .............
      }

    .............
     ]
}
```

The `init()` method is fairly standard, so we will not comment it. Now let's see how we exact these nodes: We first declare an array variable to store the nodes:

```
  Node[] VPList ;
```

We extract the children in the `initScene()` method:

```
  // Get the Group node called Viewpoints
  Node Viewpoints = browser.getNode("Viewpoints");
  // Extract the list of children
  VPList = ((EventOutMFNode)
       (Viewpoints.getEventOut("children_changed"))).getValue();
```

Note how we cast the result of `getEventOut` to `EventOutMFNode`, i.e. it will return a *list* of Nodes. And that's it. Now in order to be able to rebind the user to a Viewpoint, we are interested in getting handles on the `set_bind` field for each found viewpoint. The following lines of code relate to that:

```
   nVPs = VPList.length;
   set_VPList = new EventInSFBool[nVPs];
   ......
     for (i=0; i < nVPs ; i++) {

        // Get the handles for the Viewpoints set_bind Event fields

        set_VPList[i] = (EventInSFBool)VPList[i].getEventIn("set_bind");
      }
```

In the same loop we also generate the Java press buttons which we also store in an array (look at the source code) All we finally need to do is to implement the `action()` method that will handle user events and bind the user to a new ViewPoint. It's very much the same technology we discussed in section 6.3.1 as well as others. The only difference is that buttons are not stored in variables but in an array called VPButtons:

```
  /** Handle actions from AWT widgets */
  public boolean action(Event event, Object what) {

    if (event.target instanceof Button) {
      Button b = (Button) event.target;

      for (int i=0; i < nVPs ; i++) {
        if (b == VPButtons[i]) {
          // get the corresponding VP
          EventInSFBool set_bindVP = set_VPList[i];
          set_bindVP.setValue(true);
          System.out.println("action(): set_bound " + set_bindVP);
        }  }
    }
    return true;  }
}
```

This example assumed that the only children of the "Viewpoints" node were Viewpoints. In other words, our code would break if there were other types of children nodes.

The next example deals with this issue. The applet below adds a "tour" button that will move the user throught all viewpoints with some pause in between (Jumping is not nice I know, but this will be addressed somewhere else).

Example 6.4.2 *EAI: Retrieving children nodes with errors*

Contents:	getEventOut(), getType(), children nodes
HTML page:	eai/navigate/navigate-tour.html
JAVA code:	eai/navigate/NavigateDyn.java
Directory:	(dir)

Basically we just had to add a few lines to the `initScene()` method. The `Node.getType()` methode will return a string with the VRML Node name which you then can use to identify the kind of node you have retrieved.

```
      for (int i=0; i < nVPCs ; i++) {
        String type_found = VPCandidates[i].getType();
        if (!type_found.equals("Viewpoint")) {
          System.out.println("WARNING! initScene: Ignored node of type " + type_found);
        }
        else {
          // got a ViewPoint
          VPList[nVPs] = VPCandidates[i];
          nVPs = nVPs+1;
        }
      }
```

We will also add here some comments about a revisited version of the Tiny3D applet that extracts node definitions]

Example 6.4.3 *EAI: Tiny3d-2*

Java 1.1.x (Netscape 4x PATCHED):	eai/tiny3D-2/tiny3D-2.html
Java 1.0.x (Netscape 3x,4x):	eai/tiny3D-2-java1.0.2/tiny3D-2.html

6.5 Object manipulation and better Java Widgets

In this section we will address some issues on how to control objects from the Applet.
[REALLY under construction]

6.5.1 My first slider applet

The first Java applet Daniel ever wrote was the next one, so we leave it here, despite the fact that it uses depreciated Java 1.0x technology. You can change the sizes and position of a nice green ball.

Example 6.5.1 *Ball Grow*

Contents:	getEventIn, Java: sliders
HTML page:	eai/ballgrow/ball-grow.html
JAVA code:	eai/ballgrow/BallGrow.java
Directory:	(dir)

You will not learn any new EAI technology, but you can see how to program a simple use of sliders with Java 1.0x. In order to set up the layout we use Panels and GridLayouts[12]. GridLayOuts allow you to simply put Widgets into a Grid and their size will be adjusted to the size of the largest elements. Code on the EAI side should be familiar now:

```
public void init () {
  ......
  // Get the reference to the set_scale event
  set_scale = (EventInSFVec3f) root.getEventIn("scale");

  // Get the reference to the set_translation event
  set_translation = (EventInSFVec3f) root.getEventIn("translation");
  .......
 }

public boolean handleEvent (Event event) {

  if (event.target instanceof Scrollbar) {
  ......
  val[0] = ((float) transx.getValue()) / 10.0f;
  val[1] = ((float) transy.getValue()) / 10.0f;
  val[2] = ((float) transz.getValue()) / 10.0f;
  .........
  set_scale.setValue(val);
  .....
  set_translation.setValue(val);
}
```

In the same directory you will find more sophisticated versions of this applet, illustrating some more features and also politically better Java programming.

[12] /guides/java/jdk/docs/api/java.awt.GridLayout.html

Chapter 7

Animation and Avatars

[rough stuff, under construction since March 9 / 98]

This chapter requires mastering of topics discussed in chapter 5 on page 49.

7.1 Introduction to keyframe animation

VRML provides a set of tools for keyframe animations. An animation sequence can be understood as a cycle with a beginning (the first frame) and and end (the last frame) and everything in between. In order to build keyframe animations you define a set of so-called keyframes, i.e. specific critical moments in an animation cycle.

You then use interpolator nodes (or custom made script nodes) to compute all the intermediary positions that are necessary for a smooth animation in function of *the fraction* of the animation cycle computed. Practically this means that you use the output of the *fraction_changed* filed of =>TimeSensor[1] as input for an interpolator engine. Note that the length of the animation is just defined by the TimeSensor's *cycleTime* field. In section 5.2 we gave a short introduction to this technique.

Keyframes are usually (and simply put) virtual definitions of some VRML geometry that you want to animate, e.g. position, shape, color of some group of objects. But you can "animate" whatever exposed *fields* or *eventIns* of any node you wish.

There exist a variety of builtin =>Interpolator nodes that cover some common *linear* animations needs:

1. The =>ColorInterpolator
2. The =>CoordinateInterpolator
3. The =>NormalInterpolator
4. The =>OrientationInterpolator
5. The =>PositionInterpolator
6. The =>ScalarInterpolator

All these interpolators work the same way, i.e. they share a common set of fields:

```
eventIn      SFFloat      set_fraction
exposedField MFFloat      key           [...]
exposedField MF<type>     keyValue      [...]
eventOut     [S|M]F<type> value_changed
```

[1] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#TimeSensor

The *set_fraction* eventIn receives an SFFload event that can be any kind of floating point number. In practise it is usually a number between 0 and 1 produced by a TimeSensor's *fraction_changed* field. Now, for each "critical" fraction of the whole animation cycle you add the fraction to the array of *key*s and for each key you must specify a interpolator-specific *keyValue*. Finally, the interpolator will produce the interpolated *value_changed* eventOuts (as many as your implementation can handle).

The principle of keyframe animation is very simple. What is not so simple are for example:

- Multiple synchronized animations, e.g. a body that walks, moves legs, swings arms. etc. If we are not wrong, it seems that for creating animations in a reasonable amount of time you need a VRML authoring tool.
- Non-linear animations, e.g. shuttle flying around in VRML space. Note that you could break it down such a non-linear flight path to a large number of liner interpolation.
- Coordination of animations with interactive user input. E.g. have an avatar jump while it runs.

[... more to come]

7.2 Building a mini bot

[under construction ... more to come]

Example 7.2.1 *The MiniBot*

Contents:	Mini Avatar (under construction)
Directory:	(dir)

Chapter 8

Appendix

8.1 On-line Tools

The Tiny-3d-2 Applet This applet will allow you to create simple VRML geometry, color, size and move around it. It also allows you to generate VRML code that you can paste into your editor.

Java 1.1.x (Netscape 4/Jx):	eai/tiny3D-2/tiny3D-2.html
Java 1.0.x (Netscape 3/4x):	eai/tiny3D-2-java1.0.2/tiny3D-2.html

DegressToRadian Converter You can use this JavaScript degree/radian converter[1] to figure out rotations.

The Color Applet

Color Appplet:	eai/color/

8.2 VRML Tools

This short section is NOT an introduction to such tools, rather a place where we (maybe) will write down some experiences we had.

8.2.1 CosmoWorlds

CosmoWorlds for Irix is quite a powerful tool, but it does have its limitations. I (Daniel Schneider) would recommend to buy this tool for Irix since it's the only one available. Probably CW 2 for Windows has less limitations (but I haven't seen that yet).

VRML code produced is overally speaking rather clean. It does add a few extra useless Transforms, Groups and fields. The only major gripe I have is the placement of animation nodes.

- Documentation is only in HTML format (difficult to print) and therefore writing code while looking at the documentation on single monitor is difficult (at best)
- The Script editor will not let you do certain things, like adding "USE xx" or values to fields, or "directOutput TRUE"
- The OutLine editor will not let you move or delete nodes and the general tool is VERY restrictive about moving nodes. The worst thing is that you can't move animation nodes outside of the geometry it animates (correct me if I am wrong). This leads to very unreadable code.

[1] http://tecfa.unige.ch/vrml/tools/rad-convert.html

- The tool will not leave certain field values alone if you don't touch them (is that a bug or feature for optimized rendering if you get 0.000000something instead of a plain 0 in a translation?

- You can edit the source to clean things up, but be VERY careful. Moving around animations makes them unavailable for later editing (even if the tool does not complain when you read the file back).

- Hiding unselected objects does not work and is a pain ! (correct me if I am wrong).

- The is no support for protos, although you can hand edit the file to add them (but be careful, save a copy and make a test before you clean up all your code).

- When you import a file with an error you will not get any serious error message, it will just break and say so :(

8.3 VRML II Browsers

See also section 1.1.5 on 13 for more conceptual issues! [maybe I should move this section there]. Products mentionned in this section can be found from our VRML page (http://tecfa.unige.ch/guides/vrml/pointers.html).

Currently (Jan 98) the browser situation is very good if you own a Win95 machine. It's not great if you are the proud owner of a SGI workstation or a Mac and it's downright bad if you happen to own a Sun or Linux or whatever else box. Here are a few tips. Note that information on where to find browsers is on our VRML pointers[2] page.

8.3.1 Win95

In March 97 I preferred the Cosmo Player. Note that text nodes don't are not implemented. Use Sony's Community Place for that. In order to switch between plugins, get Sony's plug-in chooser. Those two browsers will certainly rapidly try to match the full VRML 2 standard, but other contenders may appear too.

Since April 97 I am using Worldview Beta4. It's the faster browser around and seems to implement most everything. Colors may not be as pretty, but I don't care right now.

Since Jan 98 we are back to Cosmo for general work and use Sony's and Blaxxun's browsers for multi-user things.

8.3.2 VRML on old SUNs

If you have an old SUN (like a standard Sparc5, 10 or 20) they all come with a simple 256 color card. No full VRML 2 browsers will currently (March 97) work with those, except VrWave which certainly sometimes in 98 will become a truly VRML 97 compliant browser.

Consider buying a PC. It's cheaper than getting a decent graphics card for your old Sun and for just viewing VRML, Win95 will do.

8.3.3 PowerMac

Currently (Jan 98), the best VRML 2 browser I could find is WorldView (but it lacks Java support).

In order to look at **standard VRML 2** worlds you must give your browser more memory ! (before you run Netscape: "Get Info" in the Finder's "File" menu).

[2] http://tecfa.unige.ch/guides/vrml/pointers.html

8.3.4 SGI/Irix

As of Jan 98 the news is that SGI decided not to favor development for its own hardware, so you are stuck with last year's Cosmo Player. It will lack support of Java scripting, and won't recognize Javascript if you don't call it VrmlScript. Send them angry mails and decide to learn the EAI which is a kewl interface for doing many interesting things. Use a PC for scripting Script nodes in the meantime (though nobody can tell if we EVER will get Cosmo 2 on Irix).

Here is a clue about Java Problems found at http://help.netscape.com/kb/client/970907-1.html

```
To troubleshoot your Java applets in UNIX environment,
you can start Communicator with the "-java" switch.
 #netscape -java
 Usage: -java [-options] class

 where options include:
    -help             print out this message
    -version          print out the build version
    -v -verbose       turn on verbose mode
    -debug            enable remote JAVA debugging
    -noasyncgc        don't allow asynchronous garbage collection
    -verbosegc        print a message when garbage collection occurs
    -noclassgc        disable class garbage collection
    -cs -checksource  check if source is newer when loading classes
    -ss<number>       set the maximum native stack size for any thread
    -oss<number>      set the maximum Java stack size for any thread
    -ms<number>       set the initial Java heap size
    -mx<number>       set the maximum Java heap size
    -D<name>=<value>  set a system property
    -classpath <directories separated by ':'>
                      list directories in which to look for classes
    -prof[:<file>]    output profiling data to ./java.prof or ./<file>
    -verify           verify all classes when read in
    -verifyremote     verify classes read in over the network [default]
    -noverify         do not verify any class
```

8.4 Field and Event Reference

This section describes the syntax and general semantics of fields and events, the elemental data types used by VRML nodes to define objects. It has been taken with minor modifications from the =>Field and Event Reference[3]. Nodes are composed of fields and events and defined types used by both fields and events.

There are two general classes of fields and events; fields/events that contain a single value (where a value may be a single number, a vector, or even an image), and fields/events that contain multiple values. Single-valued fields/events have names that begin with **SF**. Multiple-valued fields/events have names that begin with **MF**.

Multiple-valued fields/events are written as a series of values enclosed in square brackets, and separated by awhitespace (e.g. commas). If the field or event has zero values then only the square brackets ("[]") are written. The last value may optionally be followed by whitespace (e.g. comma). If the field has exactly one value, the brackets may be omitted and just the value written. For example, all of the following are valid for a multiple-valued MFInt32 field named *foo* containing the single integer value 1:

```
foo 1
```

[3] http://tecfa.unige.ch/guides/vrml/vrml97/spec/part1/fieldsRef.html

```
foo [1,]
foo [ 1 ]
```

SFBool A field or event containing a single boolean value. SFBools are written as TRUE or FALSE. For example,

```
fooBool FALSE
```

is an SFBool field, *fooBool*, defining a FALSE value.

The initial value of an SFBool eventOut is FALSE.

SFColor/MFColor SFColor specifies one RGB (red-green-blue) color triple, and MFColor specifies zero or more RGB triples. Each color is written to file as an RGB triple of floating point numbers in ANSI C floating point format, in the range 0.0 to 1.0. For example:

```
fooColor [ 1.0 0. 0.0, 0 1 0, 0 0 1 ]
```

is an MFColor field, *fooColor*, containing the three primary colors red, green, and blue.

The initial value of an SFColor eventOut is (0 0 0). The initial value of an MFColor eventOut is [ ].

SFFloat/MFFloat SFFloat specifies one single-precision floating point number, and MFFloat specifies zero or more single-precision floating point numbers. SFFloats and MFFloats are written to file in ANSI C floating point format. For example:

```
fooFloat [ 3.1415926, 12.5e-3, .0001 ]
```

is an MFFloat field, *fooFloat*, containing three floating point values values.

The initial value of an SFFloat eventOut is 0.0. The initial value of an MFFloat eventOut is [ ].

SFImage The SFImage field or event defines a single uncompressed 2-dimensional pixel image. SFImage fields and events are written to file as three integers representing the width, height and number of components in the image, followed by width*height hexadecimal values representing the pixels in the image, separated by whitespace:

```
fooImage <width> <height> <num components> <pixels values>
```

A one-component image specifies one-byte hexadecimal values representing the intensity of the image. For example, 0xFF is full intensity, 0x00 is no intensity. A two-component image puts the intensity in the first (high) byte and the alpha (opacity) in the second (low) byte. Pixels in a three-component image have the red component in the first (high) byte, followed by the green and blue components (0xFF0000 is red). Four-component images put the alpha byte after red/green/blue (0x0000FF80 is semi-transparent blue). A value of 0x00 is completely transparent, 0xFF is completely opaque.

Each pixel is read as a single unsigned number. For example, a 3-component pixel with value 0x0000FF may also be written as 0xFF or 255 (decimal). Pixels are specified from left to right, bottom to top. The first hexadecimal value is the lower left pixel and the last value is the upper right pixel.

For example,

```
fooImage 1 2 1 0xFF 0x00
```

is a 1 pixel wide by 2 pixel high one-component (i.e. greyscale) image, with the bottom pixel white and the top pixel black. And:

```
fooImage 2 4 3 0xFF0000 0xFF00 0 0 0 0 0xFFFFFF 0xFFFF00
               # red    green  black.. white    yellow
```

is a 2 pixel wide by 4 pixel high RGB image, with the bottom left pixel red, the bottom right pixel green, the two middle rows of pixels black, the top left pixel white, and the top right pixel yellow.

The initial value of an SFImage eventOut is (0 0 0).

SFInt32/MFInt32 The SFInt32 field and event specifies one 32-bit integer, and the MFInt32 field and event specifies zero or more 32-bit integers. SFInt32 and MFInt32 fields and events are written to file as an integer in decimal or hexadecimal (beginning with '0x') format. For example:

```
fooInt32 [ 17, -0xE20, -518820 ]
```

is an MFInt32 field containing three values.

The initial value of an SFInt32 eventOut is 0. The initial value of an MFInt32 eventOut is [ ].

SFNode/MFNode The SFNode field and event specifies a VRML node, and the MFNode field and event specifies zero or more nodes. The following example illustrates valid syntax for an MFNode field, *fooNode*, defining four nodes:

```
fooNode [ Transform { translation 1 0 0 }
          DEF CUBE Box { }
          USE CUBE
          USE SOME_OTHER_NODE ]
```

The SFNode and MFNode fields and events may contain the keyword NULL to indicate that it is empty.

The initial value of an SFNode eventOut is NULL. The initial value of an MFNode eventOut is [ ].

SFRotation/MFRotation The SFRotation field and event specifies one arbitrary rotation, and the MFRotation field and event specifies zero or more arbitrary rotations. S/MFRotations are written to file as four floating point values separated by whitespace. The first three values specify a normalized rotation axis vector about which the rotation takes place. The fourth value specifies the amount of right-handed rotation about that axis, in radians. For example, an SFRotation containing a 180 degree rotation about the Y axis is:

```
fooRot 0.0 1.0 0.0  3.14159265
```

The initial value of an SFRotation eventOut is (0 0 1 0). The initial value of an MFRotation eventOut is [ ].

SFString/MFString The SFString and MFString fields and events contain strings formatted with the UTF-8 universal character set

ISO/IEC 10646-1:1993[4] SFString specifies a single string, and the MFString specifies zero or more strings. Strings are written to file as a sequence of UTF-8 octets enclosed in double quotes (e.g. "string").

Any characters (including newlines and '#') may appear within the quotes. To include a double quote character within the string, precede it with a backslash. To include a backslash character within the string, type two backslashes. For example:

```
fooString [ "One, Two, Three", "He said, \\"Immel did it!\\"" ]
```

is a MFString field, *fooString*, with two valid strings.

The initial value of an SFString eventOut is "". The initial value of an MFRotation eventOut is [ ].

[4] http://www.iso.ch/cate/d18741.html

SFTime/MFTime The SFTIme field and event specifies a single time value, and the MFTime field and event specifies zero or more time values. Time values are written to file as a double-precision floating point number in ANSI C floating point format. Time values are specified as the number of seconds from a specific time origin. Typically, SFTime fields and events represent the number of seconds since Jan 1, 1970, 00:00:00 GMT.

The initial value of an SFTime eventOut is -1. The initial value of an MFTime eventOut is [ ].

SFVec2f/MFVec2f An SFVec2f field or event specifies a two-dimensional vector. A MFVec2f field or event specifies zero or more two-dimensional vectors separated by commas. Each vector contains floating point values separated by whitespaces. For example:

```
fooVec2f [ 42 666, 7 94 ]
```

is a MFVec2f field, named *fooVec2f*, with two valid vectors.

The initial value of an SFVec2f eventOut is (0 0). The initial value of an MFVec2f eventOut is [ ].

SFVec3f/MFVec3f An SFVec3f field or event specifies a three-dimensional vector. A MFVec3f field or event specifies zero or more three-dimensional vectors. For example:

```
fooVec3f [ 1 42 666, 7 94 0 ]
```

is a MFVec3f field, named *fooVec3f*, with two valid vectors.

The initial value of an SFVec3f eventOut is (0 0 0). The initial value of an MFVec3f eventOut is [ ].

8.5 Introduction to JAVA with VRML

Under construction !

Setting up Java and VRML on your machine Just one word for now. Working with the Java and VRML (both the internal and external) authoring interface can be quite an ugly experience right now. It somewhat contradicts all Java propaganda you may have heard from SUN about portability. Depending on what software roams in the inside of your Win95 it may take several few hours to get you going. For some stuff you'll need *days*.

If you know some french check out /guides/vrml/java/vrml_top.html[5] produced by Sylvere[6]. It may help you going, e.g. tells you how to deal with autoexec.bat and various incompatible Java/VRML libraries.

Else, please check out the Tutorials Section of our VRML Pointers Index[7]. We do collect pointers to all help we find on the Net.

In any case it is probably best *not do deal with locally run Java Applets*. Put your stuff on a WWW Server. If you don't want people to look at it, just protect it. On most Unix servers you only need to copy an `.htaccess` file to your directory, i.e. just grab one from another directory.

Learning Java: Learning JAVA is a tough thing if you never did any kind of "real" programming.

I usually recommend learning Javascript first since it is very handy for VRMLScripting anyhow. See our JavaScript Page[8] for the most important pointers. If you didn't know it before: Javascript and Java are two very different languages. JavaScript an object-based scripting language that is quite easy to learn. Note that there are many variants, e.g. Javascript for Netscape integrates

[5] /guides/vrml/java/vrml_top.html
[6] http://tecfa.unige.ch/tecfa/general/tecfa-people/sylvere.html
[7] http://tecfa.unige.ch/guides/vrml/pointers.html
[8] http://tecfa.unige.ch/guides/js/pointers.html

with HTML and the Netscape Browser. VRML plug-in versions of Javascript allow you to script VRML nodes.

Java is an object-oriented programming language that is difficult to learn. To our students (who usually never did any real programming) I recommend to study [Bishop, 1997]. This book also teaches about programming (you need to go through about half of the book before starting to play with VRML classes).

If you know some programming, you can "learn by example" (as I am doing it), but you ought to have at least 1-2 Java Tutorial books at hand if you never did any C or C++ programming. I was and am suffering from the following problems:

1. Java's syntax is awfully ugly (some instructions really look meaningless) compared to other modern well designed languages. Note that ugliness was a design choice in order get this language accepted by C programmers. Once you know that a line like:

   ```
   public void init( )
   ```

   starts defining a method that returns nothing you are catching on :)

2. Java is very faschist about typing. E.g. a boolean "VRML" variable is not the same as the "ordinary" boolean. Hmm I was so naive about inheritance. When you work with Java VRML classes you will do a *lot* of translating (casting) back and forth.

3. Java was hailed as a small and elegant language. Quite true in principle, but useless propaganda again for practical use, because in reality you have to learn a lot of libararies. It will take you weeks to master those. A lot of them are VERY badly documented if you take into account all the enthousiasm about Java).

   So: Java is *not* Java, but a particilar version of Java which may or may run with your browser PLUS zillions of library classes that you will have to master.

You may consult our See our Java Page[9] for some important pointers.

8.6 Useless stuff (Version History / ToDo List)

8.6.1 Version history

Very roughly ! Also note that sometimes, contents can change on an hourly basis, bad internal links mean that the document is being processed ... just wait a few minutes and try again.

1. 0.1 - Spring 1996 written in ONE day, i.e. they after I learned a little VRML I so please be careful!,
2. 0.2 - Spring 1996 minor corrections
3. 0.3 - Beg Feb 1997 Conversion to VRML 2
4. 0.4 - Mid Feb 1997 better structuring of contents
5. 0.5 - End Feb 1997 Some interactive VRML, also added appendix (fields and toto list)
6. 0.6 - March 2 1997 Minor corrections and additions
7. 0.7 - March 12 1997 Minor corrections, replaced example theorem macro (badly handled by the HTML translator) with tables. Discovered Intervista's WorldView and fixed Java(vrml)script examples.
8. 0.8 - March 17 1997 Started working on VRML generation

[9] http://tecfa.unige.ch/guides/java/pointers.html

9. 0.9 - April 18 1997 Started cleaning up the section on moving interactive VRML
10. 0.10a - Jan 29 1998 Started a cleaning action again (including moving examples files into different places).
11. 0.10b - Rough draft section on the EAI
12. 0.10c - Draft section intro to the EAI, style sheet for netscape 4 users
13. 1.0 - March 5 98 - First "official" version. Means that most stuff should be useful to the audience (not perfect!).
14. 1.1a - March 12 98 - Fixed Tiny3D-2 applets (somewhat). Added Color Demonstration applet. Fixed a few typos. Started a chapter on animation and avatars.

8.6.2 Things to do

Just a short list of what you can expect in some nearer future.

Little things:

- Introduction on "how to use"
- A "framed" page with the list of examples to the left and VRML to the right.
- Fix English
- External Protos (!)
- Add a few JavaScript scripting examples that manipulate SF and MF type fields.

Medium things:

- More on color (Build a color tool that shows in operation all the Material node parameters with a few different light sources).
- Lights
- Viewpoint angles and positioning (Build a tool ?)
- Complex objects (non-simple shapes)
- tricks to make your scene look good

Big things:

- An introduction to humanoid avatars (REAL SOON!)
- Good scenes with simple moving objects
- Good scenes with Javascript
- Scripts with Java (Introduction)
- Multi-user worlds

8.7 Copyright Information

Bibliography

[Ames et al., 1996a] Ames, A. L., Nadeau, D. R., et Moreland, J. L. (1996a). *The VRML 2.0 Sourcebook*. Wiley, New York. URL: http://www.wiley.com/compbooks/catalog/07/16507-7.html (source code and ordering information).

[Ames et al., 1996b] Ames, A. L., Nadeau, D. R., et Moreland, J. L. (1996b). *The VRML Sourcebook*. Wiley, New York. URL: http://www.wiley.com/Compbooks/vrmlsrbk/cover/cover.html (source code and ordering information).

[Bishop, 1997] Bishop, J. (1997). *Java Gently, Programming Principles Explained*. Addison-Wesley, Reading. URL: http://www.cs.up.ac.za/javagently.

[Carey et Bell, 1997] Carey, R. et Bell, G. (1997). *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, Reading (MA). URL: http://www.awl.com/devpress/titles/41974.html.

[Gibson, 1994] Gibson, W. (1994). *Neuromancer*. Ace, New York.

[Hartman et Wernecke, 1996] Hartman, J. et Wernecke, J. (1996). *The VRML 2.0 Handbook, Building Moving Worlds on the Web*. Addison Wesley, Reading (MA). URL: http://vrml.sgi.com/handbook/index.html.

[Hughes, 1995] Hughes, K. (1995). From Webspace to Cyberspace. EIT WWW publications. URL: http://www.eit.com/%7Ekevinh/cspace/ or http://www.scit.wlv.ac.uk/kevinh/ (UK Mirror).

[Kimen, 1997] Kimen, S. (1997). VRML Is- Java Does- And the EAI Can Help. On-line article at vrml.sgi.com. URL: http://vrml.sgi.com/features/java/java.html.

[Lea et al., 1996] Lea, R., Matsuda, K., et Miyashita, K. (1996). *Java for 3D and VRML Worlds*. New Riders, Indianapolis. URL: http://www.mcp.com/newriders/books/1-56205-689-1.html.

[Marrin, 1996] Marrin, C. (1996). Anatomy of a VRML Browser. on-line publication. URL: http://www.marrin.com/vrml/Interface.html.

[Pesce, 1995] Pesce, M. (1995). *VRML, Browsing and Building Cyberspace*. New Riders, Indianapolis. URL: http://www.mcp.com/newriders/internet/vrml/ (source code, ordering information and patches).

[Pesce et al., 1994] Pesce, M. D., Kennard, P., et S., P. A. (1994). Cyberspace. In *Proceedings of The First International Conference on The World-Wide Web*. URL: http://www1.cern.ch/PapersWWW94/mpesce.ps.

[Roehl et al., 1997] Roehl, B., Couch, J., Reed-Ballreich, C., Rohaly, T., et Brown, G. (1997). *Late Night VRML 2.0 with Java*. Ziff-Davis Press, Emeryville. URL: http://ece.uwaterloo.ca:80/%7Ebroehl/vrml/lnvj/index.html.

[Stephenson, 1992] Stephenson, N. (1992). *Snow Crash*. Bantam.

Index