EUROGRAPHICS 97

Introduction to VRML 97

Lecturer

David R. Nadeau
nadeau@sdsc.edu
http://www.sdsc.edu/~nadeau
San Diego Supercomputer Center

Tutorial notes sections

Abstract
Preface
Lecturer biography
Using the VRML examples
Using the JavaScript examples
Using the Java examples
Tutorial slides
NetscapeWorld article reprints
Abstract

VRML (the Virtual Reality Modeling Language) has emerged as the de facto standard for describing 3-D shapes and scenery on the World Wide Web. VRML’s technology has very broad applicability, including web-based entertainment, distributed visualization, 3-D user interfaces to remote web resources, 3-D collaborative environments, interactive simulations for education, virtual museums, virtual retail spaces, and more. VRML is a key technology shaping the future of the web.

Participants in this tutorial will learn how to use VRML 97 (a.k.a. ISO VRML, VRML 2.0, and Moving Worlds) to author their own 3-D virtual worlds on the World Wide Web. Participants will learn VRML concepts and terminology, and be introduced to VRML’s text format syntax. Participants also will learn tips and techniques for increasing performance and realism. The tutorial includes numerous VRML examples and information on where to find out more about VRML features and use.
Welcome to the EUROGRAPHICS 97 Introduction to VRML 97 tutorial notes! These tutorial notes have been written to give you a quick, practical, example-driven overview of VRML 97, the Web’s Virtual Reality Modeling Language. To do this, I’ve included over 500 pages of tutorial material with nearly 200 images and over 100 VRML examples.

To use these tutorial notes you will need an HTML Web browser with support for viewing VRML worlds. An up to date list of available VRML browsing and authoring software is available at:

The VRML Repository
(http://www.sdsc.edu/vrml)

What’s included in these notes

These tutorial notes primarily contain three types of information:

1. General information, such as this preface
2. Tutorial slides and examples
3. Article reprints from NetscapeWorld magazine

The tutorial slides are arranged as a sequence of 400+ hyper-linked pages containing VRML syntax notes, VRML usage comments, or images of sample VRML worlds. Clicking on a sample world’s image, or the file name underneath it, loads the VRML world into your browser for you to examine yourself.

You can view the text for any of the VRML worlds using a text editor and see how we created a particular effect. In most cases, the VRML files contain extensive comments providing information about the techniques the file illustrates.

The tutorial notes provide a necessarily terse overview of VRML. A more detailed introduction to the basic features of VRML is provided in four article reprints courtesy NetscapeWorld magazine. The articles do not cover all of VRML. I recommend that you invest in one of the VRML books on the market to get thorough coverage of the language. I am a co-author of one such VRML book, The VRML 2.0 Sourcebook. Several other good VRML books are on the market as well.

A word about VRML versions

VRML has evolved through several versions of the language, starting way back in late 1994. These tutorial notes cover VRML 97, the latest version of the language. To provide context, the following table provides a quick overview of these VRML versions and the names they have become known by.
<table>
<thead>
<tr>
<th>Version</th>
<th>Released</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRML 1.0</td>
<td>May 1995</td>
<td>Begun in late 1994, the first version of VRML was largely based upon the Open Inventor file format developed by Silicon Graphics Inc. The VRML 1.0 specification was completed in May 1995 and included support for shape building, lighting, and texturing. VRML 1.0 browser plug-ins became widely available by late 1995, though few ever supported the full range of features defined by the VRML 1.0 specification.</td>
</tr>
<tr>
<td>VRML 1.0c</td>
<td>January 1996</td>
<td>As vendors began producing VRML 1.0 browsers, a number of ambiguities in the VRML 1.0 specification surfaced. These problems were corrected in a new VRML 1.0c (clarified) specification released in January 1996. No new features were added to the language in VRML 1.0c.</td>
</tr>
<tr>
<td>VRML 1.1</td>
<td>canceled</td>
<td>In late 1995, discussion began on extensions to the VRML 1.0 specification. These extensions were intended to address language features that made browser implementation difficult or inefficient. The extended language was tentatively dubbed VRML 1.1. These enhancements were later dropped in favor of forging ahead on VRML 2.0 instead. No VRML 1.1 browsers exist.</td>
</tr>
<tr>
<td>Moving Worlds</td>
<td>January 1996</td>
<td>VRML 1.0 included features for building static, unchanging worlds suitable for architectural walk-throughs and some scientific visualization applications. To extend the language to support animation and interaction, the VRML architecture group made a call for proposals for a language redesign. Silicon Graphics, Netscape, and others worked together to create the Moving Worlds proposal, submitted in January 1996. That proposal was later accepted and became the starting point for developing VRML 2.0. The final VRML 2.0 language specification is still sometimes referred to as the Moving Worlds specification, though it differs significantly from the original Moving Worlds proposal.</td>
</tr>
<tr>
<td>VRML 2.0</td>
<td>August 1996</td>
<td>After seven months of intense effort by the VRML community, the Moving Worlds proposal evolved to become the final VRML 2.0 specification, released in August 1996. The new specification redesigned the VRML syntax and added an extensive set of new features for shape building, animation, interaction, sound, fog, backgrounds, and language extensions. Beta versions of VRML 2.0 browser plug-ins have been available since late 1997. However, as of this writing (May 1997) there are still no fully-compliant, complete VRML 2.0 browsers available on</td>
</tr>
</tbody>
</table>
In early 1997, efforts got under way to present the VRML 2.0 specification to the International Standards Organization (ISO) which oversees most of the major language specifications in use in the computing community. The ISO version of VRML 2.0 was reviewed and the specification significantly rewritten to clarify issues. A few minor changes to the language were also made. The final ISO VRML was dubbed VRML 97. The VRML 97 specification features finalized in March 1997, while the specification’s text finalized in September 1997.

One beta version of a VRML 97 browser plug-in is available as of this writing: Silicon Graphics Cosmo Player for SGI platforms. More VRML 97 compliant browsers are expected within the next few months.

VRML 1.0 and VRML 2.0 differ radically in syntax and features. A VRML 1.0 browser cannot display VRML 2.0 worlds. Most VRML 2.0 browsers, however, can display VRML 1.0 worlds.

VRML 97 differs in a few minor ways from VRML 2.0. In most cases, a VRML 2.0 browser will be able to correctly display VRML 97 files. However, for 100% accuracy, you should have a VRML 97 compliant browser for viewing the VRML files contained within these tutorial notes.

How I created these tutorial notes

These tutorial notes were developed primarily on Silicon Graphics High Impact UNIX workstations. HTML and VRML text was hand-authored using a text editor. A Perl program script was used to process raw tutorial notes text to produce the 400+ individual HTML files, one per tutorial slide.

HTML text was displayed using Netscape Navigator 3.01 on Silicon Graphics and PC systems. Colors were checked for viewability in 24-bit, 16-bit, and 8-bit display modes on a PC. Text sizes were chosen for viewability at a normal 12 point font on-screen, and at an 18 point font for presentation during the Eurographics 97 tutorial. The large text, white-on-black colors, and terse language are used to insure that slides are readable when displayed for the tutorial audience at the Eurographics 97 conference.

VRML worlds were displayed on Silicon Graphics systems using the Silicon Graphics Cosmo Player 1.02 VRML 97 compliant browser for Netscape Navigator. The same worlds were displayed on PC systems using three different VRML 2.0 compliant browsers for Netscape Navigator: Silicon Graphics Cosmo Player 1.0 beta 3a, Intervista WorldView 2.0, and Newfire Torch alpha 3.

Texture images were created using Adobe PhotoShop 4.0 on a PC with help from KAI’s PowerTools 3.0 from MetaTools. Image processing was also performed using the Image Tools suite of applications for UNIX workstations from the San Diego Supercomputer Center.
PDF tutorial notes for printing by Eurographics 97 were created by dumping individual tutorial slides to PostScript on a Silicon Graphics workstation. The PostScript was transferred to a PC where it was converted to PDF and assembled into a single PDF file using Adobe’s Distiller and Exchange.

Use of these tutorial notes

I am often asked if there are any restrictions on use of these tutorial notes. The answer is:

These tutorial notes are copyright (c) 1997 by David R. Nadeau. Users and possessors of these tutorial notes are hereby granted a nonexclusive, royalty-free copyright and design patent license to use this material in individual applications. License is not granted for commercial resale, in whole or in part, without prior written permission from the authors. This material is provided "AS IS" without express or implied warranty of any kind.

You are free to use these tutorial notes in whole or in part to help you teach your own VRML tutorial. You may translate these notes into other languages and you may post copies of these notes on your own Web site, as long as the above copyright notice is included as well. You may not, however, sell these tutorial notes for profit or include them on a CD-ROM or other media product without written permission.

If you use these tutorial notes, I ask that you:

1. Give me credit for the original material
2. Tell me since I like hearing about the use of my material!

If you find bugs in the notes, please tell me. I have worked hard to try and make the notes bug-free, but if something slipped by, I’d like to fix it before others are confused by my mistake.

Contact

David R. Nadeau
San Diego Supercomputer Center
P.O. Box 85608
San Diego, CA 92186-9784

UPS, Fed Ex: 10100 Hopkins Dr.
La Jolla, CA 92093-0505

(619) 534-5062
FAX: (619) 534-5152

nadeau@sdsc.edu
Lecturer biography

David R. Nadeau

Mr. Nadeau is a principal scientist at the San Diego Supercomputer Center (SDSC), specializing in scientific visualization and virtual reality. He is an author of technical papers on graphics and VRML, a co-author of two books on VRML (*The VRML Sourcebook*, and *The VRML 2.0 Sourcebook*), and authors the bi-monthly *VRML Technique* column for NetscapeWorld magazine. He has taught VRML courses at conferences including SIGGRAPH 96, WebNet 96, VRML 97, and SIGGRAPH 97, and is the creator of *The VRML Repository*, a principal Web site for information on VRML software and documentation. Mr. Nadeau co-chaired VRML 95, the first conference on VRML, and the *VRML Behavior Workshop*, the first workshop on behavior support for VRML. He is SDSC’s representative in the *VRML Consortium*. 
Introduction to VRML 97

Using the VRML examples

These tutorial notes include over a hundred VRML files. Almost all of the provided worlds are linked to from the tutorial slides pages.

VRML support

As noted in the preface to these tutorial notes, this tutorial covers VRML 97, the ISO standard version of VRML 2.0. There are only minor differences between VRML 97 and VRML 2.0, so any VRML 97 or VRML 2.0 browser should be able to view any of the VRML worlds contained within these tutorial notes.

The VRML 97 (and VRML 2.0) language specifications are complex and filled with powerful features for VRML content authors. Unfortunately, the richness of the language makes development of a robust VRML browser difficult. As of this writing, there are nearly a dozen VRML browsers on the market, but none support all features in VRML 97 (despite press releases to the contrary).

I am reasonably confident that all VRML examples in these tutorial notes are correct, though of course I could have missed something. Chances are that if one of the VRML examples doesn’t look right, the problem is with your VRML browser and not with the example. It’s a good idea to read carefully the release notes for your browser to see what features it does and does not support. It’s also a good idea to regularly check your VRML browser vendor’s Web site for updates. The industry is moving very fast and often produces new browser releases every month or so.

As of this writing, I have found that Silicon Graphics (SGI) Cosmo Player for SGI UNIX workstations is the most complete and robust VRML 97 browser available. It is this browser that I used for most of my VRML testing. On the PC, I found found that Intervista’s WorldView was the most complete and robust browser available, though it still had a number of flaws and unsupported features. On the Macintosh and non-SGI UNIX workstations, I was unable to find a usable VRML browser with which to test the VRML tutorial examples.

What if my VRML browser doesn’t support a VRML feature?

If your VRML browser doesn’t support a particular VRML 97 feature, then those worlds that use the feature will not load properly. Some VRML browsers display an error window when they encounter an unsupported feature. Other browsers silently ignore features they do not support yet.

When your VRML browser encounters an unsupported feature, it may elect to reject the entire VRML file, or it may load only those parts of the world that it understands. When only part of a VRML file is loaded, those portions of the world that depend upon the unsupported features will display incorrectly. Shapes may be in the wrong position, have the wrong size, be shaded incorrectly, or have the wrong texture colors. Animations may not run, sounds may not play, and interactions may not work correctly.
For most worlds I have captured an image of the world and placed it on the tutorial slide page to give you an idea of what the world should look like. If your VRML browser’s display doesn’t look like the picture, chances are the browser is missing support for one or more features used by the world. Alternately, the browser may simply have a bug or two.

In general, VRML worlds later in the tutorial use features that are harder for vendors to implement than those features used earlier in the tutorial. So, VRML worlds at the end of the tutorial are more likely to fail to load properly than VRML worlds early in the tutorial.
Using the JavaScript examples

These tutorial notes include several VRML worlds that use JavaScript program scripts within `Script` nodes. The text for these program scripts is included directly within the `Script` node within the VRML file.

JavaScript support

The VRML 97 specification does not require that a VRML browser support the use of JavaScript to create program scripts for `Script` nodes. Fortunately, most VRML browsers do support JavaScript program scripts, though you should check your VRML browser’s release notes to be sure it is JavaScript-enabled.

Some VRML browsers, particularly those from Silicon Graphics, support a derivative of JavaScript called `VRMLscript`. The language is essentially identical to JavaScript. Because of Silicon Graphics’ strength in the VRML market, most VRML browser vendors have modified their VRML browsers to support `VRMLscript` as well as JavaScript.

JavaScript and `VRMLscript` program scripts are included as a text within the `url` field of a `Script` node. To indicate the program script’s language, the field value starts with either "javascript:" for JavaScript, or "vrmlscript:" for `VRMLscript`, like this:

```javascript
Script {
  field SFFloat bounceHeight 1.0
  eventIn SFFloat set_fraction
  eventOut SFVec3f value_changed
  url "vrmlscript:
    function set_fraction( frac, tm ) {
      y = 4.0 * bounceHeight * frac * (1.0 - frac);
      value_changed[0] = 0.0;
      value_changed[1] = y;
      value_changed[2] = 0.0;
    }
  }
```

For compatibility with Silicon Graphics VRML browsers, all JavaScript program script examples in these notes are tagged as "vrmlscript:”, like the above example. If you have a VRML browser that does not support `VRMLscript`, but does support JavaScript, then you can convert our examples to JavaScript simply by changing the tag "vrmlscript:" to "javascript:" like this:

```javascript
Script {
  field SFFloat bounceHeight 1.0
  eventIn SFFloat set_fraction
  eventOut SFVec3f value_changed
  url "javascript:
    function set_fraction( frac, tm ) {
      y = 4.0 * bounceHeight * frac * (1.0 - frac);
      value_changed[0] = 0.0;
      value_changed[1] = y;
    }
  }
```
value_changed[2] = 0.0;

"}
}

What if my VRML browser doesn’t support JavaScript?

If your VRML browser doesn’t support JavaScript or VRMLscript, then those worlds that use these languages will produce an error when loaded into your VRML browser. This is unfortunate since JavaScript or VRMLscript is an essential feature that all VRML browsers should support. I recommend that you consider getting a different VRML browser.

If you can’t get another VRML browser right now, there are only a few VRML worlds in these tutorial notes that you will not be able to view. Those worlds are contained as examples in the following tutorial sections:

- Introducing script use
- Writing program scripts with JavaScript
- Creating new node types

So, if you don’t have a VRML browser with JavaScript or VRMLscript support, just skip the above sections and everything will be fine.
These tutorial notes include a few VRML worlds that use Java program scripts within `Script` nodes. The text for these program scripts is included in files with `.java` file name extensions. Before use, you will need to compile these Java program scripts to Java byte-code contained in files with `.class` file name extensions.

**Java support**

The VRML 97 specification does not require that a VRML browser support the use of Java to create program scripts for `Script` nodes. Fortunately, most VRML browsers do support Java program scripts, though you should check your VRML browser’s release notes to be sure it is Java-enabled.

In principle, all Java-enabled VRML browsers identically support the VRML Java API as documented in the VRML 97 specification. Similarly, in principle, a compiled Java program script using the VRML Java API can be executed on any type of computer within any brand of VRML browser.

In practice, neither of these ideal cases occurs. The Java language is supported somewhat differently on different platforms, particularly as the community transitions from Java 1.0 to Java 1.1 and beyond. Additionally, the VRML Java API is implemented somewhat differently by different VRML browsers, making it difficult to insure that a compiled Java class file will work for all VRML browsers available now and in the future.

Because of Java incompatibilities observed with current VRML browsers, I have elected to not include compiled Java class files in these tutorial notes. Instead, I include the uncompiled Java program scripts. Before use, you will need to compile the Java program scripts yourself on your platform with your VRML browser and your version of the Java language and support tools.

**Compiling Java**

To compile the Java examples, you will need:

- The VRML Java API class files for your VRML browser
- A Java compiler

All VRML browsers that support Java program scripts supply their own set of VRML Java API class files. Typically these are automatically installed when you install your VRML browser.

There are multiple Java compilers available for most platforms. Sun Microsystems provides the Java Development Kit (JDK) for free from its Web site at http://www.javasoft.com. The JDK includes the `javac` compiler and instructions on how to use it. Multiple commercial Java development environments are available from Microsoft, Silicon Graphics, Symantec, and others. An up to date list of available Java products is available at Gamelan’s Web site at
Once you have the VRML Java API class files and a Java compiler, you will need to compile the supplied Java files. Unfortunately, I can’t give you explicit directions on how to do this. Each platform and Java compiler is different. You’ll have to consult your software’s manuals.

Once compiled, place the .class files in the slides folder along with the other tutorial slides. Now, when you click on a VRML world using a Java program script, the class files will be automatically loaded and the example will run.

**What if my VRML browser doesn’t support Java?**

If your VRML browser doesn’t support Java, then those worlds that use these languages will produce an error when loaded into your VRML browser. This is unfortunate since Java is an essential feature that all VRML browsers should support. I recommend that you consider getting a different VRML browser.

**What if I don’t compile the Java program scripts?**

If you have a VRML browser that doesn’t support Java, or if you don’t compile the Java program scripts, those worlds that use Java will produce an error when loaded into your VRML browser. Fortunately, I have kept Java use to a minimum. In fact, Java program scripts are only used in the Writing program scripts with Java section of the tutorial slides. So, if you don’t compile the Java program scripts, then just skip the VRML examples in that section and everything will be fine.
# Table of contents

**Morning**

Part 1 - Shapes, geometry, and appearance

- Welcome!
- Introduction
- Building a VRML world
- Building primitive shapes
- Transforming shapes
- Controlling appearance with materials
- Grouping nodes
- Naming nodes
- Summary examples

Part 2 - Animation, sensors, and geometry

- Introducing animation
- Animating transforms
- Sensing viewer actions
- Building shapes out of points, lines, and faces
- Building elevation grids
- Building extruded shapes
- Controlling properties of coordinate-based geometry
- Summary examples
Afternoon

Part 3 - Textures, lights, and environment

Mapping textures
Controlling how textures are mapped
Lighting your world
Adding backgrounds
Adding fog
Adding sound
Controlling the viewpoint
Controlling navigation
Sensing the viewer
Summary examples

Part 4 - Scripts and prototypes

Controlling detail
Introducing script use
Writing program scripts with JavaScript
Writing program scripts with Java
Creating new node types
Providing information about your world
Summary examples
Miscellaneous extensions
Conclusion
## Welcome!

### Schedule for the day

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Part 1</strong></td>
<td>Shapes, geometry, appearance</td>
<td>90 minutes</td>
</tr>
<tr>
<td>Break</td>
<td></td>
<td>15 minutes</td>
</tr>
<tr>
<td><strong>Part 2</strong></td>
<td>Animation, sensors, geometry</td>
<td>105 minutes</td>
</tr>
<tr>
<td>Lunch</td>
<td></td>
<td>60 minutes</td>
</tr>
<tr>
<td><strong>Part 3</strong></td>
<td>Textures, lights, environment</td>
<td>90 minutes</td>
</tr>
<tr>
<td>Break</td>
<td></td>
<td>15 minutes</td>
</tr>
<tr>
<td><strong>Part 4</strong></td>
<td>Scripts, prototypes</td>
<td>105 minutes</td>
</tr>
</tbody>
</table>

---

Welcome!

Schedule for the day

Tutorial scope
Welcome!

Tutorial scope

- This tutorial covers *VRML 97*
  - The ISO standard revision of VRML 2.0

- You will learn:
  - VRML file structure
  - Concepts and terminology
  - Most shape building syntax
  - Most sensor and animation syntax
  - Most program scripting syntax
  - Where to find out more
What is VRML?

VRML is:

- A simple text language for describing 3-D shapes and interactive environments

VRML text files use a .wrl extension
What do I need to use VRML?

- You can view VRML files using a VRML browser:
  - A VRML helper-application
  - A VRML plug-in to an HTML browser

- You can view VRML files from your local hard disk, or from the Internet

Example

[ temple.wrl ]
**Introduction**

*How can VRML be used on a Web page?*

- Load directly into a Web browser, filling the page
  [boxes.wrl]

- Embed into a page, filling a page rectangle
  [boxes1.htm]

- Load into a page frame, filling the frame
  [boxes2.htm]

- Embed into a page frame, filling a frame rectangle
  [boxes3.htm]

- Embed multiple times into a page or frame
  [boxes4.htm]

*What do I need to develop in VRML?*

- You can construct VRML files using:
  - A text editor
  - A world builder application
  - A shape generator
  - A modeler and format converter
Introduction

Should I use a text editor?

- **Pros:**
  - No new software to buy
  - Access to all VRML features
  - Detailed control of world efficiency

- **Cons:**
  - Hard to author complex 3D shapes
  - Requires knowledge of VRML syntax

Should I use a world builder?

- **Pros:**
  - Easy 3-D drawing user interface
  - Little need to learn VRML syntax

- **Cons:**
  - May not support all VRML features
  - May not produce most efficient VRML
Introduction

Should I use a shape generator?

- Pros:
  - Easy way to generate complex shapes
  - Fractal mountains, logos, etc.

- Cons:
  - Only suitable for narrow set of shapes
  - Best used with other software

Should I use a modeler and format converter?

- Pros:
  - Very powerful features available
  - Can make photo-realistic images too

- Cons:
  - May not support all VRML features
  - Not designed for VRML
  - One-way path from modeler into VRML
  - Easy to make shapes that are too complex
Introduction

*How do I get VRML software?*

- The VRML Repository maintains links to available software:

  http://www.sdsc.edu/vrml
VRML file structure

VRML files contain:
- The file header
- Comments - notes to yourself
- Nodes - nuggets of scene information
- Fields - node attributes you can change
- Values - attribute values
- more. . .
Building a VRML world

A sample VRML file

```vrml
#VRML V2.0 utf8
# A Cylinder
Shape {
  appearance Appearance {
    material Material {
    }
  }
  geometry Cylinder {
    height 2.0
    radius 1.5
  }
}
```

Understanding the header

- `#VRML`: File contains VRML text
- `v2.0`: Text conforms to version 2.0 syntax
- `utf8`: Text uses UTF8 character set
Understanding UTF8

- utf8 is an international character set standard
- utf8 stands for:
  - UCS (Universal Character Set) Transformation Format, 8-bit
  - Encodes 24,000+ characters for many languages
  - ASCII is a subset

Using comments

- Comments start with a number-sign (#) and extend to the end of the line

# A Cylinder
Building a VRML world

*Using nodes*

Cylinder {
  
  * Nodes describe shapes, lights, sounds, etc.*

  * Every node has:
    * A *node type* (Shape, Cylinder, etc.)
    * A pair of curly-braces
    * Zero or more fields inside the curly-braces

Cylinder {
  height 2.0
  radius 1.5
}

* Fields describe node attributes
Building a VRML world

Using fields and values

height 2.0

- Every field has:
  - A field name
  - A data type (float, int, etc.)
  - A default value

- Fields are optional and given in any order

- Default value used if field not given

Summary

- The file header gives the version and encoding

- Nodes describe scene content

- Fields and values specify node attributes
Building primitive shapes

Motivation

Shapes are the building blocks of a VRML world

- Primitive Shapes are standard building blocks:
  - Box
  - Cone
  - Cylinder
  - Sphere
  - Text

Example

Syntax: Shape
Specifying geometry
Syntax: Box
Syntax: Cone
Syntax: Cylinder
Syntax: Sphere
Syntax: Text
A sample primitive shape
A sample primitive shape
Building multiple shapes
A sample file with multiple shapes
A sample file with multiple shapes
Syntax: FontStyle
Syntax: FontStyle
Summary
Building primitive shapes

Example

![Image](prim.wrl)

Syntax: Shape

- A `Shape` node builds a shape
- `appearance` - color and texture
- `geometry` - form, or structure

```wrl
Shape {
    appearance . . .
    geometry . . .
}
```
Building primitive shapes

**Specifying geometry**

- Shape geometry is built with *geometry* nodes:
  
  ```
  Box     { . . . }
  Cone    { . . . }
  Cylinder { . . . }
  Sphere  { . . . }
  Text    { . . . }
  ```

- Geometry node fields control dimensions
  - Dimensions usually in meters, but can be anything

**Syntax: Box**

- A `Box` geometry node builds a box

```wrl
Box {
    size 2.0 2.0 2.0
}
```

[ box.wrl ]
Building primitive shapes

**Syntax: Cone**

- A Cone geometry node builds an upright cone

```
Cone {
    height 2.0
    bottomRadius 1.0
}
[ cone.wrl ]
```

**Syntax: Cylinder**

- A Cylinder geometry node builds an upright cylinder

```
Cylinder {
    height 2.0
    radius 1.0
}
[ cyl.wrl ]
```
Building primitive shapes

**Syntax: Sphere**

- A *Sphere* geometry node builds a sphere

```
Sphere {
    radius 1.0
}
```

[ sphere.wrl ]

Building primitive shapes

**Syntax: Text**

- A *Text* geometry node builds text

```
Text {
    string [ "Text", "Shape" ]
    fontStyle FontStyle {
        style "BOLD"
    }
}
```

[ text.wrl ]
Building primitive shapes

A sample primitive shape

```vrml
#VRML V2.0 utf8
# A cylinder
Shape {
  appearance Appearance {
    material Material {
    }
  }
  geometry Cylinder {
    height 2.0
    radius 1.5
  }
}
```

[ cylinder.wrl ]
Building primitive shapes

Building multiple shapes

- Shapes are built centered in the world
- A VRML file can contain multiple shapes
- Shapes overlap when built at the same location

A sample file with multiple shapes

```cmavo
#VRML V2.0 utf8
Shape { . . . }
Shape { . . . }
Shape { . . . }
```
A sample file with multiple shapes

Syntax: FontStyle

- A FontStyle node describes a font
  - family - SERIF, SANS, OR TYPEWRITER
  - style - BOLD, ITALIC, BOLDITALIC, OR PLAIN,
  - more ...

```xml
Text {
    string . . .
    fontStyle FontStyle {
        family   "SERIF"
        style    "BOLD"
    }
}
```
Building primitive shapes

**Syntax: FontStyle**

- A FontStyle node describes a font
  - \textit{size} - character height
  - \textit{spacing} - row/column spacing
  - more . . .

```
Text {
  string . . .
  fontStyle FontStyle {
    size 1.0
    spacing 1.0
  }
}
```

Summary

- Shapes are built using a Shape node
- Shape geometry is built using geometry nodes, such as Box, Cone, Cylinder, Sphere, and Text
- Text fonts are controlled using a FontStyle node
Motivation

Example

Using coordinate systems

Visualizing a coordinate system

Transforming a coordinate system

Syntax: Transform

Including children

Translating

Translating

Rotating

Specifying rotation axes

Using the Right-Hand Rule

Using the Right-Hand Rule

Rotating

Scaling

Scaling

Scaling, rotating, and translating

Scaling, rotating, and translating

A sample transform group

A sample transform group

Summary

Transforming shapes

Motivation

- By default, all shapes are built at the center of the world

- A *transform* enables you to
  - Position shapes
  - Rotate shapes
  - Scale shapes
Transforming shapes

Example

[ towers.wrl ]

Using coordinate systems

- A VRML file builds components for a world
- A file’s world components are built in the file’s world coordinate system
- By default, all shapes are built at the origin of the world coordinate system
A transform creates a coordinate system that is
  • Positioned
  • Rotated
  • Scaled
relative to a parent coordinate system

Shapes built in the new coordinate system are positioned, rotated, and scaled along with it
Transforming shapes

Syntax: Transform

- The Transform group node creates a group with its own coordinate system
  - children - shapes to build
  - translation - position
  - rotation - orientation
  - scale - size

Transform { translation . . . rotation . . . scale . . . children  [ . . . ] }

Including children

- The children field includes a list of one or more nodes

Transform { . . .
  children [ 
    Shape { . . . }
    Transform { . . . }
  ]
  . . . }
}
Transforming shapes

**Translating**

- *Translation* positions a coordinate system in X, Y, and Z

```
Transform {
  #       X   Y   Z
  translation 2.0 0.0 0.0
  children [ . . . ]
}
```
Transforming shapes

**Rotating**

- Rotation orients a coordinate system about a rotation axis by a rotation angle
- Angles are measured in radians

```
Transform {
  #        X   Y   Z    Angle
  rotation 0.0 0.0 1.0  0.52
  children [ . . . ]
}
```

**Specifying rotation axes**

- A rotation axis defines a pole to rotate around
  - Like the Earth’s North-South pole

- Typical rotations are about the X, Y, or Z axes:

<table>
<thead>
<tr>
<th>Rotate about</th>
<th>Axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Axis</td>
<td>1.0 0.0 0.0</td>
</tr>
<tr>
<td>Y-Axis</td>
<td>0.0 1.0 0.0</td>
</tr>
<tr>
<td>Z-Axis</td>
<td>0.0 0.0 1.0</td>
</tr>
</tbody>
</table>
Transforming shapes

*Using the Right-Hand Rule*

- To help remember positive and negative rotation directions:
  - Open your hand
  - Stick out your thumb
  - Aim your thumb in an axis positive direction
  - Curl your fingers around the axis

- The curl direction is a positive rotation
Transforming shapes

**Rotating**

- *Scale* grows or shrinks a coordinate system by a scaling factor in X, Y, and Z

```plaintext
Transform {
    # X Y Z
    scale 0.5 0.5 0.5
    children [ . . . ]
}
```
Scaling

Scaling, rotating, and translating

- Scale, Rotate, and Translate a coordinate system, one after the other

```
Transform {
  translation 2.0 0.0 0.0
  rotation 0.0 0.0 1.0 0.52
  scale 0.5 0.5 0.5
  children [ . . . ]
}
```
Transforming shapes

Scaling, rotating, and translating

A sample transform group

```
Transform {
    translation 4.0 0.0 0.0
    rotation 0.0 1.0 0.0 0.785
    scale 0.5 0.5 0.5
    children [ . . . ]
}
```
A sample transform group

Summary

- All shapes are built in a coordinate system.
- The Transform node creates a new coordinate system relative to its parent.
- Transform node fields do:
  - translation
  - rotation
  - scale
Motivation

Example

Syntax: Shape

Syntax: Appearance

Syntax: Material

Specifying colors

Syntax: Material

Experimenting with shiny materials

Example

A sample world using appearance

A sample world using appearance

Summary

Motivation

- The primitive shapes have a default emissive (glowing) white appearance

- You can control a shape’s
  - Shading color
  - Glow color
  - Transparency
  - Shininess
  - Ambient intensity
Controlling appearance with materials

Example

[ colors.wrl ]

Syntax: Shape

- Recall that \texttt{Shape} nodes describe:
  - \textit{appearance} - color and texture
  - \textit{geometry} - form, or structure

\begin{verbatim}
Shape {
    appearance . . .
    geometry . . .
}
\end{verbatim}
Controlling appearance with materials

**Syntax: Appearance**

- An Appearance node describes overall shape appearance
  - *material* properties - color, transparency, etc.
  - more . . .

```
Shape {
  appearance Appearance {
    material . . .
  }
  geometry . . .
}
```

**Syntax: Material**

- A Material node controls shape material attributes
  - *diffuse color* - main shading color
  - *emissive color* - glowing color
  - *transparency* - opaque or not
  - more . . .

```
Material {
  diffuseColor . . .
  emissiveColor . . .
  transparency . . .
}
```
Specifying colors

- Colors specify:
  - A mixture of red, green, and blue light
  - Values between 0.0 (none) and 1.0 (lots)

<table>
<thead>
<tr>
<th>Color</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>(white)</td>
</tr>
<tr>
<td>Red</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>(red)</td>
</tr>
<tr>
<td>Yellow</td>
<td>1.0</td>
<td>1.0</td>
<td>0.0</td>
<td>(yellow)</td>
</tr>
<tr>
<td>Magenta</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td>(magenta)</td>
</tr>
<tr>
<td>Brown</td>
<td>0.5</td>
<td>0.2</td>
<td>0.0</td>
<td>(brown)</td>
</tr>
</tbody>
</table>

A Material node also controls shape shininess
- specular color - highlight color
- shininess - highlight size
- ambient intensity - ambient lighting effects

Material {
    . . .
    specularColor 0.71 0.70 0.56
    shininess 0.16
    ambientIntensity 0.4
}

Controlling appearance with materials
Controlling appearance with materials

**Experimenting with shiny materials**

<table>
<thead>
<tr>
<th>Description</th>
<th>ambient Intensity</th>
<th>diffuseColor</th>
<th>specularColor</th>
<th>shininess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aluminum</td>
<td>0.3</td>
<td>0.30 0.30 0.50</td>
<td>0.70 0.70 0.80</td>
<td>0.10</td>
</tr>
<tr>
<td>Copper</td>
<td>0.26</td>
<td>0.30 0.11 0.00</td>
<td>0.75 0.33 0.00</td>
<td>0.08</td>
</tr>
<tr>
<td>Gold</td>
<td>0.4</td>
<td>0.22 0.15 0.00</td>
<td>0.71 0.70 0.56</td>
<td>0.16</td>
</tr>
<tr>
<td>Metallic Purple</td>
<td>0.17</td>
<td>0.10 0.03 0.22</td>
<td>0.64 0.00 0.98</td>
<td>0.20</td>
</tr>
<tr>
<td>Metallic Red</td>
<td>0.15</td>
<td>0.27 0.00 0.00</td>
<td>0.61 0.13 0.18</td>
<td>0.20</td>
</tr>
<tr>
<td>Plastic Blue</td>
<td>0.10</td>
<td>0.20 0.20 0.71</td>
<td>0.83 0.83 0.83</td>
<td>0.12</td>
</tr>
</tbody>
</table>

**Example**

[ shiny.wrl ]
Controlling appearance with materials

A sample world using appearance

Shape {
    appearance Appearance {
        material Material {
            diffuseColor 1.0 1.0 1.0
        }
    }
    geometry . . .
}
Controlling appearance with materials

Summary

- The **Appearance** node controls overall shape appearance

- The **Material** node controls overall material properties including:
  - Shading color
  - Glow color
  - Transparency
  - Shininess
  - Ambient intensity
Motivation
Syntax: Group
Syntax: Switch
Syntax: Transform
Syntax: Billboard
Billboard rotation axes
A sample billboard group
A sample billboard group
Syntax: Anchor
A Sample Anchor
Syntax: Inline
A sample inlined file
A sample inlined file
Summary
Summary

• You can group shapes to compose complex shapes
• VRML has several grouping nodes, including:

  Group       { . . . }
  Switch      { . . . }
  Transform   { . . . }
  Billboard   { . . . }
  Anchor      { . . . }
  Inline      { . . . }
Grouping nodes

**Syntax: Group**

- The `Group` node creates a basic group
- *Every child* node in the group is displayed

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

**Syntax: Switch**

- The `Switch` group node creates a switched group
- *Only one child* node in the group is displayed
- You select which child

```
Switch {
  whichChoice 0
  choice [ . . . ]
}
```
Grouping nodes

**Syntax: Transform**

- The *Transform* group node creates a group with its own coordinate system
  - *Every child* node in the group is displayed

  ```
  Transform {
    translation . . .
    rotation    . . .
    scale       . . .
    children [ . . . ]
  }
  ```

Grouping nodes

**Syntax: Billboard**

- The *Billboard* group node creates a group with a special coordinate system
  - *Every child* node in the group is displayed
  - Coordinate system is turned to face viewer

  ```
  Billboard {
    axisOfRotation . . .
    children [ . . . ]
  }
  ```
Grouping nodes

**Billboard rotation axes**

- A rotation axis defines a pole to rotate round
- Similar to a Transform node’s rotation field, but no angle (auto computed)

A sample billboard group

```plaintext
Group {
    children [
        Billboard {
            axisOfRotation 0.0 1.0 0.0
            children [ ... ]
        }
        Transform { . . . }
    ]
}
```
Grouping nodes

A sample billboard group

Syntax: Anchor

- An Anchor node creates a group that acts as a clickable anchor
- Every child node in the group is displayed
- Clicking any child follows a URL
- A description names the anchor

Anchor {
  url "stairwy.wrl"
  description "Twisty Stairs"
  children [...]
}
Grouping nodes

**A Sample Anchor**

```
[ anchor.wrl ]   [ stairwy.wrl ]
```

**Syntax: Inline**

- An Inline node creates a special group from another VRML file’s contents
- Children read from file selected by a URL
- *Every child* node in group is displayed

```
Inline {
  url "table.wrl"
}
```
A sample inlined file

Inline { url "table.wrl" }

Transform {
    translation ...
    children [
        Inline { url "chair.wrl" }
    ]
}
Summary

- The **Group** node creates a basic group

- The **Switch** node creates a group with 1 choice used

- The **Transform** node creates a group with a new coordinate system

---

Summary

- The **Billboard** node creates a group with a coordinate system that rotates to face the viewer

- The **Anchor** node creates a clickable group
  - Clicking any child in the group loads a URL

- The **Inline** node creates a special group loaded from another VRML file
Motivation

- If several shapes have the same geometry or appearance, you must use multiple duplicate nodes, one for each use
- Instead, define a name for the first occurrence of a node
- Later, use that name to share the same node in a new context
Naming nodes

**Syntax: DEF**

- The **DEF** syntax gives a name to a node
  
  ```
  DEF RedColor Material {
      diffuseColor 1.0 0.0 0.0
  }
  ```

- You can name any node
- Names can be most any sequence of letters and numbers
- Names must be unique within a file

**Syntax: USE**

- The **USE** syntax uses a previously named node
  
  ```
  Appearance {
      material USE RedColor
  }
  ```

- A re-use of a named node is called an *instance*
- A named node can have any number of instances
- Each instance shares the same node description
Naming nodes

Using named nodes

- Naming and using nodes:
  - Saves typing
  - Reduces file size
  - Enables rapid changes to shapes with the same attributes
  - Speeds browser processing

- Names are also necessary for animation...

A sample use of node names

[ dinette.wrl ]
Naming nodes

\textit{Summary}

- \texttt{DEF} names a node

- \texttt{USE} uses a named node
Summary examples

A fairy-tale castle
A bar plot
A simple spaceship
A juggling hand

A fairy-tale castle

- Cylinder nodes build the towers
- Cone nodes build the roofs and tower bottoms

[ castle.wrl ]
Summary examples

A bar plot

- **Box** nodes create the bars
- **Text** nodes provide bar labels
- **Billboard** nodes keep the labels facing the viewer

[ barplot.wrl ]

Summary examples

A simple spaceship

- **Sphere** nodes make up all parts of the ship
- **Transform** nodes scale the spheres into ship parts

[ space2.wrl ]
Summary examples

*A juggling hand*

- Cylinder and Sphere nodes build fingers and joints
- Transform nodes articulate the hand

[ hand.wrl ]
Motivation

Building animation circuits

Examples

Routing events

Using node inputs and outputs

Sample inputs

Sample outputs

Syntax: ROUTE

Event data types

Following naming conventions

A sample animation

A sample animation

Using multiple routes

Summary

---

**Motivation**

- Nodes like Billboard and Anchor have built-in behavior

- You can create your own behaviors to make shapes move, rotate, scale, blink, and more

- We need a means to trigger, time, and respond to a sequence of events in order to provide better user/world interactions
Introducing animation

Building animation circuits

- Almost every node can be a component in an animation circuit
  - Nodes act like virtual electronic parts
  - Nodes can send and receive events
  - Wired routes connect nodes together

- An event is a message sent between nodes
  - A data value (such as a translation)
  - A time stamp (when did the event get sent)

Examples

- To spin a shape:
  - Connect a node that sends rotation events to a Transform node’s rotation field

- To blink a shape:
  - Connect a node that sends color events to a Material node’s diffuseColor field
To set up an animation circuit, you need:

- A node which sends events
  - The node must be named with DEF

- A node which receives events
  - The node must be named with DEF

- A route connecting them

Every node has fields, inputs, and outputs:

- field: A stored value
- eventIn: An input
- eventOut: An output

An exposedField is a short-hand for a field, eventIn, and eventOut
**Introducing animation**

**Sample inputs**

- Some **Transform** node inputs:
  - `set_translation`
  - `set_rotation`
  - `set_scale`

- Some **Material** node inputs:
  - `set_diffuseColor`
  - `set_emissiveColor`
  - `set_transparency`

**Sample outputs**

- Some **TouchSensor** node outputs:
  - `isOver`
  - `isActive`
  - `touchTime`

- An **OrientationInterpolator** node output:
  - `value_changed`

- A **PositionInterpolator** node output:
  - `value_changed`
Introducing animation

Syntax: ROUTE

- A ROUTE statement connects two nodes together using
  - The sender’s node name and eventOut name
  - The receiver’s node name and eventIn name

ROUTE MySender.rotation_changed TO MyReceiver.set_rotation

- Event data types must match!

<table>
<thead>
<tr>
<th>Event data types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SFBool</td>
<td>SFRotation / MFRotation</td>
</tr>
<tr>
<td>SFColor / MFColor</td>
<td>SFString / MFString</td>
</tr>
<tr>
<td>SFFloat / MFFloat</td>
<td>SFTime</td>
</tr>
<tr>
<td>SFImage</td>
<td>SFVec2f / MFVec2f</td>
</tr>
<tr>
<td>SFInt32 / MFIInt32</td>
<td>SFVec3f / MFVec3f</td>
</tr>
<tr>
<td>SFNode / MFNode</td>
<td></td>
</tr>
</tbody>
</table>
Introducing animation

Following naming conventions

- Most nodes have *exposedFields*

- If the exposed field name is `xxx`, then:
  - `set_xxx` is an *eventIn* to set the field
  - `xxx_changed` is an *eventOut* that sends when the field changes
- The `set_` and `_changed` suffixes are optional but recommended for clarity

- The *Transform* node has:
  - `rotation` field
  - `set_rotation` eventIn
  - `rotation_changed` eventOut

---

A sample animation

```plaintext
DEF RotateMe Transform {
  rotation 0.0 1.0 0.0 0.0
  children [ . . . ]
}
DEF Rotator OrientationInterpolator { . . . }

ROUTE Rotator.value_changed TO RotateMe.set_rotation
```
Introducing animation

A sample animation

[ colors.wrl ]

Introducing animation

Using multiple routes

- You can have *fan-out*
  - Multiple routes out of the same sender

- You can have *fan-in*
  - Multiple routes into the same receiver
Introducing animation

Summary

- Connect senders to receivers using routes

- *eventIns* are inputs, and *eventOuts* are outputs

- A route names the *sender.eventOut*, and the *receiver.eventIn*
  - Data types must match

- You can have multiple routes into or out of a node
Animating transforms

Motivation
Example
Controlling time
Using absolute time
Using fractional time
Syntax: TimeSensor
Using timers
Using timers
Using cycling timers
Using timer outputs
A sample time sensor
A sample time sensor
Converting time to position
Interpolating positions
Syntax: PositionInterpolator
Using position interpolator inputs and outputs
A sample using position interpolators
A sample using position interpolators
Using other types of interpolators
Syntax: OrientationInterpolator
Syntax: ColorInterpolator
Syntax: ScalarInterpolator

Summary
Summary
Summary
Motivation

- An animation changes something over time:
  - position - a car driving
  - orientation - an airplane banking
  - color - seasons changing

- Animation requires control over time:
  - When to start and stop
  - How fast to go
Animating transforms

**Controlling time**

- A `TimeSensor` node is similar to a stop watch
  - You control the start and stop time

- The sensor generates time events while it is running

- To animate, route time events into other nodes

---

**Using absolute time**

- A `TimeSensor` node generates *absolute* and *fractional* time events

- Absolute time events give the wall-clock time
  - Absolute time is measured in seconds since 12:00am January 1, 1970!
  - Useful for triggering events at specific dates and times
Animating transforms

*Using fractional time*

- Fractional time events give a number from 0.0 to 1.0
  - Values *cycle* from 0.0 to 1.0, then repeat
  - The number of seconds between 0.0 and 1.0 is controlled by the *cycle interval*
- The sensor can loop forever, or run once and stop

*Syntax: TimeSensor*

- A *TimeSensor* node generates events based upon time
  - *start* and *stop* time - when to run
  - *cycle interval* time - how long a cycle is
  - *looping* - whether or not to repeat cycles

```plaintext
TimeSensor {
    cycleInterval 1.0
    loop FALSE
    startTime 0.0
    stopTime 0.0
}
```
Creating continuously running timers:
- `loop TRUE`  
- `stopTime <= startTime`

Run one cycle then stop
- `loop FALSE`  
- `stopTime <= startTime`

Run until stopped, or after cycle is over
- `loop TRUE or FALSE`  
- `stopTime > startTime`

The `set_startTime` input event:
- Sets when the timer should start

The `set_stopTime` input event:
- Sets when the timer should stop
Animating transforms

Using cycling timers

- The first cycle starts at the *start time*

- The *cycle interval* is the length (in seconds) of the cycle

- Each cycle varies a fraction from 0.0 to 1.0

- If `loop` is FALSE, there is only one cycle, otherwise the timer may cycle forever

Using timer outputs

- The *isActive* output event:
  - Outputs `true` at timer start
  - Outputs `false` at timer stop

- The *time* output event:
  - Outputs the absolute time

- The *fraction_changed* output event:
  - Outputs values from 0.0 to 1.0 during a cycle
  - Resets to 0.0 at the start of each cycle
Animating transforms

A sample time sensor

DEF Monolith1Timer TimeSensor {
    cycleInterval 4.0
    loop FALSE
    startTime 0.0
    stopTime 1.0
}
ROUTE Monolith1Touch.touchTime TO Monolith1Timer.set_startTime
ROUTE Monolith1Timer.fraction_changed TO Monolith1Light.set_intensity

[ monolith.wrl ]
To animate the position of a shape you provide:
- A list of key positions for a movement path
- A time at which to be at each position

An interpolator node converts an input time to an output position
- When a time is in between two key positions, the interpolator computes an intermediate position

<table>
<thead>
<tr>
<th>Time</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0 0.0 0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>0.4 0.1 0.0</td>
</tr>
<tr>
<td>0.2</td>
<td>0.8 0.2 0.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0.5</td>
<td>4.0 1.0 0.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Animating transforms

Syntax: PositionInterpolator

- A PositionInterpolator node describes a position path
  - keys - key fractional times
  - key values - key positions

PositionInterpolator {
  key [ 0.0, . . . ]
  keyValue [ 0.0 0.0 0.0, . . . ]
}

- Route into a Transform node’s set_translation input

Using position interpolator inputs and outputs

- The set_fraction input:
  - Sets the current fractional time along the key path

- The value_changed output:
  - Outputs the position along the path each time the fraction is set
Animating transforms

*A sample using position interpolators*

```literate
DEF Mover PositionInterpolator {
  key [ 0.0, . . . ]
  keyValue [ 0.0 0.0 0.0, . . . ]
}
ROUTE Clock.fraction_changed
    TO Mover.set_fraction

ROUTE Mover.value_changed
    TO Movee.set_translation
```

[ floater.wrl ]
Animating transforms

*Using other types of interpolators*

- To animate shape orientation, use an OrientationInterpolator

- To animate shape color, use a ColorInterpolator

- To animate shape transparency, use a ScalarInterpolator

- To animate shape scale, use a trick and use a PositionInterpolator

**Syntax: OrientationInterpolator**

- A OrientationInterpolator node describes an orientation path
  - *keys* - key fractions
  - *key values* - key rotations (axis and angle)

  OrientationInterpolator {
    key [ 0.0, . . . ]
    keyValue [ 0.0 1.0 0.0 0.0, . . . ]
  }

- Route into a Transform node’s set_rotation input
Syntax: **ColorInterpolator**

- **ColorInterpolator** node describes a color path
  - `keys` - key fractions
  - `values` - key colors (red, green, blue)

```plaintext
ColorInterpolator {
  key [ 0.0, . . . ]
  keyValue [ 1.0 1.0 0.0, . . . ]
}
```

- Route into a **Material** node’s
  `set_diffuseColor` or `set_emissiveColor` inputs

Syntax: **ScalarInterpolator**

- **ScalarInterpolator** node describes a scalar path
  - `keys` - key fractions
  - `values` - key scalars (used for anything)

```plaintext
ScalarInterpolator {
  key [ 0.0, . . . ]
  keyValue [ 4.5, . . . ]
}
```

- Route into a **Material** node’s
  `set_transparency` input
Animating transforms

Syntax: PositionInterpolator

- A PositionInterpolator node describes a position or scale path
  - keys - key fractional times
  - key values - key positions (or scales)

```wrl
PositionInterpolator {
    key [ 0.0, . . . ]
    keyValue [ 0.0 0.0 0.0, . . . ]
}
```

- Route into a Transform node's set_scale input

A sample using other interpolators

[squisher.wrl]
• The `TimeSensor` node’s fields control
  • Timer start and stop times
  • The cycle interval
  • Whether the timer loops or not

• The sensor outputs
  • true/false on `isActive` at start and stop
  • absolute time on `time` while running
  • fractional time on `fraction_changed` while running

• Interpolators use key times and values and compute intermediate values

• All interpolators have:
  • a `set_fraction` input to set the fractional time
  • a `value_changed` output to send new values
Summary

- The PositionInterpolator node converts times to positions (or scales)

- The OrientationInterpolator node converts times to rotations

- The ColorInterpolator node converts times to colors

- The ScalarInterpolator node converts times to scalars (such as transparencies)
Sensing viewer actions

Motivation

Using action sensors
Sensing shapes
Syntax: TouchSensor
A sample use of a TouchSensor node
Syntax: SphereSensor
Syntax: CylinderSensor
Syntax: PlaneSensor
Using multiple sensors
A sample use of a multiple sensors
Summary

You can sense when the viewer’s cursor:
  • Is over a shape
  • Has touched a shape
  • Is dragging atop a shape

You can trigger animations on a viewer’s touch

You can enable the viewer to move and rotate shapes
There are four main action sensor types:
- **TouchSensor** senses touch
- **SphereSensor** senses drags
- **CylinderSensor** senses drags
- **PlaneSensor** senses drags

The **Anchor** node is a special-purpose action sensor with a built-in response

- All action sensors *sense* all shapes in the same group
- Sensors trigger when the viewer’s cursor *touches* a sensed shape
A TouchSensor node senses the cursor’s touch

- `isOver` - send true/false when cursor over/not over
- `isActive` - send true/false when mouse button pressed/released
- `touchTime` - send time when mouse button released

```
Transform {
  children [
    DEF Touched TouchSensor { }
  ]
}
```
Sensing viewer actions

**Syntax: SphereSensor**

- A SphereSensor node senses a cursor *drag* and generates rotations as if rotating a ball
  - `isActive` - sends true/false when mouse button pressed/released
  - `rotation_changed` - sends rotation during a drag

```
Transform {
    children [
        DEF RotateMe Transform { . . . }
        DEF Rotator  SphereSensor { }
    ]
}
ROUTE Rotator.rotation_changed TO RotateMe.set_rotation
```

**Syntax: CylinderSensor**

- A CylinderSensor node senses a cursor *drag* and generates rotations as if rotating a cylinder
  - `isActive` - sends true/false when mouse button pressed/released
  - `rotation_changed` - sends rotation during a drag

```
Transform {
    children [
        DEF RotateMe Transform { . . . }
        DEF Rotator  CylinderSensor { }
    ]
}
ROUTE Rotator.rotation_changed TO RotateMe.set_rotation
```
Sensing viewer actions

Syntax: PlaneSensor

- A PlaneSensor node senses a cursor *drag* and generates translations as if sliding on a plane
- *isActive* - sends true/false when mouse button pressed/released
- *translation_changed* - sends translations during a drag

```
Transform {
    children [
        DEF MoveMe Transform { . . . }
        DEF Mover PlaneSensor { }
    ]
}
ROUTE Mover.translation_changed TO MoveMe.set_translation
```

Sensing viewer actions

Using multiple sensors

- Multiple sensors can sense the same shape *but* . . .
- If sensors are in the same group:
  - They all respond
- If sensors are at different depths in the hierarchy:
  - The deepest sensor responds
  - The other sensors do not respond
Sensing viewer actions

A sample use of a multiple sensors

[ lamp.wrl ]

Summary

- Action sensors sense when the viewer’s cursor:
  - is over a shape
  - has touched a shape
  - is dragging atop a shape

- Sensors convert viewer actions into events to
  - Start and stop animations
  - Orient shapes
  - Position shapes
Motivation

Example

Building shapes using coordinates
Syntax: Coordinate
Using geometry coordinates
Syntax: PointSet
A sample PointSet node shape
Syntax: IndexedLineSet
Using line set coordinate indexes
Using line set coordinate index lists
A sample IndexedLineSet node shape
Syntax: IndexedFaceSet
Using face set coordinate index lists
A sample IndexedFaceSet node shape
Syntax: CoordinateInterpolator

Summary

Summary

Summary

Motivation

- Complex shapes are hard to build with primitive shapes
  - Terrain
  - Animals
  - Plants
  - Machinery

- Instead, build shapes out of atomic components:
  - Points, lines, and faces
Building shapes out of points, lines, and faces

**Example**

Building shapes using coordinates

- Shape building is like a 3-D *connect-the-dots* game:
  - Place *dots* at 3-D locations
  - Connect-the-dots to form shapes

- A *coordinate* specifies a 3-D *dot* location
  - Measured relative to a coordinate system origin

- A geometry node specifies how to connect the dots
Building shapes out of points, lines, and faces

**Syntax: Coordinate**

- A Coordinate node contains a list of coordinates for use in building a shape

```
Coordinate {
    point [
        #    X    Y    Z
        2.0 1.0 3.0,
        4.0 2.5 5.3,
        ... ...
    ]
}
```

**Using geometry coordinates**

- Build shapes using geometry nodes:
  - PointSet
  - IndexedLineSet
  - IndexedFaceSet

- For all three nodes, use a Coordinate node as the value of the coord field
Building shapes out of points, lines, and faces

**Syntax:** *PointSet*

- A *PointSet* geometry node creates geometry out of *points*
  - One point (a dot) is placed at each coordinate

    ```
    PointSet {
        coord Coordinate {
            point [ . . . ]
        }
    }
    ```

A sample *PointSet* node shape

[ ptplot.wrl ]
Building shapes out of points, lines, and faces

**Syntax: IndexedLineSet**

- An IndexedLineSet geometry node creates geometry out of *lines*
  - A straight line is drawn between pairs of selected coordinates

```
IndexedLineSet {
  coord Coordinate {
    point [ . . . ]
  }
  coordIndex [ . . . ]
}
```

Using line set coordinate indexes

- Each coordinate in a Coordinate node is implicitly numbered
  - Index 0 is the first coordinate
  - Index 1 is the second coordinate, etc.

- To build a line shape
  - Make a list of coordinates, using their indexes
  - Use an IndexedLineSet node to draw a line from coordinate to coordinate in the list
Using line set coordinate index lists

- 1, 0, 3, -1, ...
  - 1, 0, Draw from 1 to 0
  - 0, 3, Draw from 0 to 3
  - -1, End line sequence

List coordinate indexes in the coordIndex field of the IndexedLineSet node
Building shapes out of points, lines, and faces

**Syntax: IndexedFaceSet**

- An IndexedFaceSet geometry node creates geometry out of *faces*
  - A flat *facet* (polygon) is drawn using an outline specified by coordinates

IndexedFaceSet {
  coord Coordinate {
    point [ . . . ]
  }
  coordIndex [ . . . ]
}

Building shapes out of points, lines, and faces

**Using face set coordinate index lists**

- To build a face shape
  - Make a list of coordinates, using their indexes
  - Use an IndexedFaceSet node to draw a face outlined by the coordinates in the list

- List coordinate indexes in the coordIndex field of the IndexedFaceSet node
A sample IndexedFaceSet node shape

Syntax: CoordinateInterpolator

- A `CoordinateInterpolator` node describes a coordinate path
  - `keys` - key fractions
  - `values` - key coordinate lists (X,Y,Z lists)

```
CoordinateInterpolator {
  key [ 0.0, . . . ]
  keyValue [ 0.0 1.0 0.0, . . . ]
}
```
Summary

- Shapes are built by connecting together coordinates
- Coordinates are listed in a Coordinate node
- Coordinates are implicitly numbers starting at 0
- Coordinate index lists give the order in which to use coordinates

The PointSet node draws a dot at every coordinate
- The coord field value is a Coordinate node

The IndexedLineSet node draws lines between coordinates
- The coord field value is a Coordinate node
- The coordIndex field value is a list of coordinate indexes
Building shapes out of points, lines, and faces

Summary

- The IndexedFaceSet node draws faces outlined by coordinates
  - The coord field value is a Coordinate node
  - The coordIndex field value is a list of coordinate indexes

- The CoordinateInterpolator node converts times to coordinates
Building elevation grids

Motivation

- Building terrains is very common
  - Hills, valleys, mountains
  - Other tricky uses...

- You can build a terrain using an IndexedFaceSet node

- You can build terrains more efficiently using an ElevationGrid node
Building elevation grids

Example

Syntax: `ElevationGrid`

- An `ElevationGrid` geometry node creates terrains
  - `X & Z dimensions` - grid size
  - `X & Z spacings` - row and column distances
  - `more . . .`

```
ElevationGrid {
  xDimension 3
  zDimension 2
  xSpacing 1.0
  zSpacing 1.0
  . . .
}
```
Building elevation grids

Syntax: ElevationGrid

- An ElevationGrid geometry node creates terrains
  - `height` - elevations at grid points

ElevationGrid {
    . . .
    height [
        0.0, -0.5, 0.0,
        0.2, 4.0, 0.0
    ]
}

A sample elevation grid

Shape {
    . . .
    geometry ElevationGrid {
        xDimension 9
        zDimension 9
        xSpacing 1.0
        zSpacing 1.0
        height [. . .]
    }
}
Building elevation grids

A sample elevation grid

Summary

- An `ElevationGrid` node efficiently creates a terrain
- Grid size is specified in the `xDimension` and `zDimension` fields
- Grid spacing is specified in the `xSpacing` and `zSpacing` field
- Elevations at each grid point are specified in the `height` field
Building extruded shapes

Motivation

Examples
Creating extruded shapes
Extruding along a straight line
Extruding around a circle
Extruding along a helix
Syntax: Extrusion
Squishing and twisting extruded shapes
Syntax: Extrusion
Sample extrusions with scale and rotation
Summary

Extruded shapes are very common
- Tubes, pipes, bars, vases, donuts
- Other tricky uses...

- You can build extruded shapes using an IndexedFaceSet node
- You can build extruded shapes more easily and efficiently using an Extrusion node
Building extruded shapes

Examples

[ slide.wrl ]

[ donut.wrl ]

Creating extruded shapes

- Extruded shapes are described by
  - A 2-D cross-section
  - A 3-D spine along which to sweep the cross-section

- Extruded shapes are like long bubbles created with a bubble wand
  - The bubble wand’s outline is the cross-section
  - The path along which you swing the wand is the spine
Building extruded shapes

Extruding along a straight line

Extruding around a circle
Extruding along a helix

Syntax: Extrusion

• An Extrusion geometry node creates extruded geometry
  • 2-D cross-section - cross-section
  • 3-D spine - sweep path
  • more . . .

Extrusion {
  crossSection [ . . . ]
  spine [ . . . ]
  . . .
}

Building extruded shapes
Building extruded shapes

_Squishing and twisting extruded shapes_

- You can scale the cross-section along the spine
  - Vases, musical instruments
  - Surfaces of revolution

- You can rotate the cross-section along the spine
  - Twisting ribbons

_Syntax: Extrusion_

- An Extrusion geometry node creates geometry using
  - _scales_ - cross-section scaling per spine point
  - _rotations_ - cross-section rotation per spine point

Extrusion {
  . . .
  scale [ . . . ]
  orientation [ . . . ]
}

Sample extrusions with scale and rotation

[ horn.wrl ]  [ bartwist.wrl ]

Summary

- An Extrusion node efficiently creates extruded shapes
- The crossSection field specifies the cross-section
- The spine field specifies the sweep path
- The scale and orientation fields specify scaling and rotation at each spine point
Motivation

Example

Syntax: Color

Binding colors

Syntax: PointSet
A sample PointSet node shape

Syntax: IndexedLineSet
Controlling color binding for line sets
A sample IndexedLineSet node shape

Syntax: IndexedFaceSet
Controlling color binding for face sets
A sample IndexedFaceSet node shape

Syntax: ElevationGrid
Controlling color binding for elevation grids
A sample ElevationGrid node shape

Controlling shading using the crease angle
Selecting crease angles
A sample using crease angles

Syntax: Normal
Syntax: IndexedFaceSet
Controlling normal binding for face sets
Syntax: ElevationGrid
Motivation

- The Material node gives an entire shape the same color

- You can provide colors for parts of a shape using a Color node

- You can specify smooth or faceted shading using a creaseAngle field value

Example

[ cmount.wrl ]
Controlling properties of coordinate-based geometry

**Syntax: Color**

- A Color node contains a list of RGB values
  ```
  Color {
    color [ 1.0 0.0 0.0, . . . ]
  }
  ```

- Used as the color field value of
  - IndexedFaceSet, IndexedLineSet, PointSet or ElevationGrid nodes

**Binding colors**

- Colors in the Color node override those in the Material node

- You can bind colors
  - To each point, line, or face
  - To each coordinate in a line, or face
A PointSet geometry node creates geometry out of points
  * color - provides a list of colors
  * Always binds one color to each point, in order

PointSet {
  coord Coordinate { . . . }
  color Color { . . . }
}

A sample PointSet node shape

[ scatter.wrl ]
- An IndexedLineSet geometry node creates geometry out of lines
  - `color` - a list of colors
  - `color indexes` - selects colors from list (just like selecting coordinates)
  - `color per vertex` - control color binding

IndexedLineSet {
  coord Coordinate { . . . }
  coordIndex [ . . . ]
  color Color { . . . }
  colorIndex [ . . . ]
  colorPerVertex TRUE
}

- The `colorPerVertex` field controls how color indexes are used
  - `FALSE`: one color index to each line (ending at -1 coordinate indexes)
  - `TRUE`: one color index to each coordinate index of each line (including -1 coordinate indexes)
Controlling properties of coordinate-based geometry

A sample IndexedLineSet node shape

Syntax: IndexedFaceSet

- An IndexedFaceSet geometry node creates geometry out of faces
  - \textit{color} - a list of colors
  - \textit{color indexes} - selects colors from list (just like selecting coordinates)
  - \textit{color per vertex} - control color binding

IndexedFaceSet {
  coord Coordinate { . . . }
  coordIndex [ . . . ]
  color Color { . . . }
  colorIndex [ . . . ]
  colorPerVertex TRUE
}
Controlling properties of coordinate-based geometry

Controlling color binding for face sets

- The `colorPerVertex` field controls how color indexes are used (similar to line sets)
  - `FALSE`: one color index to each face (ending at -1 coordinate indexes)
  - `TRUE`: one color index to each coordinate index of each face (including -1 coordinate indexes)

A sample IndexedFaceSet node shape

[ log.wrl ]
Controlling properties of coordinate-based geometry

Syntax: `ElevationGrid`

- An `ElevationGrid` geometry node creates terrains
  - `color` - a list of colors
  - `color per vertex` - control color binding

```plaintext
ElevationGrid {
  height [ . . . ]
  color Color { . . . }
  colorPerVertex TRUE
}
```

- The `ElevationGrid` node does not use color indexes

Controlling color binding for elevation grids

- The `colorPerVertex` field controls how color indexes are used (similar to line and face sets)
  - `FALSE`: one color to each grid square
  - `TRUE`: one color to each height for each grid square
Controlling properties of coordinate-based geometry

A sample ElevationGrid node shape

Controlling shading using the crease angle

- By default, faces are drawn with faceted shading

- You can do smooth shading using the `creaseAngle` field for
  - IndexedFaceSet
  - ElevationGrid
  - Extrusion

[ cmount.wrl ]
Controlling properties of coordinate-based geometry

Selecting crease angles

- A crease angle is a threshold angle between two faces
  - If face angle $\geq$ crease angle, use facet shading
  - If face angle $<$ crease angle, use smooth shading

A sample using crease angles

crease angle = 0
crease angle = 45 deg
Controlling properties of coordinate-based geometry

**Syntax: Normal**

- A Normal node contains a list of normal vectors that override use of a crease angle
  
  ```
  Normal {
    vector [ 0.0 1.0 0.0, . . . ]
  }
  ```

- Usually automatically generated normals are good enough

- Normals can be given for IndexedFaceSet and ElevationGrid nodes

**Syntax: IndexedFaceSet**

- An IndexedFaceSet geometry node creates geometry out of faces
  - **Normal vectors** - list of normals
  - **Normal indexes** - selects normals from list (just like selecting coordinates)
  - **Normal binding** - control normal binding

  ```
  IndexedFaceSet {
    coord Coordinate { . . . }
    coordIndex [ . . . ]
    normal Normal { . . . }
    normalIndex [ . . . ]
    normalPerVertex TRUE
  }
  ```
Controlling properties of coordinate-based geometry

Controlling normal binding for face sets

- The `normalPerVertex` field controls how normal indexes are used
  - **FALSE**: one normal index to each face (ending at -1 coordinate indexes)
  - **TRUE**: one normal index to each coordinate index of each face (including -1 coordinate indexes)

Syntax: `ElevationGrid`

- An `ElevationGrid` geometry node creates terrains
  - **Normal vectors** - list of normals
  - **Normal indexes** - selects normals from list (just like selecting coordinates)
  - **Normal binding** - control normal binding

```
ElevationGrid {
    height [ . . . ]
    normal Normal { . . . }
    normalPerVertex TRUE
}
```
Controlling properties of coordinate-based geometry

Controlling normal binding for elevation grids

- The `normalPerVertex` field controls how normal indexes are used (similar to face sets)
  - **false**: one normal to each grid square
  - **true**: one normal to each height for each grid square

Syntax: `NormalInterpolator`

- A `NormalInterpolator` node describes a normal path
  - `keys` - key fractions
  - `values` - key normal lists (X,Y,Z lists)

```
NormalInterpolator {
  key [ 0.0, . . . ]
  keyValue [ 0.0 1.0 1.0, . . . ]
}
```
Summary

- The `Color` node lists colors to use for parts of a shape
  - Used as the value of the `color` field
  - Color indexes select colors to use
  - Colors override `Material` node

- The `colorPerVertex` field selects color per line/face/grid square or color per coordinate

Summary

- The `creaseAngle` field controls faceted or smooth shading

- The `Normal` node lists normal vectors to use for parts of a shape
  - Used as the value of the `normal` field
  - Normal indexes select normals to use
  - Normals override `creaseAngle` value

- The `normalPerVertex` field selects normal per face/grid square or normal per coordinate

- The `NormalInterpolator` node converts times to normals
Summary examples

A computed terrain
A twisty ribbon
A real-time clock
A timed timer
A morphing snake

236
Summary examples

A computed terrain

- An ElevationGrid node creates a computed terrain
- A Color node provides terrain colors

[ terrain1.wrl ]
Summary examples

**A twisty ribbon**

- An **Extrusion** node creates a ribbon
- **orientation** and **scale** fields make the ribbon twist and change size

[ ribbon2.wrl ]

**A real-time clock**

- A set of **TimeSensor** nodes watch the time
- A set of **OrientationInterpolator** nodes spin the clock hands

[ stopwtch.wrl ]
Summary examples

A timed timer

- A first TimeSensor node clocks a second TimeSensor node to create a periodic animation

[ timetime.wrl ]

Summary examples

A morphing snake

- A CoordinateInterpolator node animates the spine of an Extrusion node

[ snake.wrl ]
Motivation

You can model every tiny texture detail of a world using a vast number of colored faces.

- Takes a long time to write the VRML
- Takes a long time to draw

Use a trick instead

- Take a picture of the real thing
- Paste that picture on the shape, like sticking on a decal

This technique is called *Texture Mapping*
Mapping textures

Example

[can.wrl]

Example Textures
Mapping textures

**Using texture types**

- **Image textures**
  - A single image from a file
  - JPEG, GIF, or PNG format

- **Pixel textures**
  - A single image, given in the VRML file itself

- **Movie textures**
  - A movie from a file
  - MPEG format

**Syntax: Appearance**

- An Appearance node describes overall shape appearance
  - *texture* - texture source

```plaintext
Appearance {
    material Material { . . . }
    texture ImageTexture { . . . }
}
```
Mapping textures

*Using materials with textures*

- Color textures *override* the color in a Material node
- Grayscale textures *multiply* with the Material node color
  - Good for *colorizing* grayscale textures
Mapping textures

**Syntax: ImageTexture**

- An `ImageTexture` node selects a texture image for texture mapping
  - `url` - texture image file URL

```plaintext
ImageTexture {
  url "wood.jpg"
}
```

**Syntax: PixelTexture**

- A `PixelTexture` node specifies texture image pixels for texture mapping
  - `image` pixels - texture image pixels
  - `image data` - width, height, bytes/pixel, pixel values

```plaintext
PixelTexture {
  image 2 1 3 0xFFFF00 0xFF0000
}
```
A `MovieTexture` node selects a texture movie for texture mapping
  - `url` - texture movie file URL
  - When to play the movie, and how quickly (like a `TimeSensor` node)

```plaintext
MovieTexture {
  url "movie.mpg"
  loop TRUE
  speed 1.0
}
```

Texture images can include *color* and *transparency* values for each pixel.

- Pixel transparency enables you to make parts of a shape transparent
  - Windows, grillwork, holes
  - Trees, clouds
Mapping textures

A sample transparent texture

[ treewall.wrl ]
Mapping textures

Summary

- A texture is like a decal pasted to a shape
- Specify the texture using an ImageTexture, PixelTexture, or MovieTexture node in an Appearance node
- Color textures override material, grayscale textures multiply
- Textures with transparency create holes
Motivation

By default, an entire texture image is mapped once around the shape

You can also:
- Extract pieces of interest
- Create repeating patterns

Syntax: TextureCoordinate
Syntax: IndexedFaceSet
Syntax: ElevationGrid
Syntax: Appearance
Syntax: TextureTransform
A sample using no transform
A sample using translation
A sample using rotation
A sample using scale
A sample using texture coordinates
A sample using scale
A sample using scale and rotation

Summary
Controlling how textures are mapped

* Working through the texturing process

- Imagine the texture image is a big piece of rubbery cookie dough
- Select a texture image piece
  - Define the shape of a cookie cutter
  - Position and orient the cookie cutter
  - Stamp out a piece of texture dough
- Stretch the rubbery texture cookie to fit a face

* Using the texture coordinate system

- Texture images (the dough) are in a *texture coordinate system*
  - $S$ direction is horizontal
  - $T$ direction is vertical
  - $(0,0)$ at lower-left
  - $(1,1)$ at upper-right
Texture coordinates and transforms

- *Texture coordinates and texture coordinate indexes* specify a texture piece shape (the cookie cutter shape)

- *Texture transforms* translate, rotate, and scale the texture coordinates (placing the cookie cutter)

Working through the texturing process

- Select piece with texture coordinates and indexes
  - Create a cookie cutter

- Transform the texture coordinates
  - Position and orient the cookie cutter

- Bind the texture to a face
  - Stamp out the texture and stick it on a face

- The process is *very similar* to creating faces!
Controlling how textures are mapped

**Syntax:** `TextureCoordinate`

- A `TextureCoordinate` node contains a list of texture coordinates
  ```
  TextureCoordinate {
      point [ 0.2 0.2, 0.8 0.2, ... ]
  }
  ```

- Used as the `texCoord` field value of `IndexedFaceSet` or `ElevationGrid` nodes

Controlling how textures are mapped

**Syntax:** `IndexedFaceSet`

- An `IndexedFaceSet` geometry node creates geometry out of faces
  - `Texture coordinates` and `indexes` - specify texture pieces

  ```
  IndexedFaceSet {
      coord Coordinate { ... }
      coordIndex [ ... ]
      texCoord TextureCoordinate { ... }
      texCoordIndex [ ... ]
  }
  ```
Controlling how textures are mapped

**Syntax: ElevationGrid**

- An ElevationGrid geometry node creates terrains
- *Texture coordinates* - specify texture pieces
- Automatically generated texture coordinate indexes

ElevationGrid {
    height [ . . . ]
    texCoord TextureCoordinate { . . . }
}

Controlling how textures are mapped

**Syntax: Appearance**

- An Appearance node describes overall shape appearance
- *textureTransform* - the transform

Appearance {
    material Material { . . . }
    textureTransform TextureTransform {
    }
}
Controlling how textures are mapped

Syntax: `TextureTransform`

- A `TextureTransform` node transforms texture coordinates
  - `translation` - position
  - `rotation` - orientation
  - `scale` - size

```
TextureTransform {
  translation . . .
  rotation . . .
  scale . . .
}
```

A sample using no transform
Controlling how textures are mapped

*A sample using translation*

Controlling how textures are mapped

*A sample using rotation*
Controlling how textures are mapped

_A sample using scale_

Controlling how textures are mapped

_A sample using texture coordinates_

[pizza.wrl]
Controlling how textures are mapped

**A sample using scale**

[ brickb.wrl ]

Controlling how textures are mapped

**A sample using scale and rotation**

[ fence.wrl ]
Controlling how textures are mapped

**Summary**

- Texture images are in a texture coordinate system
- Texture coordinates and indexes describe a texture piece shape
- Texture transforms translate, rotate, and scale the texture coordinates
- Use one or both to fit texture to geometry and desired appearance
Motivation

Example

Using types of lights
Using common lighting features
Using common lighting features

Syntax: PointLight
Syntax: DirectionalLight
Syntax: SpotLight
Syntax: SpotLight

Example

Summary

Lighting your world

Motivation

- By default, you have one light in the scene, attached to your head

- For more realism, you can add multiple lights
  - Suns, light bulbs, candles
  - Flashlights, spotlights, firelight

- Lights can be positioned, oriented, and colored

- Lights do not cast shadows
There are three types of VRML lights:

- **Point lights** - radiate in all directions from a point
- **Directional lights** - aim in one direction from infinitely far away
- **Spot lights** - aim in one direction from a point, radiating in a cone

*Example*
All lights have several common fields:
- **on** - turn it on or off
- **intensity** - control brightness
- **ambientIntensity** - control ambient effect
- **color** - select color

Point lights and spot lights also have:
- **location** - position
- **radius** - maximum lighting distance
- **attenuation** - drop off with distance

Directional lights and spot lights also have
- **direction** - aim direction
Lighting your world

**Syntax: PointLight**

- A PointLight node illuminates radially from a point

```plaintext
PointLight {
  location 0.0 0.0 0.
  intensity 1.0
  color 1.0 1.0 1.0
}
```

**Syntax: DirectionalLight**

- A DirectionalLight node illuminates in one direction from infinitely far away

```plaintext
DirectionalLight {
  direction 1.0 0.0 0
  intensity 1.0
  color 1.0 1.0 1.0
}
```
Lighting your world

**Syntax:** SpotLight

- A **SpotLight** node illuminates from a point, in one direction, within a cone

```plaintext
SpotLight {
  location  0.0 0.0 0
  direction 1.0 0.0 0
  intensity 1.0
  color 1.0 1.0 1.0
}
```

- The maximum width of a spot light’s cone is controlled by the `cutOffAngle` field

- An inner cone region with constant brightness is controlled by the `beamWidth` field

```plaintext
SpotLight {
  . . .
  cutOffAngle 0.785
  beamWidth 1.571
}
```
There are three types of lights: point, directional, and spot.

- All lights have an on/off, intensity, ambient effect, and color.
- Point and spot lights have a location, radius, and attenuation.
- Directional and spot lights have a direction.
Adding backgrounds

Motivation

Using the background components

Syntax: Background

A sample background

Using the background components

Syntax: Background

A sample background image

A sample background

Summary

Shapes form the foreground of your scene

You can add a background to provide context

Backgrounds describe:

- Sky and ground colors
- Panorama images of mountains, cities, etc

Backgrounds are faster to draw than if you used shapes to build them
Adding backgrounds

Using the background components

- A background creates three special shapes:
  - A sky sphere
  - A ground sphere inside the sky sphere
  - A panorama box inside the ground sphere

- The sky and ground spheres are shaded with a color gradient

- The panorama box is texture mapped with six images

- Transparent parts of the ground sphere reveal the sky sphere

- Transparent parts of the panorama box reveal the ground and sky spheres

- The viewer can look up, down, and side-to-side to see different parts of the background

- The viewer can never get closer to the background
Adding backgrounds

*Syntax: Background*

- A **Background** node describes background colors
  - *ground colors and angles* - ground gradation
  - *sky colors and angles* - sky gradation
  - *more* . . .

```wrl
Background {
 groundColor [ 0.0 0.2 0.7, . . . ]
groundAngle [ 1.309, 1.571 ]
skyColor [ 0.1 0.1 0.0, . . . ]
skyAngle [ 1.309, 1.571 ]
}
```

A sample background

[ back.wrl ]
Adding backgrounds

Syntax: `Background`

- A `Background` node describes background images
  - `frontUrl` - texture image URL for box front
  - etc...

```plaintext
Background {
  frontUrl  "mountns.png"
  backUrl   "mountns.png"
  leftUrl   "mountns.png"
  rightUrl  "mountns.png"
  topUrl    "clouds.png"
  bottomUrl "ground.png"
}
```

A sample background image
Adding backgrounds

A sample background

[ back2.wrl ]

Summary

- Backgrounds describe:
  - Ground and sky color gradients on ground and sky spheres
  - Panorama images on a panorama box
- The viewer can look around, but never get closer to the background
Motivation

Fog increases realism:
- Add fog outside to create hazy worlds
- Add fog inside to create dark dungeons
- Use fog to set a mood

The further the viewer can see, the more you have to model and draw

To reduce development time and drawing time, limit the viewer’s sight by using fog
Adding fog

Examples

[ fog2.wrl ]

[ fog4.wrl ]

Using fog visibility controls

- The **fog type** selects linear or exponential visibility reduction with distance
  - Linear is easier to control
  - Exponential is more realistic and "thicker"

- The **visibility range** selects the distance where the fog reaches maximum thickness
  - Fog is "clear" at the viewer, and gradually reduces visibility
Adding fog

Selecting a fog color

- Fog has a *fog color*
  - White is typical, but black, red, etc. also possible

- *Shapes* are faded to the fog color with distance

- The background is unaffected
  - For the best effect, make the background the fog color

Syntax: Fog

- A `Fog` node creates colored fog
  - `color` - fog color
  - `type` - fog type
  - `visibility range` - maximum visibility limit

```
Fog {
  color 1.0 1.0 1.0
  fogType "LINEAR"
  visibilityRange 0.0
}
```
Adding fog

Several fog samples

[ fog1.wrl ]

[ fog2.wrl ]

[ fog3.wrl ]

Summary

- Fog has a color, a type, and a visibility range

- Fog can be used to set a mood, even indoors

- Fog limits the viewer’s sight:
  - Reduces the amount of the world you have to build
  - Reduces the amount of the world that must be drawn
Motivation

Creating sounds

Syntax: AudioClip
Syntax: MovieTexture

Selecting sound source types

Syntax: Sound
Syntax: Sound

Setting the sound range

Creating triggered sounds

A sample using triggered sound
A sample using triggered sound

Creating continuous localized sounds

Creating continuous background sounds

A sample using continuous localized sound
A sample using continuous localized sound

Summary

Motivation

- Sounds can be triggered by viewer actions
  - Clicks, horn honks, door latch noises

- Sounds can be continuous in the background
  - Wind, crowd noises, elevator music

- Sounds emit from a location, in a direction, within an area
Creating sounds

- Sounds have two components
  - A *sound source* providing a sound signal
    - Like a stereo component
  - A *sound emitter* converts a signal to virtual sound
    - Like a stereo speaker

Syntax: AudioClip

- An `AudioClip` node creates a digital sound source
  - `url` - a sound file URL
  - `pitch` - playback speed
  - playback controls, like a `TimeSensor` node

```plaintext
AudioClip {
  url "myfile.wav"
  pitch 1.0
  startTime 0.0
  stopTime 0.0
  loop FALSE
}
```
Adding sound

**Syntax: MovieTexture**

- A MovieTexture node creates a movie sound source
  - `url` - a texture movie file URL
  - `speed` - playback speed
  - playback controls, like a TimeSensor node

```plaintext
MovieTexture {
  startTime 0.0
  stopTime 0.0
  loop FALSE
  speed 1.0
  url "movie.mpg"
}
```

Selecting sound source types

- Supported by the AudioClip node:
  - `WAV` - digital sound files
    - Good for sound effects
  - `MIDI` - General MIDI musical performance files
    - MIDI files are good for background music

- Supported by the MovieTexture node:
  - `MPEG` - movie file with sound
    - Good for virtual TVs
Adding sound

**Syntax: Sound**

- A **Sound** node describes a sound emitter
  - **source** - AudioClip OR MovieTexture node
  - **location** and **direction** - emitter placement
  - more ...

    ```
    Sound {
      source AudioClip { . . . }
      location  0.0 0.0 0.0
      direction 0.0 0.0 1.0
    }
    ```

- A **Sound** node describes a sound emitter
  - **intensity** - volume
  - **spatialize** - use spatialize processing
  - **priority** - prioritize the sound
  - more ...

    ```
    Sound {
      . . .
      intensity 1.0
      spatialize TRUE
      priority 0.0
    }
    ```
Adding sound

**Syntax: Sound**

- A Sound node describes a sound emitter
  - *minimum* and *maximum* range - area in which sound can be heard

  ```
  Sound {
    . . .
    minFront 1.0
    minBack 1.0
    maxFront 10.0
    maxBack 10.0
  }
  ```

Adding sound

**Setting the sound range**

- The sound range fields specify two *ellipsoids*
  - `minFront` and `minFront` control an inner ellipsoid
  - `maxFront` and `maxFront` control an outer ellipsoid

- Sound has a constant volume inside the inner ellipsoid

- Sound drops to zero volume from the inner to the outer ellipsoid
Creating triggered sounds

- AudioClip node:
  - loop FALSE
  - Set startTime from a sensor node

- Sound node:
  - spatialize TRUE
  - minFront etc. with small values
  - priority 1.0

A sample using triggered sound

Sound {
  source DEF C4 AudioClip {
    url "tone1.wav"
    pitch 1.0
  }
}
ROUTE Touch.touchTime TO C4.set(startTime)
Adding sound

A sample using triggered sound

[ kbd.wrl ]

Creating continuous localized sounds

- AudioClip node:
  - loop TRUE
  - startTime 0.0 (default)
  - stopTime 0.0 (default)

- Sound node:
  - spatialize TRUE (default)
  - minFront etc. with medium values
  - priority 0.0 (default)
Adding sound

Creating continuous background sounds

- AudioClip node:
  - loop TRUE
  - startTime 0.0 (default)
  - stopTime 0.0 (default)

- Sound node:
  - spatialize FALSE (default)
  - minFront etc. with large values
  - priority 0.0 (default)

A sample using continuous localized sound

```json
Sound {
  source AudioClip {
    url "willow1.wav"
    loop TRUE
  }
}
```
Adding sound

A sample using continuous localized sound

[ ambient.wrl ]

Summary

- An AudioClip node or a MovieTexture node describe a sound source
  - A URL gives the sound file
  - Looping, start time, and stop time control playback

- A Sound node describes a sound emitter
  - A source node provides the sound
  - Range fields describe the sound volume
Motivation
Creating viewpoints
Syntax: Viewpoint
Summary

* By default, the viewer enters a world at (0.0, 0.0, 10.0)

* You can provide your own preferred viewpoints
  * Select the entry point position
  * Select favorite views for the viewer
  * Name the views for a browser menu
Creating viewpoints

- Viewpoints specify a desired location, an orientation, and a camera field of view lens angle

- Viewpoints can be transformed using a Transform node

- The first viewpoint found in a file is the entry point

Syntax: Viewpoint

- A viewpoint node specifies a named viewing location
- position and orientation - viewing location
- fieldOfView - camera lens angle
- description - description for viewpoint menu

Viewpoint {
  position 0.0 0.0 10.0
  orientation 0.0 0.0 1.0 0.0
  fieldOfView 0.785
  description "Entry View"
}
Controlling the viewpoint

Summary

- Specify favorite viewpoints in viewpoint nodes
- The first viewpoint in the file is the entry viewpoint
Motivation

Different types of worlds require different styles of navigation
  - Walk through a dungeon
  - Fly through a cloud world
  - Examine shapes in a CAD application

You can select the navigation type

You can describe the size and speed of the viewer’s avatar
Selecting navigation types

- There are five standard navigation keywords:
  - *walk* - walk, pulled down by gravity
  - *fly* - fly, unaffected by gravity
  - *examine* - examine an object at "arms length"
  - *none* - no navigation, movement controlled by world not viewer!
  - *any* - allows user to change navigation type

- Some browsers support additional navigation types

Specifying avatars

- Avatar size (width, height, step height) and speed can be specified
Controlling navigation

Controlling the headlight

- By default, a *headlight* is placed on the avatar’s head and aimed in the head direction.

- You can turn this headlight on and off.
  - Most browsers provide a menu option to control the headlight.
  - You can also control the headlight with the `NavigationInfo` node.

Syntax: `NavigationInfo`

- A `NavigationInfo` node selects the navigation type and avatar characteristics:
  - `type` - navigation style
  - `avatarSize` and `speed` - avatar characteristics
  - `headlight` - headlight on or off

```plaintext
NavigationInfo {
  type       [ "WALK", "ANY" ]
  avatarSize [ 0.25, 1.6, 0.75 ]
  speed      1.0
  headlight  TRUE
}
```
Controlling navigation

Summary

- The navigation type specifies how a viewer can move in a world
  - walk, fly, examine, or none

- The avatar overall size and speed specify the viewer’s avatar characteristics
Motivation

Sensing the viewer enables you to trigger animations
  • when a region is visible to the viewer
  • when the viewer is within a region
  • when the viewer collides with a shape

The LOD and Billboard nodes are special-purpose viewer sensors with built-in responses

Using visibility and proximity sensors
Syntax: ProximitySensor
Syntax: ProximitySensor
Syntax: VisibilitySensor

A sample use of a proximity sensor
Detecting viewer-shape collision
Creating collision groups
Syntax: Collision
A sample use of a collision group
Optimizing collision detection
Using multiple sensors
Summary
Summary
Summary
There are three types of viewer sensors:

- A `VisibilitySensor` node senses if the viewer can see a region.
- A `ProximitySensor` node senses if the viewer is within a region.
- A `Collision` node senses if the viewer has collided with shapes.

Using visibility and proximity sensors:

- `VisibilitySensor` and `ProximitySensor` nodes sense a box-shaped region:
  - `center` - region center
  - `size` - region dimensions

- Both nodes have similar outputs:
  - `enterTime` - sends time on visible or region entry
  - `exitTime` - sends time on not visible or region exit
  - `isActive` - sends true on entry, false on exit
Sensing the viewer

Syntax: ProximitySensor

- A ProximitySensor node senses if the viewer enters or leaves a region
  - *center* and *size* - the region’s location and size
  - *enterTime* and *exitTime* - sends time on entry/exit
  - *isActive* - sends true/false on entry/exit
  - more...

```plaintext
DEF DoorSense ProximitySensor {
    center 0.0 1.75 0.0
    size 6.0 3.5 8.0
}
ROUTE DoorSense.enterTime
    TO OpenSound.set_startTime

DEF DoorSense ProximitySensor {
    . . .
}
ROUTE DoorSense.position_changed
    TO PetRobotFollower.set_translation
```

- A ProximitySensor node senses the viewer while in a region
  - *position* and *orientation* - sends position and orientation while viewer is in the region

```plaintext
```

```
```
Sensing the viewer

**Syntax: VisibilitySensor**

- A *VisibilitySensor* node senses if the viewer can see a region
  - *center* and *size* - the region’s location and size
  - *enterTime* and *exitTime* - sends time on entry/exit
  - *isActive* - sends true/false on entry/exit

```wrl
DEF DoorSense VisibilitySensor {
    center 0.0 1.75 0.0
    size 3.0 2.5 1.0
}
ROUTE DoorSense.enterTime
    TO OpenSound.set_startTime
```

A sample use of a proximity sensor

[ prox1.wrl ]
Detecting viewer-shape collision

- A Collision grouping node senses shapes within the group
  - Detects if the viewer collides with any shape in the group
  - Automatically stops the viewer from going through the shape

- Collision occurs when the viewer’s avatar gets close to a shape
  - Collision distance is controlled by the avatar size in the NavigationInfo node

Creating collision groups

- Collision checking is expensive so, check for collision with a proxy shape instead
  - Proxy shapes are typically extremely simplified versions of the actual shapes
  - Proxy shapes are never drawn

- A collision group with a proxy shape, but no children, creates an invisible collidable shape
  - Windows and invisible railings
  - Invisible world limits
A Collision grouping node senses if the viewer collides with group shapes

- **collide** - enable/disable sensor
- **children** - children to sense
- **proxy** - simple shape to sense instead of children

```wrl
DEF DoorCollide Collision {
    proxy . . .
    children [ . . . ]
}
ROUTE DoorCollide.collideTime
    TO OpenSound.set_startTime
```
Sensing the viewer

Optimizing collision detection

- Collision is on by default
  - Turn it off whenever possible!

- However, once a parent turns off collision, a child can’t turn it back on!

- Collision results from viewer colliding with a shape, but not from a shape colliding with a viewer

Using multiple sensors

- Any number of sensors can sense at the same time
  - You can have multiple visibility, proximity, and collision sensors

- Sensor areas can overlap

- If multiple sensors should trigger, they do
Summary

- A `VisibilitySensor` node checks if a region is visible to the viewer
  - The region is described by a center and a size
  - Time is sent on entry and exit of visibility
  - True/false is sent on entry and exit of visibility

Summary

- A `ProximitySensor` node checks if the viewer is within a region
  - The region is described by a center and a size
  - Time is sent on viewer entry and exit
  - True/false is sent on viewer entry and exit
  - Position and orientation of the viewer is sent while within the sensed region
Sensing the viewer

Summary

- A Collision grouping node checks if the viewer has run into a shape
  - The shapes are defined by the group’s children or a proxy

- Collision time is sent on contact
A doorway
A mysterious temple

Summary examples

- A set of ImageTexture nodes add marble textures
- Lighting nodes create dramatic lighting
- A Fog node fades distant shapes
- A ProximitySensor node controls animation

[ doorway.wrl ]
Summary examples

* A mysterious temple

- A Background node creates a sky gradient
- A Sound node creates a spatialized sound effect
- A set of viewpoint nodes provide standard views

[ temple.wrl ]
Motivation

Example

Creating multiple shape versions

Controlling level of detail

Choosing detail ranges

Syntax: LOD

Optimizing a shape

A sample of detail versions

A sample LOD

A sample LOD

Summary

Motivation

- The further the viewer can see, the more there is to draw

- If a shape is distant:
  - The shape is smaller
  - The viewer can’t see as much detail
  - So... draw it with less detail

- Varying detail with distance reduces upfront download time, and increases drawing speed
Creating multiple shape versions

- To control detail, model the *same shape* several times
  - high detail for when the viewer is close up
  - medium detail for when the viewer is nearish
  - low detail for when the viewer is distant

- Usually, two or three different versions is enough, but you can have as many as you want
Controlling detail

Controlling level of detail

- Group the shape versions as levels in an LOD grouping node
  - LOD is short for Level of Detail
  - List them from highest to lowest detail

- Give the entire group a center point

Choosing detail ranges

- Use a list of ranges for version switch points
  - If you have 3 versions, you need 2 ranges
  - Ranges are hints to the browser

<table>
<thead>
<tr>
<th>range</th>
<th>1st child used</th>
<th>2nd child used</th>
<th>3rd child used</th>
</tr>
</thead>
<tbody>
<tr>
<td>viewer &lt; 7.5</td>
<td>1st child used</td>
<td>2nd child used</td>
<td>3rd child used</td>
</tr>
<tr>
<td>7.5 &lt;= viewer &lt; 12.0</td>
<td>2nd child used</td>
<td>3rd child used</td>
<td></td>
</tr>
<tr>
<td>12.0 &lt; viewer</td>
<td>3rd child used</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Controlling detail

Syntax: LOD

- An LOD grouping node creates a group of shapes describing different versions of the same shape
  - center - the center of the shape
  - range - a list of version switch ranges
  - level - a list of shape versions

```plaintext
LOD {
    center 0.0 0.0 0.0
    range [ . . . ]
    level [ . . . ]
}
```

Optimizing a shape

- Suggested procedure to make different versions:
  - Make the high detail shape first
  - Copy it to make a medium detail version
  - Move the medium detail shape to a desired switch distance
  - Delete parts that aren’t dominant
  - Repeat for a low detail version

- Lower detail versions should use simpler geometry, fewer textures, and no text
A sample of detail versions

[ torches3.wrl ]

Controlling detail

A sample LOD

LOD {
    center 0.0 0.0 0.0
    range [ 7.5, 12.0 ]
    level [
        Inline { url "torch1.wrl" }  
        Inline { url "torch2.wrl" }  
        Inline { url "torch3.wrl" }  
    ]
}
A sample LOD

Summary

- Increase performance by making multiple versions of shapes
  - High detail for close up viewing
  - Lower detail for more distant viewing
- Group the versions in an LOD node
  - Ordered from high detail to low detail
  - Ranges to select switching distances
Motivation

Many actions are too complex for animation nodes
  - Computed animation paths (eg. gravity)
  - Algorithmic shapes (eg. fractals)
  - Collaborative environments (eg. games)

You can create new sensors, interpolators, etc., using program scripts written in
  - *Java* - powerful general-purpose language
  - *JavaScript* - easy-to-learn language
  - *VRMLscript* - same as JavaScript
Introducing script use

**Syntax: Script**

- A **Script** node selects a program script to run:
  - *url* - choice of program script

```plaintext
DEF MyScript Script {
    url "myscript.class"
    or...
    url "myscript.js"
    or...
    url "javascript: ..."
    or...
    url "vrmlscript: ..."
}
```

---

Introducing script use

**Defining the program script interface**

- A **Script** node also declares the program script interface
  - *fields* and *events* - ins and outs
    - Each has a name and data type
    - Fields have an initial value

```plaintext
DEF Bouncer Script {
    field SFFloat bounceHeight 3.0
    eventIn SFFloat set_fraction
    eventOut SFVec3f value_changed
}
```
Introducing script use

A sample using a program script

DEF Bouncer Script { . . . }

ROUTE Clock.fraction_changed
    TO Bouncer.set_fraction

ROUTE Bouncer.value_changed
    TO Ball.set_translation

[ bounce1.wrl ]
Summary

- The `Script` node selects a program script, specified by a URL.

- Program scripts have field and event interface declarations, each with:
  - A data type
  - A name
  - An initial value (fields only)
Writing program scripts with JavaScript

Motivation
Declaring a program script interface
Initializing a program script
Shutting down a program script
Responding to events
Processing events in JavaScript
Accessing fields from JavaScript
Accessing event Outs from JavaScript
A sample JavaScript script
A sample JavaScript script
A sample JavaScript script
A sample JavaScript script
A sample JavaScript script
A sample JavaScript script
A sample JavaScript script
A sample JavaScript script
A sample JavaScript script
Building user interfaces
Building a toggle switch
Using a toggle switch
Using a toggle switch
Using a toggle switch
Building a color selector
Using a color selector

Summary
Motivation

- A program script implements the `Script` node using values from the interface
  - The script responds to inputs and sends outputs

- A program script can be written in Java, JavaScript, and other languages
  - JavaScript is easier to program
  - Java is more powerful

Declaring a program script interface

- For a JavaScript program script, typically give the script in the `Script` node’s `url` field

```javascript
DEF Bouncer Script {
    field SFFloat bounceHeight 3.0
    eventIn SFFloat set_fraction
    eventOut SFVec3f value_changed
    url "javascript: . . ."
}
```
Initializing a program script

- The optional `initialize` function is called when the script is loaded

```javascript
function initialize() {
    //...
}
```

- Initialization occurs when:
  - the `Script` node is created (typically when the browser loads the world)

Shutting down a program script

- The optional `shutdown` function is called when the script is unloaded

```javascript
function shutdown() {
    //...
}
```

- Shutdown occurs when:
  - the `Script` node is deleted
  - the browser loads a new world
Responding to events

- An *eventIn function* must be declared for each eventIn

- The eventIn function is called each time an event is received, passing the event’s
  - value
  - time stamp

function set_fraction( value, timestamp ) 

}{

If multiple events arrive at once, then multiple eventIn functions are called

The optional `eventsProcessed` function is called after all (or some) eventIn functions have been called

function eventsProcessed ( ) {
  
  
}
Accessing fields from JavaScript

- Each interface field is a JavaScript variable
  - Read a variable to access the field value
  - Write a variable to change the field value

```javascript
lastval = bounceHeight; # get field
bounceHeight = newval;  # set field
```

Accessing eventOuts from JavaScript

- Each interface eventOut is a JavaScript variable
  - Read a variable to access the last eventOut value
  - Write a variable to send an event on the eventOut

```javascript
lastval = value_changed[0];   # get last ev
value_changed[0] = newval;   # send new ev
```
• Create a *Bouncing ball interpolator* that computes a gravity-like vertical bouncing motion from a fractional time input

• Fields needed:
  • Bounce height

```javascript
DEF Bouncer Script {
  field SFFloat bounceHeight 3.0

  // Code here...

}
```

• Inputs and outputs needed:
  • Fractional time input
  • Position value output

```javascript
DEF Bouncer Script {

  eventIn SFFloat set_fraction
  eventOut SFVec3f value_changed

  // Code here...

}
```
A sample JavaScript script

- Initialization and shutdown actions needed:
  - None - all work done in eventIn function

- Event processing actions needed:
  - set_fraction eventIn function
  - No need for eventsProcessed function

DEF Bouncer Script {
    . . .
    url "javascript:
        function set_fraction( frac, tm )
            . . .
        }
    . . .
}
Writing program scripts with JavaScript

*A sample JavaScript script*

- Calculations needed:
  - Compute new ball position
  - Send new position event

- Use a ball position equation roughly based upon Physics
  - See comments in the VRML file for the derivation of the equation

```javascript
function set_fraction( frac, tm ) {
    y = 4.0 * bounceHeight * frac * (1.0 -
    value_changed[0] = 0.0;
    value_changed[1] = y;
    value_changed[2] = 0.0;
}
```
A sample JavaScript script

- Routes needed:
  - Clock into script’s set_fraction
  - Script’s value_changed into transform

ROUTE Clock.fraction_changed
    TO Bouncer.set_fraction

ROUTE Bouncer.value_changed
    TO Ball.set_translation

[ bounce1.wrl ]
Writing program scripts with JavaScript

Building user interfaces

- Program scripts can be used to help create 3D user interface widgets
  - Toggle buttons
  - Radio buttons
  - Rotary dials
  - Scrollbars
  - Text prompts
  - Debug message text

Building a toggle switch

- A toggle switch script turns on at the first touch, and off at the second
  - A TouchSensor node can supply the touch events

DEF Toggle Script {
  field    SFBool on TRUE
  eventIn  SFTime set_active
  eventOut SFBool on_changed
  url "vrmlscript:
    function set_active( b, tm ) {
      if ( b == FALSE ) return;
      if ( on == TRUE ) on = FALSE;
      else              on = TRUE;
      on_changed = on;
    }
  }"
Using a toggle switch

- Use the toggle switch to make a lamp turn on and off
  - Use a TouchSensor node to sense a switch shape
- Route the sensor node’s isActive eventOut into the script node’s set_active eventIn
- Route the script node’s on_changed eventOut into the light node’s set_on eventIn
Building a color selector

- The lamp in the previous example turns on and off, but the light bulb doesn’t change color!

- A color selector script sends an *on* color on a **TRUE** input, and an *off* color on a **FALSE** input

```javascript
DEF ColorSelector Script {
    field SFColor onColor 1.0 1.0 1.0
    field SFColor offColor 0.0 0.0 0.0
    eventIn SFBool set_selection
    eventOut SFColor color_changed
    url "vrmlscript:
        function set_selection( b, tm ) {
            if ( b == TRUE ) color_changed
            else            color_changed
        }
    }
}
```

Using a color selector

- Use the color selector to change the lamp bulb color
  - Route the toggle script node’s *on_changed* eventOut into the selector script node’s *set_selection* eventIn

- Route the selector script node’s *color_changed* eventOut into the bulb Material node’s *set_emissiveColor* eventIn
Writing program scripts with JavaScript

Using a color selector

Summary

- The initialize and shutdown functions are called at load and unload

- An eventIn function is called when an event is received

- The eventsProcessed function is called after all (or some) events have been received

- Functions can get field values and send event outputs
Motivation

Compared to JavaScript, Java enables:

- Better modularity
- Better data structures
- Potential for faster execution
- Access to the network

For simple tasks, use JavaScript
For complex tasks, use Java
**Declaring a program script interface**

- For a Java program script, give the class file in the `Script` node’s `url` field.
  - A class file is a compiled Java program script.

```java
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed
    url "bounce2.class"
}
```

**Creating the Java class**

- The program script file must import the VRML packages:
  ```java
  import vrml.*;
  import vrml.field.*;
  import vrml.node.*;
  ```

- The program script must define a public class that extends the `Script` class:
  ```java
  public class bounce2
  {
      extends Script
      {
          . . .
      }
  }
  ```
Initializing a program script

- The optional `initialize` method is called when the script is loaded
  
  ```java
  public void initialize () {
      ...
  }
  ```

- Initialization occurs when:
  - the `Script` node is created (typically when the browser loads the world)

Shutting down a program script

- The optional `shutdown` method is called when the script is unloaded
  
  ```java
  public void shutdown () {
      ...
  }
  ```

- Shutdown occurs when:
  - the `Script` node is deleted
  - the browser loads a new world
The `processEvent` method is called each time an event is received, passing an `Event` object containing the event’s
- value
- time stamp

```java
public void processEvent( Event event ) {
    ...
}
```

If multiple events arrive at once, then the `processEvent` method is called multiple times.

The optional `eventsProcessed` method is called after all (or some) events have been handled.

```java
public void eventsProcessed() {
    ...
}
```
Writing program scripts with Java

Accessing fields from Java

- Each interface field can be read and written
  - Call `getField` to get a field object
    
    ```java
    obj = (SFFloat) getField( "bounceHeight" );
    ```

  - Call `getValue` to get a field value
    
    ```java
    lastval = obj.getValue();
    ```

  - Call `setValue` to set a field value
    
    ```java
    obj.setValue( newval );
    ```

Accessing eventOuts from Java

- Each interface eventOut can be read and written
  - Call `getEventOut` to get an eventOut object
    
    ```java
    obj = (SFVec3f) getEventOut( "value_color" );
    ```

  - Call `getValue` to get the last event sent
    
    ```java
    lastval = obj.getValue();
    ```

  - Call `setValue` to send an event
    
    ```java
    obj.setValue( newval );
    ```
• Create a *Bouncing ball interpolator* that computes a gravity-like vertical bouncing motion from a fractional time input

• Give it the same interface as the JavaScript example

```java
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed
    url "bounce2.class"
}
```

• Imports and class definition needed:

```java
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class bounce2
    extends Script
{
    ...
}
```
A sample Java script

- **Class variables needed:**
  - One for the `bounceHeight` field
  - One for the `value_changed` eventOut object

    ```java
    private float bounceHeight;
    private SFVec3f value_changedObj;
    ```

- **Initialization actions needed:**
  - Get the value of the `bounceHeight` field
  - Get the `value_changedObj` eventOut object

    ```java
    public void initialize() {
        SFFloat obj = (SFFloat) getField("bounceHeight");
        bounceHeight = (float) obj.getValue();
        value_changedObj = (SFVec3f) getEventOut();
    }
    ```
• Shutdown actions needed:
  • None - all work done in `processEvent` method

• Event processing actions needed:
  • `processEvent` event method
  • No need for `eventsProcessed` method

```java
public void processEvent( Event event )
{
    . . .
}
```
- Calculations needed:
  - Compute new ball position
  - Send new position event

```java
public void processEvent( Event event )
{
    ConstSFFloat flt = (ConstSFFloat) event;
    float frac = (float) flt.getValue();

    float y = (float)(4.0 * bounceHeight * 

    float[] changed = new float[3];
    changed[0] = (float)0.0;
    changed[1] = y;
    changed[2] = (float)0.0;
    value_changedObj.setValue( changed );
}
```
Routes needed:
- Clock into script’s `set_fraction`
- Script’s `value_changed` into `transform`

ROUTE `Clock.fraction_changed` TO `Bouncer.set_fraction`
ROUTE `Bouncer.value_changed` TO `Ball.set_translation`
Summary

- The `initialize` and `shutdown` methods are called at load and unload.

- The `processEvent` method is called when an event is received.

- The `eventsProcessed` method is called after all (or some) events have been received.

- Methods can get field values and send event outputs.
Creating new node types

Motivation

You can create new node types that encapsulate:
- Shapes
- Sensors
- Interpolators
- Scripts
- anything else . . .

This creates high-level nodes
- Robots, menus, new shapes, etc.

Syntax: PROTO
Defining prototype bodies
Syntax: IS
Using IS
Using prototyped nodes
Controlling usage rules
A sample prototype use
A sample prototype use
A sample prototype use
A sample prototype use
A sample prototype use
Changing a prototype
A sample prototype use
Syntax: EXTERNPROTO
Summary
Creating new node types

Syntax: PROTO

- A PROTO statement declares a new node type
  - name - the new node type name
  - fields and events - interface to the prototype

PROTO BouncingBall [
    field SFFloat bounceHeight 1.0
    field SFTime cycleInterval 1.0
] { . . . }

Creating new node types

Defining prototype bodies

- PROTO defines:
  - body - nodes and routes for the new node type

PROTO BouncingBall [ . . . ] {
    Transform {
        children [ . . . ]
    }  
    ROUTE . . .
}
Creating new node types

**Syntax: IS**

- The **IS** syntax connects a prototype interface field, `eventIn`, or `eventOut` to the body.

```
PROTO BouncingBall [
    field SFFloat bounceHeight 1.0
    field SFTime cycleInterval 1.0
] {
    . . .
    DEF Clock TimeSensor {
        cycleInterval IS cycleInterval
        . . .
    }    . . .
}
```

**Using IS**

![Table showing possible connections with IS]

<table>
<thead>
<tr>
<th>Interface</th>
<th>Fields</th>
<th>Exposed fields</th>
<th>EventIns</th>
<th>EventOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Exposed fields</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>EventIns</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>EventOuts</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Creating new node types

Using prototyped nodes

- The new node type can be used like any other type

```plaintext
BouncingBall {
    bounceHeight 3.0
    cycleInterval 2.0
}
```

Creating new node types

Controlling usage rules

- Recall that node use must be appropriate for the context
  - A Shape node specifies shape, not color
  - A Material node specifies color, not shape
  - A Box node specifies geometry, not shape or color
Creating new node types

Controlling usage rules

- The context for a new node type depends upon the first node in the PROTO body

- For example, if the first node is a geometry node:
  - The prototype creates a new geometry node type

- The new node type can be used wherever the first node of the prototype body can be used

A sample prototype use

- Create a BouncingBall node type that:
  - Builds a beachball
  - Creates an animation clock
    - Using a PROTO field to select the cycle interval
  - Bounces the beachball
    - Using the bouncing ball program script
    - Using a PROTO field to select the bounce height
Creating new node types

A sample prototype use

- Fields needed:
  - Bounce height
  - Cycle interval

PROTO BouncingBall [ 
    field SFFloat bounceHeight 1.0
    field SFTime cycleInterval 1.0
  ] {...}

- Inputs and outputs needed:
  - None - a TimeSensor node is built in to the new node
Creating new node types

*A sample prototype use*

- Body needed:
  - A ball shape inside a transform
  - An animation clock
  - A bouncing ball program script
  - Routes connecting it all together

```wrl
PROTO BouncingBall [ . . . ] {
    DEF Ball Transform {
        children [
            Shape { . . . }
        ]
    }
    DEF Clock   TimeSensor { . . . }
    DEF Bouncer Script { . . . }
    ROUTE . . .
}
```

[ bounce3.wrl ]
Creating new node types

Changing a prototype

- If you change a prototype, all uses of that prototype change as well
  - Prototypes enable world modularity
  - Large worlds make heavy use of prototypes

- For the BouncingBall prototype, adding a shadow to the prototype makes all balls have a shadow

A sample prototype use

[ bounce4.wrl ]
Creating new node types

Syntax: `EXTERNPROTO`

- Prototypes are typically in a separate external file

- An `EXTERNPROTO` declares a new node type in an external file
  - `name, fields, events` - as from `PROTO`
  - `url` - the URL of the prototype file

```
EXTERNPROTO BouncingBall [ 
    field SFFloat bounceHeight 1.0 
    field SFTime  cycleInterval 1.0 
]  "bounce.wrl#BouncingBall"
```

Summary

- `PROTO` declares a new node type and defines its node body

- `EXTERNPROTO` declares a new node type, specified by URL

- The new node anywhere the `first` node in the prototype body can be used
Motivation

Syntax: WorldInfo

- After you’ve created a great world, sign it!
- You can provide a title and a description embedded within the file
Providing information about your world

**Syntax: WorldInfo**

- A `WorldInfo` node provides title and description information for your world
  - `title` - the name for your world
  - `info` - any additional information

```plaintext
WorldInfo {
    title "My Masterpiece"
    info [ "Copyright (c) 1997 Me." ]
}
```
An animated flame node

A torch node

---

Summary examples

**An animated flame node**

- A `script` node cycles between flame textures
- A `proto` encapsulates the flame shape, script, and routes into a `Flames` node

[ match.wrl ]
Summary examples

**A torch node**

- A Flame node creates animated flame
- An LOD node selects among torches using the flame
- A Proto encapsulates the torches into a Torch node

[ columns.wrl ]
Several VRML extensions are in progress

- Binary file format
- External authoring interface
- Multi-user framework
Using the binary file format

- The binary file format enables smaller files for faster download

- The binary file format includes
  - Binary representation of nodes and fields
  - Support for prototypes
  - Geometry compression

Most authoring will be done with world builders that output binary VRML files directly

- Hand-authored text VRML will be compiled to the binary format

- Converters back to text VRML will become available
  - Comments will be lost by translation
  - WorldInfo nodes will be retained
Using the external authoring interface

- Program scripts in a Script node are Internal
  - Inside the world
  - Connected by routes

- External program scripts can be written in Java using the External Authoring Interface (EAI)
  - Outside the world, on an HTML page
  - No need to use routes!

A typical Web page contains:
- HTML text
- An embedded VRML browser plug-in
- A Java applet

The EAI enables the Java applet to "talk" to the VRML browser

The EAI is not part of the VRML standard (yet), but it is widely supported
- Check your browser’s release notes for EAI support
Using the multi-user framework

- Several extensions are in progress to create a framework for multi-user worlds
  - Shared objects and spaces
  - Piloted objects (like avatars)
  - Common avatar descriptions
This morning we covered:

- Building primitive shapes
- Building complex shapes
- Translating, rotating, and scaling shapes
- Controlling appearance
- Grouping shapes
- Animating transforms
- Interpolating values
- Sensing viewer actions
Coverage

This afternoon we covered:
- Controlling texture
- Controlling shading
- Adding lights
- Adding backgrounds and fog
- Controlling detail
- Controlling viewing
- Adding sound
- Sensing the viewer
- Using and writing program scripts
- Building new node types

Where to find out more

- The VRML 2.0 specification
  http://vag.vrml.org/VRML2.0/FINAL

- The VRML 97 specification
  http://vrml.sgi.com/moving-worlds

- The VRML Repository
  http://www.sdsc.edu/vrml
Conclusion

Introduction to VRML 97

Thanks for coming!
IDG’s NetscapeWorld on-line monthly magazine publishes articles on Web technologies and trends, including articles on Web browsers, Web servers, development tools, push technologies, HTML, Java, JavaScript, and VRML. In his regular VRML Technique column, David R. Nadeau writes about VRML world-building technique, market trends, and VRML technology news.

Included here are reprints of four introductory VRML Technique columns published by NetscapeWorld in December 1996 through March 1997. These articles provide detailed tutorials on beginning VRML nodes, including those for creating predefined shapes, positioning, orienting, and scaling shapes, creating animations, and sensing the viewer.

Additional VRML Technique columns, as well as other articles on Web technologies, are available at NetscapeWorld magazine’s Web site:

http://www.netscapeworld.com

VRML Technique column reprints

Columns
See what VRML 2.0 is all about and start building shapes today
The first in a series, we introduce VRML’s shape-building features to create boxes, cylinders, cones, and spheres. (December 1996)

Building virtual structures
How to position, orient, and resize shapes in VRML 2.0. (January 1997)

Animating shapes
How to animate the position, orientation, and size of shapes in VRML 2.0. (February 1997)

Sensing the viewer’s touch
How to sense the viewer’s touch to start and stop animations in VRML 2.0. (March 1997)

Sidebars
How to view VRML 2.0
Finding and installing the right VRML browser for your computer

The UTF-8 character set
VRML 2.0’s international character set

VRML 2.0 glossary
The key terms you need to know to get started with VRML
See what VRML 2.0 is all about and start building shapes today

The first in a series, we introduce VRML’s shape-building features to create boxes, cylinders, cones, and spheres

By David R. Nadeau

Summary
In August 1996, the members of the VRML community completed the eagerly-anticipated specification for VRML 2.0. This latest version dramatically extends the popular 3-D content language, updating it to enable faster drawing and introducing new features for interaction, animation, scripting, sounds, and much more!

Beginning with this issue, Netscape World introduces a new monthly column: VRML Technique. Written for the beginning 3-D content author, each month’s column introduces new VRML 2.0 features, explains their use and syntax, and provides tips and techniques for efficient and creative authoring.

This month’s VRML Technique column introduces the VRML 2.0 language, discusses key language concepts (file header, nodes, fields, and values), and provides syntax and examples for VRML 2.0’s shape building primitives (box, cone, cylinder, and sphere). (6,000 words)

Table of contents
Building VRML 2.0 worlds

Version 2.0 of VRML, the Virtual Reality Modeling Language, is a rich text language for the description of 3-D interactive virtual worlds. Like version 1.0, version 2.0 of VRML enables you to build complex, realistic 3-D environments, complete with shiny materials, textured surfaces, and multiple light sources. VRML 2.0’s new features enable you to make your worlds come alive with embedded animations and sound tracks. VRML 2.0 worlds can sense the viewer’s touch, position, and gaze direction, trigger sounds and animations on viewer proximity, fly the viewer on a guided tour of the world, and even communicate with other applications and users on the Internet.

You can author VRML 2.0 worlds using any text editor or word processor on PCs, Macintoshes, and Unix systems. World builder applications, just now entering the market, enable you to author VRML 2.0 worlds within an interactive 3-D drawing environment.

Once authored, you can view your worlds using a VRML browser. VRML browsers are available as plug-ins to Netscape Navigator, add-ins to Microsoft Internet Explorer, or as stand-alone helper-applications for any Web browser. Several VRML 2.0 browsers are available now for PCs running Windows 95 or Windows NT, or for Silicon Graphics Unix workstations. Macintosh VRML 2.0 browsers are expected within the next few months.

Note: VRML 1.0 browsers, such as Netscape’s Live3D 1.0, cannot load and display VRML 2.0 worlds. To view the VRML 2.0 worlds in this column you will need to install a VRML 2.0 browser. See the sidebar on VRML 2.0 Browsers for information on obtaining and installing VRML 2.0 browsers. Also see the VRML Vendors chart for a list of VRML browsers and plug-ins.

Figure 1 shows a few sample worlds to try out. All of the sample worlds include animations only
possible with the advent of VRML 2.0. Click on any of the images to load the associated VRML 2.0 world into your VRML 2.0 browser. Beneath each image is a note giving the size of the world, in bytes, and the expected download time using a 14.4 modem.

(a) Frames that slowly spin when touched, creating an evolving 3-D spiral pattern (10kbytes = 6 seconds)

(b) Floating pads that form geometric patterns as they endlessly slide back and forth (6kbytes = 4 seconds)

(c) Darkened monoliths that glow when touched (33kbytes = 23 seconds)

(d) A dungeon hallway with wooden spikes that slide out when you approach (53kbytes = 37 seconds)

Figure 1. Sample VRML 2.0 worlds you can view with a VRML 2.0 browser. Click on an image to load the world.

All four sample worlds have something to explore. In Figure 1a, click on the colored frames to start them rotating in an evolving 3-D spiral pattern. In Figure 1b, fly through a world of floating pads that slide back and forth in complex geometric patterns. In Figure 1c, click on any gray monolith to start it glowing. In Figure 1d, run the gauntlet in a dungeon hallway, avoiding wooden spikes that shoot out as you approach.

Viewing tip: Once loaded into your VRML 2.0 browser, if these worlds run a little slowly, try reducing the size of the browser window. A smaller window means there’s less screen area for the browser to redraw each time something moves in the world. This reduction in drawing area speeds up the browser and enables it to animate more smoothly, or respond more quickly to user actions.

Note: These sample worlds use advanced features of VRML 2.0 that may not be fully supported yet by some VRML 2.0 browsers. Care has been taken to insure that the worlds load in all VRML 2.0 browsers. Nevertheless, due to different levels of support, the appearance, interactivity, and animation may be somewhat different from
These sample worlds illustrate a few of the animation and interaction features available in VRML 2.0. In this month’s column I’ll focus on the basics of shape building. In the months to come, I’ll return to these examples and explore how you can create animations and interactions, like these, using VRML 2.0.

Using VRML 2.0 files
VRML 2.0 world files contain text instructions that describe how to build 3-D shapes, where to put them, what color to make them, how to animate them, and more. By convention, VRML 2.0 files are named with a ".wrl" file name extension (".wrl" is short for "world"). You can load VRML 2.0 files from your hard disk or off the Web.

Figure 2 shows a simple VRML 2.0 file containing several VRML instructions.

```vrml
#VRML V2.0 utf8
# Build a cylinder shape
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.8 0.8 0.8
    }
  }
  geometry Cylinder {
    height 2.0
    radius 1.0
  }
}
```

![Figure 2. A sample VRML 2.0 file](image)

VRML 2.0 syntax is quite intuitive. Just from reading the words within the VRML file in Figure 2, you can already guess that this file builds a Shape from Appearance and Cylinder descriptions. The Cylinder description uses height and radius values to select the cylinder’s dimensions, and the Appearance description uses Material and diffuseColor values to control shape coloration.

When loaded by a VRML browser, the browser follows the VRML file’s instructions, then builds and displays the virtual world. Using browser menus and buttons, users can move about within the virtual world, view its shapes from any angle they chose, and interact with its animations. If the virtual world contains music and sound effects instructions, the browser plays the sounds, varying their volume and panning to create a 3-D sound experience to match the visuals.

Figure 3 shows three images generated by loading the cylindrical shape world whose VRML 2.0 instructions are shown in Figure 2. Figure 3a shows the world loaded into Netscape Navigator 3.0 using Intervista’s WorldView VRML 2.0 browser plug-in on a PC. Figure 3b shows the same world loaded into Netscape Navigator 3.0 using Silicon Graphics’ Cosmo Player VRML 2.0 browser plug-in on a PC. Figure 3c shows the world loaded into Sony’s Community Place VRML 2.0 browser helper-application for Netscape Navigator 3.0. Sony also provides a Netscape Navigator 3.0 plug-in with a similar user interface. (See the sidebar on VRML 2.0 Browsers for information on obtaining and installing VRML 2.0 browsers. Also see the VRML Vendors chart for a list of VRML browsers and plug-ins.)
Understanding VRML 2.0 syntax

VRML 2.0 files contain these main syntactic elements:

- The VRML 2.0 file header
- Comments
- Nodes
- Fields and Field Values

The VRML 2.0 file header

The first line of every VRML 2.0 file must be the *VRML 2.0 file header*. VRML browsers are case-sensitive, so the header must use upper- and lower-case characters exactly as shown in the following syntax box.

```
Syntax: VRML 2.0 File Header

#VRML V2.0 utf8
```

The VRML file header is a single line indicating that the file is:
A VRML file
- Compliant with version 2.0 of the VRML specification
- A file using the international UTF-8 character set (see the sidebar The UTF-8 character set)

Comments
A comment is an arbitrary note, copyright message, or other type of extra information included in a VRML file. Comments begin with a number-sign (#) and end with a line break. VRML browsers skip past comments wherever they occur in a VRML 2.0 file.

Nodes
Nodes are the basic building-blocks of VRML 2.0 world-building instructions. A VRML 2.0 file always has at least one node in it, and often contains hundreds or even thousands of nodes. Individual nodes build shapes, control shape appearance, describe shape geometry, and so on.

Each node in a VRML file contains:
- The name of a type of node
- An opening curly-brace
- Zero or more fields and field values
- A closing curly-brace

A node’s type name indicates the kind of information contained within the node. VRML 2.0 supports over 50 built-in node types, plus the ability to define new node types. Some browser vendors provide additional extension node types for added functionality. Typical node types include Shape for building a shape, Appearance for describing the appearance of a shape, Cylinder for describing the geometry of a shape, and so on.

VRML browsers are case-sensitive, so shape and SHAPE are not the same as Shape. By convention, all built-in node types in VRML 2.0 use an upper-case character at the beginning of each word in the type name. For instance, Shape, ElevationGrid, and IndexedFaceSet are all built-in node types in VRML 2.0. Authors of new node types and browser vendor extensions should follow the same naming convention.

Indentation and curly-brace style is up to you. VRML 2.0 browsers ignore spaces, tabs, carriage-returns, line-feeds, and commas.

Figure 4 shows a VRML file with the nodes highlighted. Each node type name is followed by an open curly-brace, zero or more fields and their values, and then a matching closing curly-brace.

```
#VRML V2.0 utf8
# Build a cylinder shape
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.8 0.8 0.8
    }
  }
  geometry Cylinder {
    height 2.0
    radius 1.0
  }
}
```

Figure 4. A VRML 2.0 file with the nodes highlighted

Fields and field values
Fields and their field values provide parameters for a node. A node’s curly-braces group together the field information associated with the node.

Each field in a node has a name followed by one or more values. Typical values include floating-point numbers and text strings. Some fields even use nodes as field values.

Different node types have different fields available. The Cylinder node type, for instance, has radius and height fields, while the FontStyle node type has family, style, and size fields.

When a node type has multiple fields, you can provide them in any order within the curly-braces of a node. If you give the same field more than once within the same node, then the last one overrides any given earlier. If you omit a field, the node uses a default value for the field.

Figure 5 shows a VRML file with the fields highlighted. The Shape node has appearance and geometry fields. The Appearance node has a material field, and the Material node has a diffuseColor field. The Cylinder node has height and radius fields. Each field’s name is always followed by a field value.

```
#VRML V2.0 utf8
# Build a cylinder shape
Shape { 
  appearance Appearance { 
    material Material { 
      diffuseColor 0.8 0.8 0.8 
    } 
  } 
  geometry Cylinder { 
    height 2.0 
    radius 1.0 
  } 
}
```

Figure 5. A VRML 2.0 file with the fields highlighted

Each field expects one or more values of a specific field data type. A field data type describes the kind of value a field expects. The height field of a Cylinder node type, for instance, expects a floating-point number giving the cylinder’s height. The diffuseColor field of a Material node type expects a numeric color description, and so forth.

Each field data type has a name, such as SFColor or MFVec3f. Names starting with "SF" indicate data types that hold a single value, such as a floating-point number or a numeric color description. Names starting with "MF" indicate data types that hold multiple values, such as a 3-D coordinate list. The values for multiple value fields must be enclosed within square-brackets when typed into a VRML 2.0 file.

The table below summarizes the field data types available within VRML 2.0.

<table>
<thead>
<tr>
<th>Field type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFBool</td>
<td>A Boolean TRUE or FALSE value</td>
</tr>
<tr>
<td>SFColor</td>
<td>A color specified by three floating-point values selecting the amount of red, green, and blue to be mixed together to form a</td>
</tr>
</tbody>
</table>
desired color

SFFloat  A floating-point value
MFFloat

SFImage An image described by a series of pixel color values

SFInt32  A 32-bit integer value
MFInt32

SFNode  A VRML node value
MFNode

SFRotation  A rotation specified by four floating-point values selecting a rotation axis and rotation angle
MFRotation

SFString  A text string, surrounded by double-quotes
MFString

SFTime  A time specified as a floating-point value, measured in seconds

SFVec2f  A 2-D vector consisting of a pair of floating-point values
MFVec2f

SFVec3f  A 3-D vector consisting of a triple of floating-point values
MFVec3f

In this column, each time a new node type is introduced, a syntax box will be provided to show a quick summary of a node type’s fields, field default values, and field data types. For example, the following is a syntax box for VRML 2.0’s Cylinder node type.

<table>
<thead>
<tr>
<th>Syntax: Cylinder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylinder {</td>
</tr>
<tr>
<td>radius           1.0 # field SFFloat</td>
</tr>
<tr>
<td>height           2.0 # field SFFloat</td>
</tr>
<tr>
<td>bottom           TRUE # field SFBool</td>
</tr>
<tr>
<td>top              TRUE # field SFBool</td>
</tr>
<tr>
<td>side             TRUE # field SFBool</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

The Cylinder node type’s syntax box indicates that the node type has five fields: radius, height, bottom, top, and side. Each field has a default value, such as 1.0 for the radius field, and TRUE for the top field.

Each field line in the syntax box also indicates the field’s data type. For instance, the first two fields of the Cylinder node type expect single floating-point values, and the last three expect single Boolean values.

EventIns, eventOuts, fields, and exposed fields

VRML 2.0 provides nodes for building shapes, creating lights, placing sounds, and more. To make a virtual world come alive, you can connect nodes together, wiring them into an animation circuit. Each connected node in the circuit acts like an electronic component with its own input and output connection points. By wiring the output of one node into the input of another, you can establish a route along which can flow data values, or events.
For example, to make a light blink you can wire the on/off switch input of a lighting node to a node that outputs on/off events. Each time an "on" event flows along the route to the light, the light turns on. Each time an "off" event flows along the route, the light turns off. You can construct similar circuits to make shapes move, rotate, change color, and so on.

An eventIn is an input connection point for a node. An eventOut is an output connection point. Like fields, eventIns and eventOuts have names and data types. Different node types have different eventIns and eventOuts available. The SpotLight node type, for instance, has a set_on eventIn for turning the light on and off. The PositionInterpolator node type has a value_changed eventOut that outputs positions you can use to animation the position of a shape.

An exposed field is a special type of field that combines together a standard field, an eventIn to set that field, and an eventOut that outputs the field value each time the field is set. The on exposed field of a SpotLight node type, for example, has an implicit set_on eventIn and an implicit on_changed eventOut.

Animation circuits, exposed fields, eventIns, and eventOuts will be discussed in greater depth in future columns. To enable this month’s column to be used later as a syntax reference, the syntax box for each node type discussed below indicates fields, exposed fields, eventIns, and eventOuts.

---

**Giving shape dimensions**

Many node types include fields for setting the dimensions of a shape. The Cylinder node type, for instance, has height and radius fields to specify the height of the cylinder, and its radius. By convention, these dimensions should be given in meters. The default values for the Cylinder node type, for instance, create a cylinder 2.0 meters tall with a 1.0 meter radius.

For some worlds, using meters is awkward or inappropriate. A world depicting a model of a molecule, for instance, may measure dimensions in Angstroms instead of meters. A world depicting a spiral galaxy may use dimensions measured in lightyears. Because of these special needs of some worlds, VRML 2.0 does not impose any required unit of measure for shape dimensions. The interpretation of dimension numbers is largely up to you.

In this column, all shape dimensions are expressed generically in terms of units. So, instead of saying a cylinder is 2.0 meters high, I’ll say it is 2.0 units high and let you and your VRML 2.0 application decide if units are meters, Angstroms, lightyears, or whatever.

---

**Building shapes**

Most VRML 2.0 files build one or more shapes. Each VRML 2.0 shape is described by specifying the shape’s geometry and appearance. Shape geometry attributes provide the form, or structure of the shape. Shape appearance attributes provide the coloring of the shape. A car tire shape, for instance, has a cylindrical geometry and a black appearance. A planet shape has a spherical geometry and a multi-colored cloudy appearance.

**The Shape node type**

All shapes are built using the Shape node type. Each Shape node combines together your choices for geometry and appearance via the node’s geometry and appearance fields.
Syntax: Shape

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

The value of the `geometry` field specifies a node that defines the 3-D form, or geometry, of the shape. Typical `geometry` field values include primitive geometry `Box`, `Cone`, `Cylinder`, and `Sphere` node types discussed below. The default `NULL` value for this field indicates the absence of shape geometry.

The value of the `appearance` field specifies a node defining the coloration of the shape. Typical `appearance` field values include the `Appearance` node type discussed below. The default `NULL` value for this field indicates a black appearance.

Note: Currently there is wide variability in how VRML 2.0 browsers treat the `NULL` value case for the `appearance` field. Some browsers draw null-appearance shapes in black, while others draw such shapes in flat white, or in shaded white. To guarantee consistent treatment in all browsers, you should always provide an `Appearance` node value for the `appearance` field, thereby avoiding null-appearance issues.

Notice that the `geometry` and `appearance` fields both use entire nodes as their values. Those nodes may, in turn, use further nodes as field values, and so on. This node within a node structure of VRML 2.0 helps to group features together in logical building-blocks, making the syntax easier to learn and use. For example, Figure 6 shows a `Shape` node using an `Appearance` node to specify the shape coloration, and a `Box` node to create a 3-D rectangular box shape geometry.

```
#VRML V2.0 utf8
Shape {
    appearance Appearance {
        material Material {
            diffuseColor 0.8 0.8 0.8
        }
    }
    geometry Box {
        size 2.0 2.0 2.0
    }
}
```

Figure 6. A shape built using a Shape node.

Specifying shape geometry

VRML 2.0 provides several `geometry` node types that you can use with a `Shape` node’s `geometry` field to specify the form and structure of a shape. Geometry node types include the `Box`, `Cone`, `Cylinder`, and `Sphere` node types, as well as more advanced geometry node types. Each geometry node type has one or more fields that enable you to specify geometry dimensions, such as the height of a cylinder, or the radius of a sphere.

The Box node type

The `Box` geometry node type creates a 3-D rectangular box when used as the value of the `geometry` field in a `Shape` node.
The value of the `Box` node type’s `size` field specifies the dimensions of the box. The first value in the field is the box’s width, the second its height, and the third its depth. All three dimensions must be greater than 0.0. The default `size` field values build a box 2.0 units wide, tall, and deep.

A `Box` node can be used as a value for the `geometry` field of a `Shape` node, like that shown in Figure 7.

```vrml
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.8 0.8 0.8
    }
  }
  geometry Box {
    size 2.0 2.0 2.0
  }
}
```

**Figure 7.** A box shape built using Shape and Box nodes. Click on the image to load the world.

The `Cone` node type

The `Cone` geometry node type creates a 3-D upright cone when used as the value of the `geometry` field in a `Shape` node.

```vrml
Syntax: Cone
Cone {
  height 2.0 # field SFFloat
  bottomRadius 1.0 # field SFFloat
  side TRUE # field SFBool
  bottom TRUE # field SFBool
}
```

The values of the `Cone` node type’s `height` and `bottomRadius` fields specify the height of a cone, and the radius of its bottom. Both values must be greater than 0.0. The default values create a cone with a height of 2.0 units, and a bottom radius of 1.0 unit.

The values of the `side` and `bottom` fields specify whether or not the sloping sides and bottom of the cone are built. If a field value is `TRUE`, the corresponding part of the cone is built. If a field value is `FALSE`, the corresponding cone part is not built. The default value for both fields is `TRUE`.

A `Cone` node can be used as a value for the `geometry` field of a `Shape` node, like that shown in Figure 8.
The Cylinder node type

The Cylinder geometry node type creates a 3-D upright cylinder when used as the value of the geometry field in a Shape node.

Syntax: Cylinder

<table>
<thead>
<tr>
<th>Cylinder {</th>
<th># field</th>
<th>SFFloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>bottom</td>
<td>TRUE</td>
<td>SFFloat</td>
</tr>
<tr>
<td>top</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>side</td>
<td>TRUE</td>
<td></td>
</tr>
</tbody>
</table>

The values of the Cylinder node type’s height and radius fields specify the height and radius of a cylinder. Both values must be greater than 0.0. The default values create a cylinder with a height of 2.0 units, and a radius of 1.0 unit.

The values of the side, bottom, and top fields specify whether or not the curved sides, bottom, and top of the cylinder are built. If a field value is TRUE, the corresponding part of the cylinder is built. If a field value is FALSE, the corresponding cylinder part is not built. The default value for all three fields is TRUE.

A Cylinder node can be used as a value for the geometry field of a Shape node, like that shown in Figure 9.
The Sphere node type
The Sphere geometry node type creates a 3-D sphere, or ball, when used as the value of the geometry field in a Shape node.

Syntax: Sphere

```
Sphere {
  radius           1.0          # field        SFFloat
}
```

The value of the Sphere node type’s radius field specifies the radius of a sphere. The radius field value must be greater than 0.0. The default value builds a sphere with a radius of 1.0 unit.

A Sphere node can be used as a value for the geometry field of a Shape node, like that shown in Figure 10.

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.8 0.8 0.8
    }
  }
  geometry Sphere {
    radius 1.0
  }
}
```

Figure 10. A sphere shape built using Shape and Sphere nodes.

Click on the image to load the world.

Specifying shape appearance
The Shape node type’s appearance field is used to specify the appearance of a shape. VRML 2.0’s rich set of appearance controls enable you to select shape color, glow color, material finish, and transparency levels. With VRML 2.0’s texture mapping features, you can place an image from an image file onto the sides of a shape, like sticking a decal on a model airplane. Using advanced appearance controls, you can color individual shape parts and create color gradients across the sides of a shape.

In this month’s column, I’ll take a first look at two of the node types available for controlling shape appearance: Appearance and Material.

The Appearance node type
The Appearance node type specifies the appearance attributes of a shape, and may be used as the value of the appearance field of a Shape node.

Syntax: Appearance

```
Appearance {
  material         NULL         # exposedField SFNode
  texture          NULL         # exposedField SFNode
  textureTransform NULL         # exposedField SFNode
}
```

The value of the Appearance node type’s material field specifies a node that defines the material
coloration and finish attributes of the appearance. Typical `material` field values include the `Material` node type. The default `NULL` value for this field indicates a black material.

The `texture` and `textureTransform` fields enable you to paste a texture image on to the sides of a shape. These features are discussed in a future column.

*Note:* As with the `NULL` value case for the `appearance` field of a `Shape` node, there is also wide variability in how VRML 2.0 browsers treat the `NULL` value case for the `material` field of an `Appearance` node. Some browsers draw null-material shapes in black, while others draw such shapes in flat white, or in shaded white. To guarantee consistent treatment in all browsers, you should always provide a `Material` node value for the `material` field, thereby avoiding null-material issues.

### The Material node type

The `Material` node type specifies the material color attributes of a shape appearance, and may be used as the value of the `material` field of an `Appearance` node.

```
Syntax: Material
Material {
  diffuseColor     0.8 0.8 0.8  # exposedField SFColor
  ambientIntensity 0.2          # exposedField SFFloat
  emissiveColor    0.0 0.0 0.0  # exposedField SFColor
  shininess        0.2          # exposedField SFFloat
  specularColor    0.0 0.0 0.0  # exposedField SFColor
  transparency     0.0          # exposedField SFFloat
}
```

The value of the `Material` node type’s `diffuseColor` field specifies a color for the material. A material color is specified using three floating-point values between 0.0 and 1.0 that indicate the amount of red, green, and blue light to be mixed together to form a color. A value of 0.0 for a red, green, or blue amount means that color is turned off. A value of 1.0 for a red, green, or blue amount means that color is turned on completely. Values between 0.0 and 1.0 mean a color is partially turned on. The default value for this field is a medium-bright white created by mixing together 0.8 of red light, 0.8 of green light, and 0.8 of blue light.

Colors created by mixing together red, green, and blue light are called **RGB colors** (“RGB” comes from the first letters of the three color components). The table below provides a brief list of a few RGB colors and their corresponding red, green, and blue values used in a `diffuseColor` field.

<table>
<thead>
<tr>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Pure red</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>Pure green</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>Pure blue</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>White</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Black</td>
</tr>
</tbody>
</table>
1.0  1.0  0.0  Yellow
0.0  1.0  1.0  Cyan
1.0  0.0  1.0  Magenta
0.75 0.75 0.75  Light gray
0.5   0.5   0.5   Medium gray
0.25 0.25 0.25  Dark gray
0.5   0.0   0.0   Dark red
0.0   0.5   0.0   Dark green
0.0   0.0   0.5   Dark blue

Note: Whereas VRML 2.0 uses red, green, and blue color values between 0.0 (off) and 1.0 (on), many drawing and painting applications instead use red, green, and blue values between 0 (off) and 255 (on). These two value ranges are equivalent. You can convert from a 0-255 RGB color to a VRML 2.0 RGB color by dividing each red, green, and blue value by 255.0.

To give shapes a 3-D look, the VRML browser automatically computes darker colors as it shades the sides of a shape, gradually darkening the shading color as it progresses from the lighted side of a shape to the unlighted sides.

The remaining fields of the Material node type enable you to control the emissive (glowing) color, vary its transparency, and specify its material finish. These features are discussed in a future column.

The Appearance and Material node types are always used together with a Shape node. An Appearance node is used as the value of the Shape node’s appearance field, and a Material node is used as the value of the Appearance node’s material field. Figure 11 shows a sample use of these node types.

```
#VRML V2.0 utf8
Shape { 
  appearance Appearance { 
    material Material { 
      diffuseColor 1.0 0.0 0.0 
    } 
  } 
  geometry Sphere { 
    radius 1.0 
  } 
}
```

![Figure 11. A red sphere shape with appearance controlled by Appearance and Material nodes.](image)

Click on the image to load the world.

**Experimenting with VRML 2.0**
With VRML 2.0’s shape building tools in hand, it’s time to experiment! Each of the VRML examples in Figures 7 through 11 used a single Shape node to build a single shape in a virtual world. Figure 11, for instance, built a single red sphere. To make more interesting worlds, you can combine together multiple Shape nodes within the same VRML 2.0 file.
Figure 12 builds a multi-colored 3-D plus-sign by building three box shapes within the same VRML 2.0 file. The first box is 8.0 units wide, 0.5 units tall and deep, and shaded with a dark blue appearance. The second box is 8.0 units tall, 0.5 units wide and deep, and shaded with a cyan appearance. The third box is 8.0 units deep, 0.5 units tall and wide, and shaded with a purple appearance.

By default, all shapes are built at the center of the world. If you build multiple shapes at the same location, then they overlap and intersect each other. You can use this feature of VRML 2.0 to create complex 3-D shapes by using multiple overlapping shapes.

Figure 13 builds a stair-stepped tetrahedron out of a series of overlapping box shapes, all placed at
the center of the world. The first box is wide, flat, and yellow. Successive boxes decrease in width, increase in height, and turn more red. The last box is narrow, tall, and red.
You can build complex shapes by using a variety of geometry node types. For example, Figure 14 creates a "space-probe" using a series of shapes of varying sizes and colors.
Figure 14. A "space-probe" made from multiple overlapping shapes.

Click on the image to load the world.

Next in the VRML Technique column
Next month I’ll continue discussing shape building and introduce VRML 2.0 features for positioning, orienting, and scaling shapes using the **Transform** node type. I’ll also continue discussing the **Material** node type and discuss how you can make shapes semi-transparent or make them appear to glow.
make them appear to glow.

---

### Resources

- A list of David Nadeau’s VRML Technique columns in *NetscapeWorld*
- VRML 2.0 browsers *NetscapeWorld*’s guide to finding and installing a VRML browser on your computer.
- VRML 2.0 glossary
- *NetscapeWorld*’s VRML vendors chart A handy reference of VRML browser and server companies including their plug-ins to Web browsers -- with updated items in bold to aid your review -- and links to all the vendors.
- The UTF-8 character set sidebar accompanying the first VRML Technique column.

### Specifications

- VRML 2.0 specification [http://vag.vrml.org/VRML2.0/FINAL/](http://vag.vrml.org/VRML2.0/FINAL/)
- ISO 10646-1:1993 Universal Character Set (UCS) specification sales information
  - [http://www.iso.ch/cate/d18741.html](http://www.iso.ch/cate/d18741.html)
- UTF-8 character encoding scheme for UCS

### Sites

- The VRML Repository [http://www.sdsc.edu/vrml](http://www.sdsc.edu/vrml)
- VRML Architecture Group [http://vag.vrml.org](http://vag.vrml.org)

---

### About the author

David R. Nadeau is a co-author of *The VRML 2.0 Sourcebook*, published by John Wiley & Sons and written with Andrea L. Ames and John L. Moreland. David is a staff researcher at the San Diego Supercomputer Center where he is a specialist in 3-D computer graphics, virtual reality, and scientific visualization. He is also the creator of The VRML Repository, a Web site providing extensive information on VRML software, documentation, and 3-D worlds.

⚠️ You can buy David R. Nadeau’s *The VRML 2.0 Sourcebook* at a 20% discount from Amazon.com Books.

---

Feedback: editors@netscapeworld.com

URL:
[http://www.netscapeworld.com/netscapeworld/nw-12-1996/nw-12-vrmltechnique.htm](http://www.netscapeworld.com/netscapeworld/nw-12-1996/nw-12-vrmltechnique.htm)

Last updated: Tuesday, March 11, 1997
The Transform node type to position, orient, and resize shapes to build virtual structures. Along the way, I discuss VRML 2.0’s DEF and USE features that enable you to use the same shape repeatedly to build structural patterns, such as a row of identical marble columns or a grid of windows on a skyscraper. (4,700 words)

Table of contents

Building virtual structures
Repeating nodes using defined names
  • Defining and using node names
  • The syntax of DEF and USE
  • Experimenting with DEF and USE
Building shapes in three dimensions
  • Using the right-hand rule for axis directions
  • Determining 3-D coordinates
Using the world’s default coordinate system
Creating and positioning coordinate systems
  • Parent and child coordinate systems
  • The Transform node type
  • Experimenting with coordinate system translation
Orienting coordinate systems
  • Specifying a rotation axis
  • Table: Common rotation axes
  • Specifying a rotation angle
  • Table: Common rotation angles
  • Using the right-hand rule for rotations
  • The Transform node type, revisited
  • Experimenting with coordinate system rotation
Scaling coordinate systems
  • The Transform node type, revisited again
  • Experimenting with coordinate system scaling

Next in the VRML Technique column
Resources
About the author

Building virtual structures
Last month’s column introduced VRML 2.0’s shape building features, including the Shape node
type, and four of VRML’s geometry node types: **Box, Cone, Cylinder**, and **Sphere**. Using these node types, you can create 3-D shapes and view them interactively within a VRML 2.0 browser.

(See *NetscapeWorld’s* sidebar on VRML 2.0 Browsers for information on obtaining and installing VRML 2.0 browsers. Also see the *NetscapeWorld* VRML Vendors chart for a list of VRML browsers and plug-ins, and our glossary of VRML 2.0 terms. You need a VRML browser or plug-in to view the 3D examples presented in this series.)

By combining multiple shapes, you can create complex virtual structures. Figure 1 shows two sample structures built by positioning, orienting, and resizing multiple shapes. Click on an image to load the associated VRML 2.0 world into your VRML 2.0 browser. The captions below each figure give the size of the world in bytes, and the expected download time.

![Sample VRML 2.0 structures built using boxes, cones, cylinders, and spheres.](image)

(a) A fairy-tale castle built from cylinders and cones  
(15 kilobytes = 10 seconds @ 14.4bps)

(b) A structure with a complex ceiling built from boxes  
(11 kilobytes = 8 seconds @ 14.4bps)

**Figure 1. Sample VRML 2.0 structures built using boxes, cones, cylinders, and spheres.**  
*Click on an image to load the world.*

*Viewing tip:* Once loaded into your VRML 2.0 browser, if these worlds run a little slowly, try reducing the size of the browser window. A smaller window means there is less screen area for the browser to redraw each time you move in the world. This reduction in drawing area speeds up the browser and enables it to animate more smoothly, or respond more quickly to user actions.

---

**Repeating nodes using defined names**

Last month’s column ("Building Shapes") discussed key syntactic features of VRML 2.0, including the VRML file header, comments, nodes, fields, and field values. A VRML 2.0 file, for instance, always starts with a VRML file header, followed by one or more nodes. Each node has a node type name, such as **Shape**, followed by an opening curly-brace, zero or more fields and values, and a closing curly-brace. The fields within the curly-braces provide values for named attributes of a node. The **Shape** node type, for instance, has fields to describe the appearance and geometry of a 3-D shape. The **Box** node type has a field to describe the width, height, and depth of a 3-D box.

The VRML world shown in Figure 2 builds a 3-D "plus-sign" from three red box shapes. Each **Shape** node builds a box using a **Box** node. The **Appearance** and **Material** nodes specify a red appearance for the boxes.
In Figure 2, notice that all three box shapes use identical Appearance and Material nodes to achieve a uniform red appearance. Repeatedly specifying the same set of nodes and values is tedious. Such repetition also makes it awkward to make global changes to a set of shapes, such as to turn all three red boxes blue.

**Defining and using node names**
To reduce redundancy, you can define a name for any node in a VRML 2.0 file. Once a node has a name, you can re-use that same node later in the file without retyping the node’s description.

For example, you can give the name Red to the Appearance node used to describe the red color of the first box in Figure 2. Then, you can re-use the same Red node twice more to shade the remaining two boxes the same shade of red.

A node with a defined name is called an original node, and each re-use of that node is called an instance. You can have any number of named original nodes in a VRML world, and each original can be instanced any number of times.

Fields and their values are only specified when creating an original node. Each instance re-uses the original’s field values without change. This enables you to define once the node that makes up,
say, a red appearance, then later create an instance of that red appearance node each time you need
 to make another shape red. Later, if you make a change to the original red appearance node, all of
 the instances of the red appearance change as well. This feature enables you to make rapid
 changes throughout your world by only modifying the original nodes.

The syntax of DEF and USE
To define a node for use in instancing, precede the node type name with the word "DEF" and a
 node name of your choosing. While a VRML 2.0 file may contain any number of named nodes, no
two may have the same name.

**Syntax: DEF**

```
DEF node_name node_type { . . . }
```

For example, you can give the name *Red* to an *Appearance* node like this:

```
. . .
Shape {
  appearance DEF Red Appearance {
    material Material {
      diffuseColor 1.0 0.0 0.0
    }
    geometry Box { size 8.0 0.5 0.5 }
  }
. . .
```

Once you have defined a name for a node, you can re-use that node again and again within the
 same file by typing only the word "USE" and the node name. There is no need to re-specify the
 node type, curly-braces, fields, or field values: The VRML browser automatically fills these in
 from the original node.

**Syntax: USE**

```
USE node_name
```

For example, you can use a named node *Red* in place of an *Appearance* node like this:

```
. . .
Shape {
  appearance USE Red
  geometry Box { size 0.5 8.0 0.5 }
} . . .
```

Node names may be any convenient sequence of characters, and are case-sensitive. For example,
 "RED" and "red" are different names. Node names may include letters, numbers, and underscores.
The following are examples of legal node names:

```
Red  WallColor3  my_chair
PianoKey  GurgleSound  Kitchen_Design
NCC1701  BrightLight  MarbleTexture
```

Node names cannot start with a number and cannot include non-printing ASCII characters, like
spaces, tabs, line-feeds, form-feeds, and carriage-returns. Names also cannot include double or single quotes, number signs, plus signs, minus signs, commas, periods, square brackets, backslashes, or curly braces. The following names are prohibited, since they are words used for other, specific purposes within VRML 2.0:

DEF  EXTERNPROTO  FALSE
IS    NULL         PROTO
ROUTE TO           TRUE
USE   eventIn      eventOut
exposedField  field

**Experimenting with DEF and USE**

You can use DEF and USE to simplify the red box world shown in Figure 2. Start by adding DEF Red to define a name for the Appearance node for the first box shape. For each of the remaining two box shapes, substitute the full Appearance node specification with USE Red. Figure 3 shows the simplified world that results.

![Image of a red box world](image)

```#VRML V2.0 utf8
Shape { 
  appearance DEF Red Appearance { 
    material Material { 
      diffuseColor 1.0 0.0 0.0 
    } 
    geometry Box { size 8.0 0.5 0.5 } 
  } 
} 
Shape { 
  appearance USE Red 
  geometry Box { size 0.5 8.0 0.5 } 
} 
Shape { 
  appearance USE Red 
  geometry Box { size 0.5 0.5 8.0 } 
} 
```

**Figure 3. An Appearance node, with a defined name, used twice more later in the file.**

Click on the image to load the world.

You can name and instance any node in a VRML 2.0 file, including Appearance, Material, Shape, and geometry nodes. Node instancing eliminates redundancy and decreases the size of a VRML file. Additionally, node instancing increases the performance of a VRML browser by enabling it to share node descriptions and the processing associated with them.

**Building shapes in three dimensions**
Like the real world, 3-D shapes built in a VRML 2.0 virtual world have width, height, and depth. To provide reference directions along which to measure these shape dimensions, you can imagine a trio of arrows drawn through the middle of a shape extending left-to-right, bottom-to-top, and back-to-front. Figure 4 shows such a set of arrows drawn through a box shape.

![Figure 4. A box shape with reference arrows for width, height, and depth measurement.](image)

A direction arrow, like any of those shown in Figure 4, is called an axis. Conventionally, the left-to-right arrow, shown in red, is called the X axis, the bottom-to-top arrow, shown in green, is called the Y axis, and the back-to-front arrow, shown in blue, is called the Z axis. Using these axes, you can measure the width of a shape along the X axis, the height along the Y axis, and the depth along the Z axis.

The center point where all three axes cross is called the origin. VRML 2.0 browsers build all shapes so that their centers are at the origin.

You can treat each axis like a ruler along which to measure positive and negative distances from the origin. A positive distance extends in the direction of the axis arrow, while a negative distance extends in the reverse direction.

**Using the right-hand rule for axis directions**

While building complex worlds, it can be surprisingly hard to remember which axis points in which direction. To help keep the axis positive directions straight, you can use the right-hand rule. For this rule, hold up your right hand, stick your thumb out as if hitch-hiking, point your index finger straight up, and point your second finger straight forward. Curl your other fingers under. In this configuration, your thumb points in the positive X direction, your index finger in the positive Y direction, and your second finger in the positive Z direction. Figure 5 shows this hand configuration.
Determining 3-D Coordinates
A triple of X, Y, and Z distance measurements uniquely describe a location in 3-D. Such a triple of distances is called a 3-D coordinate. You can use 3-D coordinates to describe the location of key features of a shape, such as the corners of a box. Figure 6 and the table below, for example, show the 3-D coordinates for the eight corners of a box with a width, height, and depth of 2.0 units.

3-D coordinates for a box shape

<table>
<thead>
<tr>
<th>Corner</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front, top, right</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Front, top, left</td>
<td>-1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Front, bottom, right</td>
<td>1.0</td>
<td>-1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Building 3-D shapes in VRML 2.0 is like playing a child’s game of connect-the-dots. A geometry node, such as a **Box** node, places 3-D coordinates and connects them together to form the faces of a shape. The four geometry node types discussed in last month’s column, **Box**, **Cone**, **Cylinder**, and **Sphere**, each automatically computes a set of 3-D coordinates and connecting faces.

---

**Using the world’s default coordinate system**

A set of reference X, Y, and Z axes define a coordinate system in which you can measure distances to locate 3-D coordinates. Shapes whose coordinates are measured with respect to a particular coordinate system are said to be in that coordinate system.

Every VRML 2.0 world file has a default coordinate system in which to build the world’s shapes. That coordinate system is called, simply, the *world coordinate system*. By default, all shapes are built centered at the origin of this world coordinate system. For example, all three red boxes in Figures 2 and 3 are built centered in the world coordinate system by default.

---

**Creating and positioning coordinate systems**

VRML 2.0 enables you to position shapes through the creation of new coordinate systems using the **Transform** node type. The origin of a new coordinate system is positioned by you at a 3-D coordinate measured relative to the origin of another coordinate system, such as the world coordinate system. Any shape you build in the new coordinate system is centered at the new coordinate system origin instead of at the world origin.

Using new coordinate systems, you can build and position shapes anywhere in the world. For example, to position a shape 4.0 units to the right of the world origin, create a new coordinate system 4.0 units to the right, then build the shape centered within that new coordinate system. Later, if you want the shape 6.0 units to the right instead, change the position of the shape’s coordinate system. When the coordinate system moves, the shape built within it moves as well.

*Tip:* You can think of a coordinate system as an imaginary box that can contain shapes. If you move a coordinate system box about, then the shapes within the box move as well.

Figure 7 shows a series of images illustrating steps in creating a row of marble columns. Figure 7a starts the series with a set of axes indicating the world coordinate system. Figure 7b shows a marble column built at the world coordinate system origin. Figure 7c shows a new coordinate system positioned to the right relative to the world coordinate system, and Figure 7d shows a column centered in the new coordinate system. Figures 7e, 7f, 7g, and 7h repeat the process, showing two more coordinate systems and a column built in each one.
(a) The world coordinate system alone, indicated by a set of axes.

(b) A marble column built at the center of the world coordinate system.

(c) A second coordinate system positioned to the right relative to the world coordinate system origin.

(d) A marble column built at the center of the second coordinate system.

(e) A third coordinate system positioned behind the world coordinate system origin.

(f) A marble column built at the center of the third coordinate system.

(g) A fourth coordinate system positioned to the right and behind the world coordinate system origin.

(h) A marble column built at the center of the fourth coordinate system.
Parent and child coordinate systems
When a shape is built within a particular coordinate system, we say that the shape is a child of that coordinate system. The coordinate system enclosing the children is a parent coordinate system. Each of the marble columns above, for instance, are children of their respective parent coordinate systems.

Similarly, when a new coordinate system is positioned relative to another, we say that the new coordinate system is a child coordinate system. Each of the marble column coordinate systems used in Figures 7c through Figure 7h are positioned relative to the world coordinate system, and are therefore children of that coordinate system.

Parent coordinate systems can, in turn, be children of other parent coordinate systems, and so on. This parent-child relationship of coordinate systems and shapes creates a family tree of world scenery. The top-most parent of the family tree is the VRML file’s world coordinate system. The entire family tree is called a scene graph.

Figure 8 shows a diagram of the scene graph for the columns in Figure 7. The first column is a child of the world coordinate system at the top of the family tree. The second, third, and fourth columns are children of three new coordinate systems that are, themselves, children of the world coordinate system.

Typical VRML 2.0 worlds contain dozens or even thousands of coordinate systems in the scene graph. It is common, for instance, to use a new coordinate system for each shape in a world. If the world individually positions thousands of shapes, then there will be thousands of coordinate systems as well.

The Transform node type
The Transform node type creates a new coordinate system relative to its parent coordinate system. Shapes created as children of a Transform node are built relative to that new coordinate system’s origin.
The values in the children field provide child nodes to be built within the new coordinate system and centered at its origin. The list of children is enclosed within square-brackets. The default value for this field is an empty list of children.

Typical children field values include Shape and Transform nodes. A Transform node may be the child of another Transform node, which may be a child again, and so on up the family tree of coordinate systems.

The values of the translation field specify the positive or negative distances in the X, Y, and Z directions between the parent’s coordinate system origin and the origin of the new coordinate system. The default zero values for this field cause no translation in X, Y, or Z. This places the new coordinate system in exactly the same place as the parent coordinate system.

**Experimenting with coordinate system translation**

Figure 9 shows a castle tower built using a white cylinder and a red cone. The cylinder is built centered within the world coordinate system. The cone is built centered within a new coordinate system positioned 25.0 units up the Y axis. In the VRML text, notice that the cone shape is described within the children list of the Transform node. That child list, like all VRML 2.0 value lists, is enclosed within square-brackets.
# Tower
Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Cylinder {
    radius 5.0
    height 30.0
    top FALSE
  }
}

# Roof
Transform {
  translation 0.0 25.0 0.0
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1.0 0.0 0.0
        }
      }
      geometry Cone {
        bottomRadius 8.0
        height 20.0
      }
    }
  ]
}

Figure 9. A castle tower built using a cylinder in the world coordinate system and a cone in a new translated coordinate system.

Click on the image to load the world.

You can use any number of Transform nodes within the same VRML 2.0 file and provide any number of child shapes and coordinate systems within the children field. Figure 10, for example, uses multiple Transform nodes to build a castle using four towers and a central box. Notice that shapes can overlap, even when built within separate coordinate systems. Also notice the use of DEF and USE to share appearance and shape definitions.
Orienting coordinate systems

VRML 2.0’s Transform node type enables you to create and position a new coordinate system anywhere in your world. Imagine, for instance, that you create an airplane shape within the new coordinate system. Using the translation field of the coordinate system’s Transform node, you can "fly" the airplane about.

To make the airplane’s flight more realistic, you can control the airplane’s orientation by rotating the airplane’s coordinate system. You can tilt the airplane upward during take-off, turn it to face the direction in which it is flying, or bank it in a turn.

Specifying a rotation axis

Coordinate system rotation is described by a rotation axis and a rotation angle specified in the rotation field of a Transform node. The rotation axis defines an imaginary line about which to rotate the coordinate system. The rotation angle indicates the amount to rotate about the axis.

A rotation axis can point in any direction. For example, the rotation axis for a toy top is vertical, while that for a car wheel is horizontal. To specify a rotation axis, imagine drawing a line between two 3-D coordinates. One coordinate is always the origin of a new coordinate system. The second coordinate’s X, Y, and Z values are specified in the Transform node’s rotation field. The imaginary line between these two coordinates defines a rotation axis.

For example, to define a vertical rotation axis for a toy top, use a rotation axis that points straight up from the origin. The second coordinate for such an axis is directly above the origin, such as (0.0, 1.0, 0.0).

The distance between the origin and the second coordinate does not matter. Any point on the imaginary line is valid. To define a rotation axis that points straight up along the Y axis, (0.0, 2.0, 0.0), (0.0, 0.589, 0.0), (0.0, 1823789.0, 0.0), and (0.0, 1.0, 0.0) are all equivalent because they all point straight up.

Note: Technically, the rotation axis is a vector whose direction orients the rotation. The magnitude of the vector is ignored.

While you can specify a rotation axis in any direction, in practice most rotation axes aim to the right along the X axis, up along the Y axis, or out along the Z axis. The table below provides the
rotation axis values used to create these common axes.

**Common rotation axes**

<table>
<thead>
<tr>
<th>Direction</th>
<th>Rotation axis values</th>
</tr>
</thead>
<tbody>
<tr>
<td>To the right along the X axis</td>
<td>1.0 0.0 0.0</td>
</tr>
<tr>
<td>Up along the Y axis</td>
<td>0.0 1.0 0.0</td>
</tr>
<tr>
<td>Out along the Z axis</td>
<td>0.0 0.0 1.0</td>
</tr>
</tbody>
</table>

For example, airplane orientation is typically described using three rotations: *pitch*, *yaw*, and *roll*. Pitch rotations tilt an airplane’s nose up or down with rotation about the X axis. Yaw rotations spin an airplane around the Y axis. Roll rotations turn an airplane about the Z axis. Figure 11 shows these three rotation axes for an airplane model.

![Figure 11. Airplane rotation about X, Y, and Z axes.](image)

**Specifying a rotation angle**

Along with a rotation axis, a rotation angle specifies the amount by which to rotate around the chosen axis. Rotation angles may be positive or negative and are measured in *radians*, instead of the more familiar degrees.

Recall that an angle measurement in degrees varies from 0.0 to 360.0 in a full circle. Half way around the circle is 180.0 degrees, and a quarter of the way is 90.0 degrees. An angle measurement in radians varies from 0.0 to \( \pi = 6.283 \) in a full circle. Halfway around the circle is \( \pi = 3.142 \) radians, and a quarter of the way is 0.5 \( \pi = 1.57 \) radians.
You can convert between degrees and radians using these simple formulae:

\[
\begin{align*}
\text{radians} & = \text{degrees} \times \frac{3.142}{180.0} \\
\text{degrees} & = \text{radians} \times \frac{180.0}{3.142}
\end{align*}
\]

The table below shows several common rotation angles in degrees and radians.

### Common rotation angles

<table>
<thead>
<tr>
<th>Degrees</th>
<th>Radians</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>10.0</td>
<td>0.175</td>
</tr>
<tr>
<td>45.0</td>
<td>0.785</td>
</tr>
<tr>
<td>90.0</td>
<td>1.571</td>
</tr>
<tr>
<td>180.0</td>
<td>3.142</td>
</tr>
<tr>
<td>270.0</td>
<td>4.712</td>
</tr>
</tbody>
</table>

**Using the right-hand rule for rotations**

If you look down an axis from the arrow end, a positive rotation angle turns in a *counter-clockwise* direction. When worlds get complex, it can be hard to decide if a positive or negative rotation is needed to get a desired rotation. To help keep rotation directions straight, you can use a variation of the *right-hand rule* introduced earlier. Hold up your right hand, and stick your thumb out as if hitch-hiking. Orient your hand so that your thumb points in the positive direction of the X, Y, or Z axis. Curl your fingers around as if gripping the axis. The circular direction in which your fingers curl is a positive rotation direction around that axis. Figure 12 shows a hand illustrating the positive rotation directions for the X, Y, and Z axes.
The Transform node type, revisited
As before, the Transform node type creates a new coordinate system relative to its parent coordinate system.

Syntax: Transform

```plaintext
Transform {
    translation 0.0 0.0 0.0  # exposedField SFVec3f
    rotation 0.0 0.0 1.0 0.0  # exposedField SFRotation
    . . .
    children [ ]  # exposedField MFNode
}
```

The values of the rotation field provide a rotation axis and rotation angle with which to orient the new coordinate system. The first three values in the field specify the X, Y, and Z values for a rotation axis. The last field value specifies a rotation angle, measured in radians. All field values may be positive or negative. The default field values specify a rotation axis aimed outward along the Z axis with a zero radian rotation angle.

Rotation and translation can be used together to first orient a new coordinate system, then position it relative to a parent coordinate system.

Experimenting with coordinate system rotation
Figure 13 shows an archway built using two vertical columns, a horizontal cross-piece, and two tilted roof blocks. Each shape is built within its own coordinate system and translated into position. Each roof piece is tilted using a rotation field with a Z-axis rotation by 0.524 radians = 30.0 degrees. Notice the use of DEF and USE to share shapes and appearances.
#VRML V2.0 utf8

# Left and right columns
Transform {
    translation -2.0 3.0 0.0
    children [
        DEF Column Shape {
            appearance DEF White Appearance {
                material Material {
                }
            }
            geometry Cylinder {
                radius 0.3
                height 6.0
                top FALSE
            }
        }
    ]
}
Transform {
    translation 2.0 3.0 0.0
    children [ USE Column ]
}

# Cross-piece
Transform {
    translation 0.0 6.05 0.0
    children [
        Shape {
            appearance USE White
            geometry Box [ size 4.6 0.4 0.6 ]
        }
    ]
}

# Roof pieces
Transform {
    translation -1.15 7.12 0.0
    rotation 0.0 0.0 1.0 0.524
    children [
        DEF Roof Shape {
            appearance USE White
            geometry Box [ size 2.86 0.4 0.6 ]
        }
    ]
}
Transform {
    translation 1.15 7.12 0.0
    rotation 0.0 0.0 1.0 -0.524
    children [ USE Roof ]
}
You can use multiple Transform nodes, and Transform nodes as children of Transform nodes to build complex structures. Figure 14, for instance, extends Figure 13 by repeating the same arch structure three times. Each repetition turns the arch further around the Y axis. With a floor added, the resulting structure is a rotunda, or gazebo-like building.
Figure 14. A rotunda created by using four archways, rotated about the Y axis.
Scaling coordinate systems
The Transform node type’s translation and rotation fields enable you to create a new coordinate system that is positioned and oriented as you desire. In addition, you can change the size of shapes within a new coordinate system using a Transform node type’s scale field.

In the real world, construction blueprints provide a scaling factor that indicates a ratio between the size of the desired construction, and that described by the blueprints. For instance, a scaling factor of 10.0 indicates that the desired construction is 10.0 times larger than that depicted in the blueprints. Similarly, a scaling factor of 0.5 indicates construction should be half the size of that shown in the blueprints.

VRML 2.0’s Transform node type uses a similar scaling factor to indicate the amount to increase or decrease the size of shapes within a new coordinate system. A scaling factor of 10.0 increases shape sizes, growing them ten-fold. A scaling factor of 0.5 reduces shapes to half-size.

You can scale a coordinate system’s shapes by any positive factor. Factors between 0.0 and 1.0 decrease shape size, while those greater than 1.0 increase the size of shapes. A scaling factor of 1.0 leaves shape sizes unchanged.

To enable you to warp shapes, you can provide different scaling factors for the X, Y, and Z directions. For instance, you can stretch a sphere into an ellipsoid, or flatten a cone into a triangle.

The Transform node type, yet again
As before, the Transform node type creates a new coordinate system relative to its parent coordinate system.

<table>
<thead>
<tr>
<th>Syntax: Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform {</td>
</tr>
<tr>
<td>translation   0.0 0.0 0.0 # exposedField SFVec3f</td>
</tr>
<tr>
<td>rotation      0.0 0.0 1.0 0.0 # exposedField SF Rotation</td>
</tr>
<tr>
<td>scale         1.0 1.0 1.0 # exposedField SFVec3f</td>
</tr>
<tr>
<td>. . .</td>
</tr>
<tr>
<td>children      [ ] # exposedField MFNode</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

The values of the scale field specify positive X, Y, and Z scaling factors with which to increase or decrease the size of the new coordinate system and any shapes built within it. The default field values specify a 1.0 scaling factor for the X, Y, and Z directions and result in no size change.

Scaling, rotation, and translation can be used together first to scale a coordinate system, then orient it and position it relative to a parent coordinate system.

There are several more fields in the Transform node type. Discussion of these less commonly used fields is left to a future column.

Experimenting with coordinate system scaling
Figure 15 shows a spacecraft built entirely with spheres. Two spheres are scaled nearly flat, rotated, and positioned to form swept-back wings. Another sphere is elongated to form the
fuselage. A final sphere is scaled and positioned to form a cockpit dome.
#VRML V2.0 utf8

# Wing
Transform {
  translation 0.0 0.0 -0.9
  rotation 0.0 1.0 0.0 0.52
  scale 0.4 0.035 1.5
  children [ DEF WingSphere Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0.7 0.7 1.0
      }
    }
    geometry Sphere {}
  } ]
}

Transform {
  translation 0.0 0.0 0.9
  rotation 0.0 1.0 0.0 -0.52
  scale 0.4 0.035 1.5
  children [ USE WingSphere ]
}

# Fuselage
Transform {
  scale 2.0 0.2 0.5
  children [ Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0.5 0.5 1.0
      }
    }
    geometry Sphere {}
  } ]
}

# Dome
Transform {
  scale 0.6 0.4 0.375
  children [ Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0.7 0.5 1.0
      }
    }
    geometry Sphere {}
  } ]
}
You can use multiple `Transform` nodes to create complex structures, scaling, rotating, and translating each new coordinate system. Figure 16, for example, extends Figure 15 by adding a tail and engines to the spacecraft. The tail is formed by repeating the wing and fuselage shapes, scaled down and translated into position. The engines use two more scaled and translated spheres.

```vrml
#VRML V2.0 utf8
# Wing
DEF LeftWing Transform { 
  translation 0.0 0.0 -0.9 
  rotation 0.0 1.0 0.0 0.52 
  scale 0.4 0.035 1.5 
  children [ 
    DEF WingSphere Shape { 
      appearance Appearance { 
        material Material { 
          diffuseColor 0.7 0.7 1.0 
        } 
        geometry Sphere { } 
      } 
    ] 
  } 
}
DEF RightWing Transform { 
  translation 0.0 0.0 0.9 
  rotation 0.0 1.0 0.0 -0.52 
  scale 0.4 0.035 1.5 
  children [ USE WingSphere ] 
}
# Fuselage
DEF Fuselage Transform { 
  scale 2.0 0.2 0.5 
  children [ 
    DEF FuselageSphere Shape { 
      appearance Appearance { 
        material Material { 
          diffuseColor 0.5 0.5 1.0 
        } 
        geometry Sphere { } 
      } 
    ] 
  } 
}
# Dome
```
Next in the VRML Technique column

The **Transform** node type is clearly a powerful and essential VRML 2.0 feature enabling you to construct complex structures in your virtual world. Next month I will continue discussion of the **Transform** node type and introduce VRML 2.0 features for animating the position, orientation, and scale of coordinate systems and their shapes.
**Resources**

- A list of David Nadeau’s VRML Technique columns in *NetscapeWorld*
- VRML 2.0 browsers *NetscapeWorld*’s guide to finding and installing a VRML browser on your computer.
- VRML 2.0 glossary
- *NetscapeWorld*’s VRML vendors chart A handy reference of VRML browser and server companies including their plug-ins to Web browsers -- with updated items in bold to aid your review -- and links to all the vendors.
  http://www.netscapeworld.com/netscapeworld/common/nw.vrmltable.html
- The UTF-8 character set sidebar accompanying the first VRML Technique column.

**Specifications**

- VRML 2.0 specification http://vag.vrml.org/VRML2.0/FINAL/
- ISO 10646-1:1993 Universal Character Set (UCS) specification sales information  
  http://www.iso.ch/cate/d18741.html
- UTF-8 character encoding scheme for UCS  
  http://www.dkug.dk/JTC1/SC2/WG2/docs/n1335

**Sites**

- The VRML Repository http://www.sdsc.edu/vrml
- VRML Architecture Group http://vag.vrml.org

**About the author**

David R. Nadeau is a co-author of *The VRML 2.0 Sourcebook*, published by John Wiley & Sons and written with Andrea L. Ames and John L. Moreland. David is a staff researcher at the San Diego Supercomputer Center where he is a specialist in 3-D computer graphics, virtual reality, and scientific visualization. He is also the creator of *The VRML Repository*, a Web site providing extensive information on VRML software, documentation, and 3-D worlds.

⚠️ You can buy David R. Nadeau’s *The VRML 2.0 Sourcebook* at a 20% discount from Amazon.com Books.

**Feedback:** nweditors@netscapeworld.com  
**URL:** http://www.netscapeworld.com/netscapeworld/nw-01-1997/nw-01-vrmltechnique.html  
**Last updated:** Tuesday, March 11, 1997
Animating shapes

How to animate the position, orientation, and size of shapes in VRML 2.0

By David R. Nadeau

Summary
Perhaps the most exciting aspects of VRML 2.0 are features that enable you to create dynamic, animated virtual environments. You can make shapes fly hither and yon, spin about, grow and shrink, change color, fade in and out, morph from one form to another, and much more.

This month’s VRML Technique column introduces VRML 2.0’s animation features and shows how you can use them together with the Transform node type to create animations that position, orient, and resize shapes. Along the way, I introduce VRML 2.0’s animation circuit concept, explain events, and present the ROUTE statement. (5,300 words)

Table of contents

<table>
<thead>
<tr>
<th>Animating shapes</th>
<th>Describing how to animate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Describing animations</td>
<td>● Keyframe animation</td>
</tr>
<tr>
<td>Understanding events and animation circuits</td>
<td>● Linear interpolation</td>
</tr>
<tr>
<td>● Inputs and outputs</td>
<td>● The PositionInterpolator node type</td>
</tr>
<tr>
<td>● EventIn and eventOut data types</td>
<td>● The OrientationInterpolator node type</td>
</tr>
<tr>
<td>Building animation circuits</td>
<td>● Using a PositionInterpolator node</td>
</tr>
<tr>
<td>● The ROUTE statement</td>
<td>Experimenting with VRML 2.0 animation</td>
</tr>
<tr>
<td>Describing when to animate</td>
<td>● Using an OrientationInterpolator node</td>
</tr>
<tr>
<td>● The TimeSensor node type</td>
<td>● Animating multiple shapes using the same interpolator</td>
</tr>
<tr>
<td>● Using TimeSensor nodes</td>
<td>● Using multiple TimeSensor nodes</td>
</tr>
<tr>
<td>● Using start and stop times</td>
<td>● Using longer motion paths</td>
</tr>
</tbody>
</table>

Next in the VRML Technique column

Resources

About the author

Animating shapes
Last month’s column ("Building virtual structures") introduced VRML 2.0’s Transform node type and discussed its use in creating new coordinate systems. Using the translation, rotation,
and scale fields of a Transform node, you can position, orient, and resize shapes built within a new coordinate system. These features enable you to create 3D virtual structures and walk through them interactively within a VRML 2.0 browser.

(See NetscapeWorld’s sidebar on VRML 2.0 Browsers for information on obtaining and installing VRML 2.0 browsers. Also see the NetscapeWorld VRML Vendors chart for a list of VRML browsers and plug-ins, and our glossary of VRML 2.0 terms. You need a VRML browser or plug-in to view the 3D examples presented in this series.)

With VRML 2.0’s animation features, you can create virtual structures with moving parts. A windmill’s sails can rotate and an escalator’s stairs can slide up or down. Cars can move endlessly in a virtual city’s traffic patterns, a sun can rise and set daily, clouds can slide across the sky, and much more. Figure 1 shows two sample animated virtual worlds. Click on an image to load the associated VRML 2.0 world into your VRML 2.0 browser. The captions below each figure give the size of the world in bytes, and the expected download time.

![Image of a windmill with rotating sails](image1.png)  ![Image of a bouncing ball and glowing rings](image2.png)

(a) A windmill with rotating sails (11 kilobytes = 7 seconds @ 14.4bps)  
(b) A bouncing ball and glowing rings (18 kilobytes = 13 seconds @ 14.4bps)

**Figure 1. Sample VRML 2.0 worlds containing automatic continuous animations**

Click on an image to load the world.

Viewing tip: Once loaded into your VRML 2.0 browser, if these worlds run a little slowly, try reducing the size of the browser window. A smaller window means there is less screen area for the browser to redraw each time you move in the world. This reduction in drawing area speeds up the browser and enables it to animate more smoothly, or respond more quickly to user actions.

---

**Describing animations**

To build an animation for your VRML 2.0 virtual world, you need to describe:

- what to animate
- how to animate
- when to animate

For example, if you want to animate an elevator going up and down all day, then the "elevator" is what to animate, the "up and down" motion is how to animate, and "all day" is when to animate. The what, how, and when parts of a VRML 2.0 animation are fairly independent. For instance, you can increase the up and down range of the elevator animation without changing the elevator
shape or "all day" time frame.

The independence of what, how, and when parts of a VRML 2.0 animation description also enable you to create new animations by re-using parts of prior animations. For example, you can create a dramatically bouncing pogo-stick animation by combining a new pogo-stick shape with the "up and down" and "all day" parts of the elevator animation.

To enable this kind of easy mix-and-match animation creation, VRML 2.0 provides separate nodes dedicated to describing the what, how, and when parts of an animation. Together with these nodes, VRML 2.0 provides a special ROUTE statement that hooks what, how, and when nodes together into a complete animation description.

You can describe the what part of an animation using the Shape and Transform node types discussed in previous columns. To enable you to describe how and when to animate, this column introduces three new node types: TimeSensor, PositionInterpolator, and OrientationInterpolator.

For instance, the VRML 2.0 example in Figure 2 builds an elevator that continually moves up and down. To describe what to animate, the VRML file uses several Box, Shape, and Transform nodes to build and position the floor, ceiling, and three walls of a simple elevator. The how and when parts of the elevator’s animation are described by TimeSensor and PositionInterpolator nodes discussed in the next few sections.
Understanding events and animation circuits

To build an animation out of what, how, and when nodes, VRML 2.0 enables you to wire nodes together into an animation circuit. Each node in the circuit acts like an electronic component with its own input and output connection points. By wiring the output of one node into the input of another node, you can create a route along which can flow data values called events.

For example, to make an elevator go up and down you can wire the output of a node that generates up and down events into a node that creates an elevator shape. Each time an up or down event flows along the wired route to the elevator, the elevator moves up or down. If the events stop flowing, the elevator stops moving. Using animation circuits like this you can animate the position, orientation, and size of shapes or change other shape attributes.

Inputs and outputs

An eventIn is an input connection point for a node. An eventOut is an output connection point. Like fields, a node type’s eventIns and eventOuts have names. Different node types have different eventIns and eventOuts available. The Material node type, for instance, has a set_diffuseColor eventIn for changing the shading color of a shape. The PositionInterpolator node type, discussed later in this column, has a value_changed eventOut that outputs position events.

An exposed field is a special type of field that combines together a standard field, an eventIn to set
that field, and an eventOut that outputs the field value each time the field is set. The `translation` exposed field of a `Transform` node, for example, has an implicit `set_translation` eventIn, and an implicit `translation_changed` eventOut.

The syntax boxes used in this column provide a quick summary of a node type’s fields, exposed fields, eventIns, and eventOuts. The syntax box below, for instance, describes the `Transform` node type introduced in last month’s column. Notice that each of the node type’s fields are exposed fields and therefore have implicit eventIns and eventOuts.

```plaintext
Syntax: Transform
Transform {
  translation    0.0 0.0 0.0     # exposedField SFVec3f
  rotation       0.0 0.0 1.0 0.0 # exposedField SFRotation
  scale          1.0 1.0 1.0     # exposedField SFVec3f
  ...            
  children       [ ]             # exposedField MFNode
}
```

### EventIn and eventOut data types
Like fields, eventIns and eventOuts have a *data type*. The data type of an eventOut indicates the kind of event data it sends when wired into an animation circuit. The data type of an eventIn indicates the kind of event data it expects from a circuit.

When wiring an animation circuit from an eventOut to an eventIn, the data types of the eventOut and eventIn must match. It is inappropriate, for instance, to wire an eventOut that generates color data into an eventIn that expects positions.

### Building animation circuits
Like a computer circuit board, a virtual world’s animation circuitry is built by wiring components together one at a time. Each VRML 2.0 wire, or *route*, connects two nodes together, enabling events to flow between the nodes.

### The ROUTE statement
VRML 2.0’s `ROUTE` statement wires a route between an eventOut of one node and the eventIn of another.

```plaintext
Syntax: ROUTE
ROUTE outName.eventOutName  TO  inName.eventInName
```

Every `ROUTE` statement includes four pieces:

- `outName` the name of a node that sends events
- `eventOutName` the name of an eventOut for the sending node
- `inName` the name of a node that receives events
- `eventInName` the name of an eventIn for the receiving node

To wire a route between sending and receiving nodes, both nodes must have names. You can give
a node a name using the **DEF** syntax introduced in last month’s column.

Along with the names of sending and receiving nodes, the **ROUTE** statement selects the sender’s eventOut and the receiver’s eventIn to connect together. For example, the following **ROUTE** statement connects the **value_changed** eventOut of a node named **HowToMove** to the **set_translation** eventIn of a node named **WhatToMove**.

```
ROUTE HowToMove.value_changed TO WhatToMove.set_translation
```

The example in Figure 3, below, extends the elevator example shown in Figure 2. The example wires routes between two pairs of nodes. The first **ROUTE** statement wires a route between **AllDay** node’s **fraction_changed** eventOut and the **UpAndDown** node’s **set_fraction** eventIn. The second **ROUTE** statement wires a route between the **UpAndDown** node’s **value_changed** eventOut and the **Elevator** node’s **set_translation** eventIn. This completed animation circuit enables events to flow from **AllDay** into **UpAndDown**, and then from **UpAndDown** into **Elevator**, causing the elevator to animate.

```
#VRML V2.0 utf8
DEF Elevator Transform { . . . }
DEF AllDay TimeSensor { . . . }
DEF UpAndDown PositionInterpolator { . . . }

ROUTE AllDay.fraction_changed TO UpAndDown.set_fraction
ROUTE UpAndDown.value_changed TO Elevator.set_translation
```

Figure 3. What, how, and when parts of an elevator animation wired together using two routes

*Click on the image to load the world.*

A VRML 2.0 file may contain any number of **ROUTE** statements, each one of which adds another wire into an animation circuit. The same node inputs and outputs may be wired into multiple routes, enabling a single eventOut to connect to multiple eventIns, or a single eventIn to connect to multiple eventOuts.

**ROUTE** statements may be placed anywhere within a VRML 2.0 file. Typically **ROUTE** statements are placed at the end of the VRML 2.0 file to make it easy to find them while editing.
Describing when to animate
To indicate when to animate, an animation needs to sense the passage of time. Such time sensing abilities are provided by VRML 2.0’s TimeSensor node type.

The TimeSensor node type
In an animation circuit, a TimeSensor node provides eventOuts that can be wired into other nodes. As time ticks away, the sensor outputs a variety of time-related values that you can use to start and stop animations and control their playback speed.

```
Syntax: TimeSensor

TimeSensor {
  enabled          TRUE            # exposedField SFBool
  startTime        0.0             # exposedField SFTime
  stopTime         0.0             # exposedField SFTime
  cycleInterval    1.0             # exposedField SFTime
  loop             FALSE           # exposedField SFBool
  isActive                         # eventOut     SFBool
  time                             # eventOut     SFTime
  cycleTime                        # eventOut     SFTime
  fraction_changed                 # eventOut     SFFloat
}
```

A TimeSensor node acts a little like an electronic stop-watch. When turned on, a TimeSensor node starts and stops when you tell it to, only generating outputs between the start and stop times.

The TRUE or FALSE value of the enabled exposed field turns the sensor on and off. The values of the startTime and stopTime exposed fields tell the sensor when to start generating events, and when to stop.

Start and stop time values are measured in seconds, counting from 12:00 midnight, GMT, January 1st, 1970. This seemingly odd basis for measuring time is an artifact of the computer’s internal way of measuring time. In practice, this is not a problem since the values of the startTime and stopTime fields are usually not set explicitly within a VRML 2.0 file. Instead, these fields are typically wired into an animation circuit and set automatically via the output from some other node.

When the sensor’s stop time is later than the start time, the sensor runs from the start time to the stop time, then stops, just like a stop-watch. However, if the stop time is earlier than the start time, then the stop time is ignored and the sensor runs forever.

Sensors that run forever are common in VRML worlds. Such sensors are used to control animations that play back continually, such as animations that make the sun rise and fall, or that cycle stop lights.

The isActive eventOut sends a TRUE event at the start time, and a FALSE event at the stop time. Using these output events, you can use a TimeSensor node like an alarm clock and trigger animation actions at specific times.

The time eventOut repeatedly sends the current time while the sensor is running. Output time values are measured in seconds since 12:00 midnight, GMT, January 1st, 1970.
The remaining exposed fields and eventOuts work together to enable a **TimeSensor** node to manage a concept of *fractional time*. Normal, absolute time, like that in the real world, always marches forward. By contrast, VRML’s fractional time is cyclical: it starts at a given time, advances for awhile, then starts over. This kind of fractional time is particularly useful for creating repeating, cyclical animations. A windmill animation, for instance, requires that the windmill’s sails rotate 360.0 degrees, then start over in a repeating cycle. Similarly, an orbiting virtual planet repeatedly rotates around a sun, cycle after cycle. The fractional time abilities of the **TimeSensor** node type are the principal mechanism by which VRML animations like these are controlled. All of the examples in the rest of this column use fractional times to control cyclical animations.

The value of the **cycleInterval** exposed field specifies the length of a single cycle, measured in seconds. The first cycle starts at the start time selected by the **startTime** exposed field.

The value of the **loop** exposed field indicates if the sensor should run for a single cycle or continue to cycle indefinitely. When the **loop** exposed field value is **FALSE**, the sensor runs for a single cycle then stops, even if the time in the **stopTime** field hasn’t been reached yet. When the **loop** exposed field value is **TRUE**, the sensor runs for a potentially infinite number of cycles, halting only when the stop time is reached, if ever.

The **cycleTime** eventOut sends the current time each time a cycle is started. Like the **time** eventOut, the time output by the **cycleTime** eventOut is measured in seconds since 12:00 midnight, GMT, January 1st, 1970.

The **fraction_changed** eventOut is the most important output of a **TimeSensor** node. During each cycle of the sensor, the **fraction_changed** eventOut sends floating-point number events that vary from 0.0 at the start of a cycle to 1.0 at the end. At the end of a cycle, the sensor’s fractional time resets back to 0.0, ready for the next cycle.

The fractional time outputs of a **TimeSensor** node are almost always wired into one of the **interpolator** node types discussed in the next section. In such an animation circuit, the **TimeSensor** node controls *when* to animate, and the interpolator nodes control *how* to animate.

### Using TimeSensor nodes

The example in Figure 4 shows an expanded version of the elevator example from Figures 2 and 3. To describe *when* to animate the elevator, the VRML file uses a **TimeSensor** node named **AllDay**. The node’s fields indicate it should loop through 4.0 second long cycles starting at 1.0 second after 12:00 midnight, GMT, January 1st, 1970. Since the stop time is 1.0 second earlier than the start time, the sensor ignores the stop time and cycles forever.

At each tick of the **TimeSensor** node, a fractional time event is output from the node’s **fraction_changed** eventOut and routed into an interpolator. The interpolator uses these fractional time values to compute positions with which to animate the elevator.
Using start and stop times
The use of start and stop times measured since 12:00 midnight, GMT, January 1st, 1970, may seem confusing at first. This is, however, a very powerful feature of VRML 2.0.

Conceptually, a VRML 2.0 virtual world exists independent from the real world. Your virtual world has its own shapes, and its own activities going on, even if you aren’t watching them right now. The animation circuits you wire together describe these activities. Once wired, you can let go and the animations continue on without you.

When you set up a looping TimeSensor node, the sensor cycles over and over from the start time to the stop time. That cycling continues, conceptually, whether or not the world is currently loaded into your VRML 2.0 browser. Each time you load your world into your browser, the browser computes what is currently happening in your virtual world and displays it.

For example, imagine that you set up a TimeSensor node so that an animation starts at 12:00 midnight, PST, the morning of February 1st, 1997, and stops at the same time on February 28th, 1997. If you load this virtual world any time in February, you’ll see the animation. However, if you load the world in January, the animation won’t have started yet, and if you load it in March, the animation will already have completed.

The use of start and stop times for animations enables you to give your virtual worlds a history and a future. You can specify exactly what has happened, and what will happen in your world.
For example, the elevator VRML file shown in Figure 4 above uses a TimeSensor node that started its cycling 27 years ago, 1 second after midnight, GMT, January 1st, 1970. Since then, the sensor has cycled over and over every 4.0 seconds until the present day. Since the sensor has a stop time earlier than its start time, the stop time is ignored and the sensor will continue cycling forever. When you load this world into your browser, you experience a brief portion of this elevator’s continuing up and down existence.

VRML 2.0’s use of 12:00 midnight, GMT, January 1st, 1970 as the beginning of your virtual world calendar is arbitrary. This calendar starting date and time is one commonly used by other computer time measurements, making it a convenient choice for VRML 2.0.

**Describing how to animate**

To describe how to animate, VRML 2.0 provides a variety of interpolator node types. Two of the most common interpolators are the PositionInterpolator and the OrientationInterpolator node types. Both of these node types use a TimeSensor node’s fractional time output to help them compute and output position or orientation values for your animations. By wiring a route from an interpolator node into a shape’s Transform node, you can use an interpolator’s position or orientation outputs to animate the position or orientation of a shape.

**Keyframe animation**

To animate a shape’s position or orientation, your animation description must provide a new position or orientation for every moment during which the shape is animating. For cyclical animations using a TimeSensor node’s fractional time, you only need to provide a new position or orientation for every fractional time value between 0.0 and 1.0.

The most straightforward approach for an animation description is to use a table of positions or orientations, one for each possible fractional time between 0.0 and 1.0. Unfortunately, there are an infinite number of possible fractional time values between 0.0 and 1.0, which makes a table like this impractical.

Instead, animation descriptions use a technique called keyframe animation, where a position or orientation is specified for only a few, key fractional times. The position or orientation values at these times are called key values. VRML 2.0’s interpolator nodes use these key fractional times and key values as a rough sketch of the animation and fill in the values between those specified as needed. Using keyframe animation, an animation description specifies only a few positions and orientations, instead of an infinite number of them.

For example, to cause an elevator to rise from the bottom floor to the top floor as fractional time proceeds from 0.0 to 1.0, a keyframe animation can use just two key fractional times, 0.0 and 1.0, and just two key values, the bottom and top of the elevator shaft. An interpolator node can automatically compute positions between these two key positions for fractional times between 0.0 and 1.0. At fractional time 0.5, for instance, an interpolator computes the elevator’s position as exactly halfway between the bottom and top floors.

**Linear interpolation**

All VRML 2.0 interpolator nodes use linear interpolation to compute intermediate values between the key values you provide. Linear interpolation can be visualized by first imagining two key-value positions plotted as dots on a piece of graph paper. Next, using an imaginary ruler, draw a linear, or straight line between the two dots. All points along the drawn line are intermediate
A linear interpolator computes an intermediate position or orientation each time an output is needed. Any number of intermediate values can be computed between your key positions and orientations.

The use of interpolation is especially important when playing an animation at different speeds. For a quick animation, your VRML browser may only have time to draw the world a few times between the time the animation starts and the time it stops. In this case, your browser may only need to linearly interpolate values at a few fractional times between the key fractional times you provide.

For a slow animation, your VRML browser may have the time to draw the world many times and may need a large number of interpolated positions or orientations. In this case, your browser may interpolate values at many fractional times between your key fractional times.

Using keyframe animation and linear interpolation, you can describe an animation independent of the playback speed of the animation. During playback, an appropriate number of intermediate values are computed automatically.

**The PositionInterpolator node type**
The **PositionInterpolator** node type describes a linear interpolator for use in the keyframe animation of shape positions.

```plaintext
Syntax: PositionInterpolator
PositionInterpolator {
    key              [ ]             # exposedField MFFloat
    keyValue         [ ]             # exposedField MFVec3f
    set_fraction                     # eventIn      SFFloat
    value_changed                    # eventOut     SFVec3f
}
```

The value of the **key** exposed field specifies a list of key fractional times. Typically, fractional times are between 0.0 and 1.0, such as those output by a **TimeSensor** node’s **fraction_changed** eventOut. Key fractional times, however, may be positive or negative floating-point numbers of any size as long as they are listed in non-decreasing order.

The value of the **keyValue** exposed field specifies a list of key positions. Each key position is a 3D coordinate composed of an X, a Y, and a Z distance.

The key fractional times and positions are used together so that the first key fractional time specifies the time for the first key position, the second key fractional time for the second key position, and so forth. The lists, together, may provide any number of fractional times and positions, but both lists must contain the same number of values.

The **set_fraction** eventIn accepts floating-point fractional time events, such as those output by a **TimeSensor** node’s **fraction_changed** eventOut. Each time a fractional time event is received, the **PositionInterpolator** node computes by linear interpolation a new position based upon the list of key positions and their corresponding key fractional times. The new computed position is output via the **value_changed** eventOut.
In typical use, the \texttt{value\_changed} eventOut of a \texttt{PositionInterpolator} node is routed into a \texttt{Transform} node’s \texttt{set\_translation} eventIn. Each time the interpolator outputs a new position event, the \texttt{Transform} node sets its \texttt{translation} field, causing the shapes built within the \texttt{Transform} node’s coordinate system to change position.

**The OrientationInterpolator node type**
The \texttt{OrientationInterpolator} node type describes a linear interpolator for use in keyframe animation of shape orientations.

```
Syntax: OrientationInterpolator
OrientationInterpolator {
    key              [ ]             # exposedField MFFloat
    keyValue         [ ]             # exposedField MFRotation
    set\_fraction                     # eventIn      SFFloat
    value\_changed                    # eventOut     SFRotation
}
```

The \texttt{OrientationInterpolator} node type performs in a way analogous to the \texttt{PositionInterpolator} node type. The \texttt{key} exposed field specifies a list of key fractional times, while the \texttt{keyValue} exposed field specifies a list of key rotations. Each key rotation is a set of four floating-point numbers where the first three values describe a rotation axis, and the last value describes a rotation angle about that axis, measured in radians. \texttt{OrientationInterpolator} node type rotations are identical to those used in the \texttt{rotation} field of the \texttt{Transform} node type described in last month’s column.

Similar to the \texttt{PositionInterpolator} node type, the \texttt{set\_fraction} eventIn accepts a fractional time event and causes the interpolator to compute and output a new rotation value via the \texttt{value\_changed} eventOut. Output rotations are computed by linearly interpolating between the list of key rotations.

**Using a PositionInterpolator node**
Figure 5 expands upon the elevator example used in Figures 2, 3, and 4 earlier. To describe how the elevator moves up and down, this example uses a \texttt{PositionInterpolator} node with three key fractional times and three key positions. At fractional time 0.0, the associate key position is at the origin: 0.0 0.0 0.0. At fractional time 0.5, the associated position is 2.0 units up the Y axis from the origin. At fractional time 1.0, the associated position is again at the origin. When this animation plays back, the interpolator automatically generates intermediate positions up and down the elevator’s path.
Experimenting with VRML 2.0 animation
The elevator example shown in Figures 2 through 5 uses a TimeSensor node to control a PositionInterpolator node. On each TimeSensor node output, the interpolator computes a new 3D position and sends it into the translation field of a Transform node. In response, the Transform node adjusts the position of its coordinate system, thereby moving the shapes making up the elevator.

You can use interpolators to create a variety of animations, varying positions and orientations of multiple shapes in your world. The examples below illustrate a few uses of VRML interpolators.

Using an OrientationInterpolator node
You can use an OrientationInterpolator node to cause a shape to spin. The example in Figure 6, for instance, uses a TimeSensor node to control an OrientationInterpolator node, which in turn changes the rotation field value of a purple bar’s Transform node. As the TimeSensor node ticks away, the interpolator computes new rotations and the purple bar spins.
Figure 6. An animation circuit to spin a purple bar
Click on the image to load the world.

**Animating multiple shapes using the same interpolator**
You can use a single interpolator node to animate the position or orientation of more than one shape. For instance, Figure 7 extends the spinning purple bar example of Figure 6, adding five more spinning purple bars arranged on the six sides of a cube. A single TimeSensor node controls a single OrientationInterpolator node. The interpolator’s rotation value outputs are routed into all six purple bars, causing them all to rotate in sync.
#VRML V2.0 utf8
#  Rotating bars positioned as six faces of a cube
# DEF Bar1 Transform {
  translation 0.0 0.0 1.5
  children DEF PurpleBar Shape {
    appearance Appearance {
      material Material { diffuseColor 0.5 0.0 1.0 }
    }
    geometry Box { size 0.5 3.0 0.5 }
  }
  Transform {
    rotation 0.0 1.0 0.0  1.57
    children DEF Bar2 Transform {
      translation 0.0 0.0 1.5
      children USE PurpleBar
    }
  }
  Transform {
    rotation 0.0 1.0 0.0  3.14
    children DEF Bar3 Transform {
      translation 0.0 0.0 1.5
      children USE PurpleBar
    }
  }
  Transform {
    rotation 0.0 1.0 0.0 -1.57
    children DEF Bar4 Transform {
      translation 0.0 0.0 1.5
      children USE PurpleBar
    }
  }
  Transform {
    translation 0.0 1.5 0.0
    rotation 1.0 0.0 0.0 -1.57
    children DEF Bar5 Transform {
      children USE PurpleBar
    }
  }
  Transform {
    translation 0.0 -1.5 0.0
    rotation 1.0 0.0 0.0  1.57
    children DEF Bar6 Transform {
      children USE PurpleBar
    }
  }
}
#  Master timer used for all rotating bars
# DEF Forever TimeSensor {
  cycleInterval 6.0
  loop TRUE
# Master spinner used for all rotating bars

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

ROUTE Forever.fraction_changed TO FullCircle.set_fraction  
ROUTE FullCircle.value_changed TO Bar1.set_rotation  
ROUTE FullCircle.value_changed TO Bar2.set_rotation  
ROUTE FullCircle.value_changed TO Bar3.set_rotation  
ROUTE FullCircle.value_changed TO Bar4.set_rotation  
ROUTE FullCircle.value_changed TO Bar5.set_rotation  
ROUTE FullCircle.value_changed TO Bar6.set_rotation

Figure 7. An animation circuit to spin six purple bars using a single OrientationInterpolator

Click on the image to load the world.

Using multiple TimeSensor nodes

Each of the previous examples use a single TimeSensor node to control all the motion in the world. You can also create animation circuits with multiple TimeSensor nodes.

The example in Figure 8 creates an abbreviated model of the Solar System. The model includes a stationary central Sun, and three orbiting planets: Mercury, Venus, and Earth. Each planet has a different color, size, and orbital radius. To make the planets orbit the Sun, each planet is animated by the output from a separate OrientationInterpolator node describing a circular path for the planet. To simulate (very roughly) the different orbital speeds of the planets, each planet’s OrientationInterpolator node is controlled by a separate TimeSensor node with its own cycle length. In this virtual world, Mercury’s TimeSensor node takes 2.0 seconds to complete a cycle, while Venus’ takes 3.5 seconds and Earth’s takes 5.0 seconds.

![Solar System Animation](image)

#VRML V2.0 utf8  
#  Stationary sun and three orbiting planets
Shape {
    appearance Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
    }
    geometry Sphere {
    }
}
DEF Mercury Transform {
    children Transform {
        translation 2.0 0.0 0.0
        children Shape {
            appearance Appearance {
                material Material { diffuseColor 0.9 0.2 0.0 }
            }
            geometry Sphere { radius 0.2 }
        }
    }
}
DEF Venus Transform {
    children Transform {
        translation 3.0 0.0 0.0
        children Shape {
            appearance Appearance {
                material Material { diffuseColor 0.5 0.5 0.8 }
            }
            geometry Sphere { radius 0.25 }
        }
    }
}
DEF Earth Transform {
    children Transform {
        translation 4.0 0.0 0.0
        children Shape {
            appearance Appearance {
                material Material { diffuseColor 0.0 0.5 1.0 }
            }
            geometry Sphere { radius 0.4 }
        }
    }
}
#
#  Timers, one per planet
#
DEF MercuryForever TimeSensor {
    cycleInterval 2.0
    loop TRUE
    startTime 1.0
    stopTime 0.0
}
DEF VenusForever TimeSensor {
    cycleInterval 3.5
    loop TRUE
    startTime 1.0
    stopTime 0.0
}
DEF EarthForever TimeSensor {
    cycleInterval 5.0
    loop TRUE
    startTime 1.0
    stopTime 0.0
}
#
#  Orbital paths, one per planet (all identical)
#
DEF MercuryOrbit OrientationInterpolator {
    key [ 0.0, 0.5, 1.0 ]
    keyValue [
        0.0 1.0 0.0 0.0,
        0.0 1.0 0.0 3.14,
        0.0 1.0 0.0 6.28,
        ]
}
DEF VenusOrbit OrientationInterpolator {
    key [ 0.0, 0.5, 1.0 ]
    keyValue [
        0.0 1.0 0.0 0.0,
        0.0 1.0 0.0 3.14,
        0.0 1.0 0.0 6.28,
        ]
}
Using longer motion paths

In the elevator example shown earlier, the PositionInterpolator node uses only three key positions to describe the up and down motion path of the elevator. Similarly, each of the examples using OrientationInterpolator nodes use only three key orientations. You can, however, use any number of positions or orientations in the key value list for an interpolator. For very complex motion paths, you may have hundreds, or even thousands of separate positions or orientations listed in an interpolator.

The example shown in Figure 9 builds a simple escalator with four stairs. Each stair is identical and follows an identical motion path containing seven positions. The motion path positions describe a diagonal, upward path for an escalator stair. At the top of the escalator, the stair drops down and returns to the bottom of the escalator, ready to travel upwards again on the next cycle. An identical motion path is specified in each of four PositionInterpolator nodes, one per stair. To cause the four stairs to travel upwards, offset from each other, each stair uses its own TimeSensor node with an offset start time.

Click on the image to load the world.
appearance Appearance {
  material Material { diffuseColor 0.0 0.5 1.0 }
} 
geometry Box { size 1.0 0.1 2.0 }
}

DEF Stair2 Transform { children USE Platform }
DEF Stair3 Transform { children USE Platform }
DEF Stair4 Transform { children USE Platform }

# # Four timers, one per stair (each offset by 1 second)
#
DEF Forever1 TimeSensor {
  cycleInterval 4.0
  loop TRUE
  startTime 1.0
  stopTime 0.0
}
DEF Forever2 TimeSensor {
  cycleInterval 4.0
  loop TRUE
  startTime 2.0
  stopTime 0.0
}
DEF Forever3 TimeSensor {
  cycleInterval 4.0
  loop TRUE
  startTime 3.0
  stopTime 0.0
}
DEF Forever4 TimeSensor {
  cycleInterval 4.0
  loop TRUE
  startTime 4.0
  stopTime 0.0
}

# # Four animation paths, one per stair (all identical)
#
DEF Diagonal1 PositionInterpolator {
  key [ 0.0, 0.4, 0.45, 0.5, 0.9, 0.95, 1.0 ]
  keyValue [
    0.0 0.0 0.0, 4.0 2.0 0.0,
    4.5 2.0 0.0, 4.5 1.8 0.0,
    0.5 -0.2 0.0, 0.0 -0.2 0.0,
    0.0 0.0 0.0,
  ]
}
DEF Diagonal2 PositionInterpolator {
  key [ 0.0, 0.4, 0.45, 0.5, 0.9, 0.95, 1.0 ]
  keyValue [
    0.0 0.0 0.0, 4.0 2.0 0.0,
    4.5 2.0 0.0, 4.5 1.8 0.0,
    0.5 -0.2 0.0, 0.0 -0.2 0.0,
    0.0 0.0 0.0,
  ]
}
DEF Diagonal3 PositionInterpolator {
  key [ 0.0, 0.4, 0.45, 0.5, 0.9, 0.95, 1.0 ]
  keyValue [
    0.0 0.0 0.0, 4.0 2.0 0.0,
    4.5 2.0 0.0, 4.5 1.8 0.0,
    0.5 -0.2 0.0, 0.0 -0.2 0.0,
    0.0 0.0 0.0,
  ]
}
DEF Diagonal4 PositionInterpolator {
  key [ 0.0, 0.4, 0.45, 0.5, 0.9, 0.95, 1.0 ]
  keyValue [
    0.0 0.0 0.0, 4.0 2.0 0.0,
    4.5 2.0 0.0, 4.5 1.8 0.0,
    0.5 -0.2 0.0, 0.0 -0.2 0.0,
    0.0 0.0 0.0,
  ]
}
ROUTE Forever1.fraction_changed TO Diagonal1.set_fraction
ROUTE Forever2.fraction_changed TO Diagonal2.set_fraction
ROUTE Forever3.fraction_changed TO Diagonal3.set_fraction
ROUTE Forever4.fraction_changed TO Diagonal4.set_fraction
ROUTE Diagonal1.value_changed   TO Stair1.set_translation
ROUTE Diagonal2.value_changed   TO Stair2.set_translation
ROUTE Diagonal3.value_changed   TO Stair3.set_translation
ROUTE Diagonal4.value_changed   TO Stair4.set_translation

Figure 9. A simple escalator containing four identical stairs traveling along identical diagonal paths, offset in time

Click on the image to load the world.

Animating the size of a shape
The PositionInterpolator node can be used to vary the position of a shape, and the OrientationInterpolator node used to vary a shape’s orientation. VRML 2.0 does not include an interpolator dedicated to animating a shape’s size. However, you can use a PositionInterpolator node to achieve this purpose.

Output events of a PositionInterpolator node include X, Y, and Z values with an SFVec3f data type. This data type is appropriate for use in animating a Transform node’s translation field value, as in the elevator examples. This data type is also appropriate for use in animating the scale field value of a Transform node.

The example in Figure 10 uses a PositionInterpolator node to animate the scale field value of a yellow sphere’s Transform node. Key values in the PositionInterpolator node are set to be X, Y, and Z scaling factors instead of 3D positions. Each output from the interpolator is routed into the Transform node’s scale field and changes the X, Y, and Z scaling factors for the yellow sphere. The effect of the animation is to repeatedly squish the yellow sphere.
Combining together multiple interpolators
You can create complex animations by combining together multiple interpolators. The example in Figure 11 extends the squishy yellow sphere example in Figure 10, creating a whimsical rotating gadget that repeatedly moves a yellow ball within reach of a pair of plunger shafts that slide in and squish the ball.

An OrientationInterpolator node rotates the gadget. A pair of PositionInterpolator nodes slide the plunger shafts in and out. Four more PositionInterpolator nodes squish one each of the yellow spheres on the gadget. A single TimeSensor node controls all of the interpolators.
# Weird rotating gadget with four squishable balls

DEF Gadget Transform {
  children [
    # Four squishable balls positioned around a circle
    DEF Squishy1 Transform {
      translation 0.0 0.0 4.0
      children DEF Ball Shape {
        appearance Appearance {
          material Material { diffuseColor 1.0 1.0 0.0 }
        }
        geometry Sphere {
      }
    }]
    DEF Squishy2 Transform {
      translation 0.0 4.0 0.0
      children USE Ball
    }
    DEF Squishy3 Transform {
      translation 0.0 0.0 -4.0
      children USE Ball
    }
    DEF Squishy4 Transform {
      translation 0.0 -4.0 0.0
      children USE Ball
    }
    # A central plate and spokes for the gadget
    Transform {
      rotation 0.0 0.0 1.0 1.57
      children Shape {
        appearance DEF Gray Appearance {
          material Material {} 
        }
        geometry Cylinder {
          radius 2.0
          height 0.2
        }
      }
    }
    DEF Spoke Shape {
      appearance USE Gray
      geometry Cylinder {
        height 6.0
        radius 0.3
      }
    }
  ]
}

# Sliding squishing apparatus with two shafts

Transform {
  translation 0.0 0.0 4.0
  children [
    # Left shaft
    DEF Left Transform {
      rotation 0.0 0.0 1.0 -1.57
      children DEF Shaft Transform {
        translation 0.0 -1.25 0.0
        children [
          # Main shaft
          Shape {
            appearance USE Gray
            geometry Cylinder {
              height 2.0
              radius 0.4
            }
          }
          # Squishing head on the shaft
          Transform {
            translation 0.0 1.125 0.0
            children Shape {
              appearance USE Gray
              geometry Cylinder {
                height 0.25
                radius 0.6
              }
            }
          }
        ]
      }
    }
  ]
}
# Right shaft
DEF Right Transform {
  translation 0.0 0.0 4.0
  rotation 0.0 0.0 1.0  1.57
  children USE Shaft
}

# Animation timer
# DEF Forever TimeSensor {
  cycleInterval 10.0
  loop TRUE
  startTime 1.0
  stopTime 0.0
}

# Rotation path for the ball-holder gadget
# DEF Rotater OrientationInterpolator {
  key [
    0.00, 0.0625, 0.125,
    0.25, 0.3125, 0.375,
    0.50, 0.5625, 0.625,
    0.75, 0.8125, 0.875,
    1.0
  ]
  keyValue [
    1.0 0.0 0.0 0.0, 1.0 0.0 0.0 0.0, 1.0 0.0 0.0 0.0,
    1.0 0.0 0.0 1.57, 1.0 0.0 0.0 1.57, 1.0 0.0 0.0 1.57,
    1.0 0.0 0.0 3.14, 1.0 0.0 0.0 3.14, 1.0 0.0 0.0 3.14,
    1.0 0.0 0.0 4.71, 1.0 0.0 0.0 4.71, 1.0 0.0 0.0 4.71,
    1.0 0.0 0.0 6.28,
  ]
}

# Scaling for the four squishable balls
# DEF Squisher1 PositionInterpolator {
  key [
    0.00, 0.0625, 0.125,
    0.25, 0.3125, 0.375,
    0.50, 0.5625, 0.625,
    0.75, 0.8125, 0.875,
    1.0
  ]
  keyValue [
    # Scaling factors, not positions...
    1.0 1.0 1.0, 0.5 1.4 1.4, 1.0 1.0 1.0,
    1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0,
    1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0,
    1.0 1.0 1.0,
  ]
}

DEF Squisher2 PositionInterpolator {
  key [
    0.00, 0.0625, 0.125,
    0.25, 0.3125, 0.375,
    0.50, 0.5625, 0.625,
    0.75, 0.8125, 0.875,
    1.0
  ]
  keyValue [
    # Scaling factors, not positions...
    1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0,
    1.0 1.0 1.0, 0.5 1.4 1.4, 1.0 1.0 1.0,
    1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0,
    1.0 1.0 1.0,
  ]
}

DEF Squisher3 PositionInterpolator {
key [
  0.00, 0.0625, 0.125, 
  0.25, 0.3125, 0.375, 
  0.50, 0.5625, 0.625, 
  0.75, 0.8125, 0.875, 
  1.0
]
keyValue [
  # Scaling factors, not positions... 
  1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0, 
  1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0, 
  0.5 1.4 1.4, 1.0 1.0 1.0, 
  1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0, 
  1.0 1.0 1.0,
]
}

DEF Squisher4 PositionInterpolator {
  key [
    0.00, 0.0625, 0.125, 
    0.25, 0.3125, 0.375, 
    0.50, 0.5625, 0.625, 
    0.75, 0.8125, 0.875, 
    1.0
  ]
  keyValue [
    # Scaling factors, not positions... 
    1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0, 
    1.0 1.0 1.0, 1.0 1.0 1.0, 1.0 1.0 1.0, 
    1.0 1.0 1.0, 0.5 1.4 1.4, 1.0 1.0 1.0, 
    1.0 1.0 1.0, 
  ]
}

#  Paths for the left and right squisher shafts
#
DEF LeftToRight PositionInterpolator {
  key [
    0.00, 0.0625, 0.125, 
    0.25, 0.3125, 0.375, 
    0.50, 0.5625, 0.625, 
    0.75, 0.8125, 0.875, 
    1.0
  ]
  keyValue [
    -1.0 0.0 0.0, -0.4 0.0 0.0, -1.0 0.0 0.0, 
    -1.0 0.0 0.0, -0.4 0.0 0.0, -1.0 0.0 0.0, 
    -1.0 0.0 0.0, -0.4 0.0 0.0, -1.0 0.0 0.0, 
    -1.0 0.0 0.0, 
  ]
}

DEF RightToLeft PositionInterpolator {
  key [
    0.00, 0.0625, 0.125, 
    0.25, 0.3125, 0.375, 
    0.50, 0.5625, 0.625, 
    0.75, 0.8125, 0.875, 
    1.0
  ]
  keyValue [
    1.0 0.0 0.0, 0.4 0.0 0.0, 1.0 0.0 0.0, 
    1.0 0.0 0.0, 0.4 0.0 0.0, 1.0 0.0 0.0, 
    1.0 0.0 0.0, 0.4 0.0 0.0, 1.0 0.0 0.0, 
    1.0 0.0 0.0, 
  ]
}

ROUTE Forever.fraction_changed TO Rotater.set_fraction
ROUTE Rotater.value_changed TO Gadget.set_rotation
ROUTE Forever.fraction_changed TO Squisher1.set_fraction
ROUTE Forever.fraction_changed TO Squisher2.set_fraction
ROUTE Forever.fraction_changed TO Squisher3.set_fraction
ROUTE Forever.fraction_changed TO Squisher4.set_fraction
ROUTE Squisher1.value_changed TO Squishy1.set_scale
ROUTE Squisher2.value_changed TO Squishy2.set_scale
ROUTE Squisher3.value_changed TO Squishy3.set_scale
ROUTE Squisher4.value_changed TO Squishy4.set_scale
Next in the VRML Technique column
VRML 2.0’s animation features enable you to make your worlds come alive with animating shapes whose position, orientation, and size change as time progresses. All of the examples in this column use looping animations that repeat forever. In next month’s column I’ll introduce the TouchSensor node type with which you can start and stop animations at the user’s touch.

Resources
- A list of David Nadeau’s VRML Technique columns in NetscapeWorld
- VRML 2.0 browsers NetscapeWorld’s guide to finding and installing a VRML browser on your computer.
- VRML 2.0 glossary
- NetscapeWorld’s VRML vendors chart A handy reference of VRML browser and server companies including their plug-ins to Web browsers -- with updated items in bold to aid your review -- and links to all the vendors.
  http://www.netscapeworld.com/netscapeworld/common/nw.vrmltable.html
- The UTF-8 character set sidebar accompanying the first VRML Technique column.

Specifications
- VRML 2.0 specification http://vag.vrml.org/VRML2.0/FINAL/
- UTF-8 character encoding scheme for UCS http://www.dkuug.dk/JTC1/SC2/WG2/docs/n1335

Sites
- The VRML Repository http://www.sdsc.edu/vrml
- VRML Architecture Group http://vag.vrml.org

About the author
David R. Nadeau is a co-author of The VRML 2.0 Sourcebook, published by John Wiley & Sons and written with Andrea L. Ames and John L. Moreland. David is a staff researcher at the San Diego Supercomputer Center where he is a specialist in 3-D computer graphics, virtual reality, and scientific visualization. He is also the creator of The VRML Repository, a Web site providing extensive information on VRML software, documentation, and 3-D worlds.

⚠️ You can buy David R. Nadeau’s The VRML 2.0 Sourcebook at a 20% discount from Amazon.com Books.
Sensing the viewer’s touch

How to sense the viewer’s touch to start and stop animations in VRML 2.0

By David R. Nadeau

Summary
To enable your virtual worlds to come alive and interact with the viewer, VRML 2.0 provides nodes that sense the viewer’s actions. Using sensor nodes, you can create doors that open and close at the viewer’s knock. Virtual control panels can steer virtual space craft or direct the movements of a virtual robot. Gizmos can whirl to life and virtual creatures scuttle away at the viewer’s touch.

In this month’s VRML Technique column I’ll introduce VRML 2.0’s TouchSensor node type with which you can author worlds that sense the touch of the viewer’s cursor. Along the way I’ll discuss advanced uses of VRML 2.0’s TimeSensor node and show how you can create animations that run periodically, run for a selected number of cycles, or keep track of wall-clock time. (4,500 words)

Table of contents

<table>
<thead>
<tr>
<th>Animating at the viewer’s touch</th>
<th>Experimenting with time sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensing touch</td>
<td>• Counting timer cycles</td>
</tr>
<tr>
<td>• The TouchSensor node type</td>
<td>• Creating periodic animations</td>
</tr>
<tr>
<td>Experimenting with touch sensors</td>
<td>• Creating a stop-watch</td>
</tr>
<tr>
<td>• Triggering animations with cursor proximity</td>
<td>Next in the VRML Technique column</td>
</tr>
<tr>
<td>• Triggering animations using proxy shapes</td>
<td>Resources</td>
</tr>
<tr>
<td>• Triggering animations with mouse button presses</td>
<td>About the author</td>
</tr>
<tr>
<td>• Triggering animations with touch time</td>
<td></td>
</tr>
<tr>
<td>• Creating 3D buttons</td>
<td></td>
</tr>
</tbody>
</table>

Animating at the viewer’s touch

Last month’s column ("Animating shapes") introduced VRML 2.0’s animation features, discussing animation circuits, events, routes, and the TimeSensor, PositionInterpolator, and OrientationInterpolator node types. Using these nodes, you can animate the position, orientation, and scale of shapes. You can create spinning sails on a windmill, orbit planets about a sun, and construct all sorts of virtual mechanical gadgets.
To make your world interactive, you can attach to a shape a sensor that senses viewer actions with a pointing device, such as a mouse. When the viewer clicks on a shape with an attached sensor, the sensor outputs events that can be routed into other nodes to start and stop animations. Using shape sensors you can create shapes that react to the touch of the viewer’s cursor.

Figure 1 shows a sample virtual world containing a robot and control panel. Pressing control panel buttons activate the robot. Click on the image to load the robot world into your VRML 2.0 browser. The caption below the figure gives the size of the world in bytes, and the expected download time.

(See NetscapeWorld’s sidebar on VRML 2.0 Browsers for information on obtaining and installing VRML 2.0 browsers. Also see the NetscapeWorld VRML Vendors chart for a list of VRML browsers and plug-ins, and our glossary of VRML 2.0 terms. You need a VRML browser or plug-in to view the 3D examples presented in this series.)

![A sample VRML 2.0 world containing touch sensitive shapes](image)

Figure 1. A sample VRML 2.0 world containing touch sensitive shapes
(45 kilobytes = 31 seconds @ 14.4bps)
Click on the image to load the world.

**Viewing tip:** Once loaded into your VRML 2.0 browser, if these worlds run a little slowly, try reducing the size of the browser window. A smaller window means there is less screen area for the browser to redraw each time you move in the world. This reduction in drawing area speeds up the browser and enables it to animate more smoothly, or respond more quickly to user actions.

### Sensing touch

Most computers today provide a pointing device to move the cursor on the screen. A mouse with one, two, or three buttons is probably the most common pointing device, but joysticks, trackballs, touchpads, and other such devices are also available. To interact with an application, the viewer moves the cursor about to point at items of interest. When an interesting item is found, the viewer can perform one of three actions:

- **Move:** without pressing a mouse button, move the cursor over an item.
- **Click:** while the cursor is over an item, press the mouse button, then immediately...
Drag: while the cursor is over an item, press the mouse button, move the mouse, then release the button.

In most applications, each of these familiar actions causes something specific to happen. In Microsoft Windows, for instance, movement of the cursor so that it rests on a button causes a message to pop up telling a viewer what will happen if they press the button. In a drawing application, clicking on a shape selects the shape so that its size or color can be changed. Similarly, in a drawing application, a drag action moves a shape across the screen.

In VRML 2.0, you can attach a sensor node to a shape to detect move, click, and drag viewer actions. You can wire the outputs of a sensor node into a circuit to cause shapes to move and animations to play when the viewer interacts with a sensed shape.

The number of buttons available on a pointing device, like a mouse, varies from computer to computer. Macintoshes typically have one-button mice, PCs have two-button mice, and UNIX workstations usually have three-button mice. To insure that VRML 2.0 worlds can be viewed on any type of computer, VRML 2.0 sensors assume there is only a single mouse button available. On a computer with a multiple-button mouse, the left mouse button is usually the button sensed. The remaining mouse buttons, if any, may be used by a VRML browser to select among menu items or steer the viewer as they move through your world.

**The TouchSensor node type**

VRML 2.0’s TouchSensor node detects move, click, and drag actions by the viewer’s pointing device, such as a mouse. The sensor can be included within any group of shapes, such as that managed by a Transform node. When in such a group, a TouchSensor node senses when the viewer’s cursor moves over or clicks on any shape built in that group.

The ability of a TouchSensor node to sense all the shapes in a group enables you to create complex sensed shapes. You can, for instance, build an entire car within a group, then add to the group a TouchSensor node. When the viewer clicks anywhere on the car, the sensor detects the touch and sends events out its outputs. You could use such outputs to control an animation that drives the car about within a virtual city.

<table>
<thead>
<tr>
<th>Syntax: TouchSensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>TouchSensor {</td>
</tr>
<tr>
<td>enabled   TRUE   # exposedField SFBool</td>
</tr>
<tr>
<td>isOver    # eventOut SFBool</td>
</tr>
<tr>
<td>isActive  # eventOut SFBool</td>
</tr>
<tr>
<td>touchTime # eventOut SFBool</td>
</tr>
<tr>
<td>hitPoint_changed # eventOut SFVec3f</td>
</tr>
<tr>
<td>hitNormal_changed # eventOut SFVec3f</td>
</tr>
<tr>
<td>hitTexCoord_changed # eventOut SFVec2f</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

The value of the enabled exposed field turns the sensor on and off. When TRUE, the sensor actively monitors the viewer and generates outputs on one or more of its eventOuts. When FALSE, the sensor is disabled and ignores viewer actions.

When the viewer moves the cursor over a shape sensed by a TouchSensor node, the sensor node outputs a TRUE event using the isOver eventOut. When the viewer moves the cursor off the sensed shape, a FALSE event is output using the isOver eventOut. You can use this TRUE or
FALSE output to cause a shape to highlight, blink, or wiggle whenever the viewer’s cursor is moved over the shape.

When the viewer presses a mouse button while the cursor is over a sensed shape, the sensor node outputs a TRUE event using the isActive eventOut. Later, when the viewer releases the mouse button, a FALSE event is output using the isActive eventOut and the current time is output using the touchTime eventOut. You can use the isActive eventOut to make a 3D button click in and out when the viewer presses it. You can use the touchTime eventOut to start and stop animations at the viewer’s touch.

The remaining three eventOuts of the TouchSensor node, hitPoint_changed, hitNormal_changed, and hitTexCoord_changed, are primarily used along with advanced VRML 2.0 features, such as Script nodes. Discussion of these eventOuts is left to a future column.

---

**Experimenting with touch sensors**

The TouchSensor node type enables you to create virtual control panels with buttons the viewer can press, and shapes that animate in response. Each of the node’s outputs are designed for use in creating a different user interface effect.

**Triggering animations with cursor proximity**

Recall from last month’s VRML Technique column that a TimeSensor node has an enabled field. When this field’s value is FALSE, the timer is silent and outputs no values. If this field’s value is set to TRUE, the timer starts running when the timer’s start time is reached. If you wire a TouchSensor node’s isOver eventOut into the enabled field of a TimeSensor node, you can automatically enable and disable the timer whenever the viewer’s cursor moves over and off a sensed shape.

The VRML text in Figure 2 builds a pair of box shapes forming two spokes on a wheel. Both spokes are built within a Transform node group and sensed by a TouchSensor node included in that group. The TouchSensor node’s isOver eventOut is routed into a TimeSensor node’s enabled field. The TimeSensor node’s output is routed into an OrientationInterpolator node, whose output is routed into the rotation field of the Transform node for the spokes.

When the viewer’s cursor moves over a spoke, the TouchSensor node outputs TRUE using its isOver eventOut. This enables the TimeSensor node and starts the spokes rotating. When the viewer’s cursor moves off a spoke, a FALSE is sent using the TouchSensor node’s isOver eventOut, disabling the TimeSensor and stopping the spoke animation.
Spin while the cursor is over the spokes

```vrml
DEF Spokes Transform {
  # rotation animated
  children [
    DEF Start TouchSensor { }
    Shape {
      appearance DEF SpokeColor Appearance {
        material Material { diffuseColor 1.0 1.0 0.0 }
        geometry Box { size 0.5 4.0 0.5 }
      }
      Shape {
        appearance USE SpokeColor
        geometry Box { size 4.0 0.5 0.5 }
      }
    }
    DEF Clock TimeSensor {
      enabled FALSE
      # enabled set on over
cycleInterval 4.0
      loop TRUE
      startTime 1.0
      stopTime 0.0
    }
    DEF Spinner OrientationInterpolator {
      keyValue [
        0.0 0.0 1.0 0.0,
        0.0 0.0 1.0 -3.14,
        0.0 0.0 1.0 -6.28
      ]
    }
    ROUTE Start.isOver           TO Clock.set_enabled
    ROUTE Clock.fraction_changed TO Spinner.set_fraction
    ROUTE Spinner.value_changed  TO Spokes.set_rotation
  ]
}
```

Figure 2. A pair of wheel spokes that spin when the viewer’s cursor moves over them

Click on the image to load the world.

As you experiment with the example in Figure 2, notice that the animation stops if you move the cursor off a spoke. But what happens if a spoke rotates out from under the cursor?

The `TouchSensor` node only checks if the cursor is over a shape each time the cursor is moved. If the viewer leaves the cursor still and the spokes rotate out from under the cursor, then the
**TouchSensor** node won’t notice the change and won’t send a **FALSE** to stop the animation. Later, if the viewer jiggles the cursor, the **TouchSensor** node will notice the change, check the cursor’s new location, and stop the animation if the cursor is no longer over the sensed shape.

**Triggering animations using proxy shapes**
To avoid problems with a shape animating out from under the cursor, you can use a *proxy shape* instead to control the animation. A proxy shape is an invisible stand-in for a normal shape, sensed but not seen. Like any other shape, a proxy shape is built with a **Shape** node, positioned using a **Transform** node, and can be sensed by a **TouchSensor** node. The only difference is that the proxy shape is typically made invisible by setting the **transparency** field value to 1.0 for the shape’s **Material** node.

Proxy shapes can be used to create touch sensitive invisible areas in your world. For example, the VRML text in Figure 3 uses an invisible box proxy shape to make the rectangular area around the spinning spokes touch sensitive. Movement of the viewer’s cursor over the invisible proxy shape starts the spokes spinning. The proxy shape is stationary, enabling the animation to continue to run even if the spokes rotate out from under the viewer’s cursor. Only movement of the viewer’s cursor off the stationary proxy shape stops the animation.
Figure 3. A pair of wheel spokes that spin when the viewer’s cursor moves over an invisible box proxy shape

Click on the image to load the world.

While building a world, it can be helpful to make proxy shapes partially visible by setting their Material node’s transparency field value to a number between 0.0 (opaque) and 1.0 (fully transparent). Figure 4 shows a partially visible view of the proxy shape used in Figure 3.
Triggering animations with mouse button presses
The isOver eventOut of a TouchSensor node sends TRUE and FALSE values when the viewer’s cursor moves over and off a sensed shape. The isActive eventOut, however, sends TRUE and FALSE values when the viewer presses and releases a mouse button over a sensed shape.

The VRML text in Figure 5 builds a ball that bounces when the viewer presses the mouse button over the shape, and stops bouncing when the mouse button is released. A TouchSensor node senses the ball shape. The sensor’s isActive eventOut is routed into a TimeSensor node’s enabled field, and the TimeSensor node’s output routed into interpolators to bounce the ball, squishing it a bit each time it lands.

When the viewer’s cursor moves over the ball and the mouse button is pressed, the TouchSensor node sends a TRUE using its isActive eventOut, starting the bouncing animation. When the mouse button is released, a FALSE is sent using the isActive eventOut, stopping the animation.
# Ball that bounces while the cursor is over it and the mouse button is pressed

DEF Ball Transform {
    # translation animated
    # scale animated
    children {
        DEF Touch TouchSensor { }
        Shape {
            appearance Appearance {
                material Material { diffuseColor 0.0 0.7 1.0 }
            }
            geometry Sphere { }
        }
    }
}

DEF BounceClock TimeSensor {
    enabled FALSE
    # enabled set on mouse button press
    cycleInterval 1.0
    loop TRUE
    startTime 1.0
    stopTime 0.0
}

DEF BouncePosition PositionInterpolator {
    key [
    # Squish and leap...
    0.0, 0.055, 0.11,
    # Parabolic arc up and down...
    0.22, 0.33, 0.44, 0.55, 0.66, 0.77, 0.88,
    # Land...
    1.0
    ]
    keyValue [
    # Squish and leap...
    0.00 0.00 0.50, # 0.0
    0.00 -0.20 0.00, # 0.055
    0.00 0.00 0.00, # 0.11
    # Parabolic arc up and down...
    0.00 0.35 0.00, # 0.22
    0.00 0.59 0.00, # 0.33
    0.00 0.73 0.00, # 0.44
    0.00 0.78 0.00, # 0.55
    0.00 0.73 0.00, # 0.66
    0.00 0.59 0.00, # 0.77
    0.00 0.35 0.00, # 0.88
    # Land...
    0.00 0.00 0.00, # 1.0
    ]
}

DEF BounceSquish PositionInterpolator {
    key [
    # Squish and leap...
    0.0, 0.055, 0.11,
    # Parabolic arc up and down...
    1.0
    ]
    keyValue [
    # Squish and leap...
    1.0 1.0 1.0, # 0.0
    1.0 0.8 1.1, # 0.055
    1.0 1.0 1.0, # 0.11
    # Parabolic arc up and down...
    1.0
    ]
}

ROUTE Touch.isActive TO BounceClock.set_enabled
ROUTE BounceClock.fraction_changed TO BouncePosition.set_fraction
ROUTE BounceClock.fraction_changed TO BounceSquish.set_fraction
ROUTE BouncePosition.value_changed TO Ball.set_translation
ROUTE BounceSquish.value_changed TO Ball.set_scale

Figure 5. A ball that bounces when the viewer’s mouse button is pressed

Click on the image to load the world.
As you experiment with the bouncing ball, notice that the ball stops its bouncing when you release the mouse button. However, the next time you press the mouse button the ball *does not start where it left off*. Why?

A **TimeSensor** node senses the passage of time even when its **enabled** field value is **FALSE**. While disabled, the sensor continues computing new fractional time values *as if it were enabled*, but doesn’t output them. Later, if the sensor is enabled, the values again flow out of the sensor.

The effect seen by disabling and enabling a **TimeSensor** node is similar to fiddling with the volume knob on your stereo while a cassette tape is playing. Disabling a **TimeSensor** node is like turning down the volume knob; the output is disabled, but the cassette tape continues to play. Enabling a **TimeSensor** node is like turning the volume back up: the output returns to normal, joining the cassette tape’s playback in progress.

The VRML examples in Figures 3 and 5 both use **TouchSensor** node outputs to enable and disable a running **TimeSensor** node. Each time the node is enabled, the viewer joins the animation in progress. This causes a jump in the animated shape’s position as it leaps from its old position to where it should be in the in-progress animation. If this isn’t the effect you want, you can use the **touchTime** eventOut of a **TouchSensor** instead.

**Triggering animations with touch time**
Recall that a **TimeSensor** node has a start time and a stop time. If the start time is set to a time greater than the stop time, then the timer starts running at the start time and continues forever (if the **loop** field value is **TRUE**), or runs only for a single cycle (if the **loop** field value is **FALSE**). If you wire a circuit into a **TimeSensor** node’s **startTime** field, you can set the time at which the sensor starts running, and thereby control the start time of any animation to which the **TimeSensor** node is wired.

The VRML text in Figure 6 builds three boxes that spin 90.0 degrees when animated. To start the animation, the **touchTime** eventOut of a **TouchSensor** node is routed into the **startTime** field of a **TimeSensor** node. The **TimeSensor** node’s output is then routed into three **OrientationInterpolator** nodes which are, in turn, routed into three **Transform** nodes for the three boxes.

When the viewer clicks the mouse button atop the sensed shape, the **TouchSensor** node sends the time at which the shape was touched using its **touchTime** eventOut. The touch time sets the **TimeSensor** node’s start time, and the animation begins. By using a **FALSE** value for the **TimeSensor** node’s **loop** field, the timer runs for a single cycle then stops. When the viewer clicks the mouse button atop the sensed shape again, the **TimeSensor** node starts again, and the animation runs through another cycle.
Figure 6. Three boxes that spin when the viewer clicks on them

Click on the image to load the world.

Creating 3D buttons
Recall that the `stopTime` field of a `TimeSensor` node sets the time at which the timer stops. By wiring an animation circuit into a `TimeSensor` node’s `startTime` and `stopTime` fields, you can start and stop the timer and thereby start and stop any animation controlled by the timer.

The VRML text in Figure 7 builds a simple control panel with two buttons. A `TouchSensor` node senses a green "on" button that, when pressed, sets a `TimeSensor` node’s `startTime` and `stopTime` fields, you can start and stop the timer and thereby start and stop any animation controlled by the timer.

Similarly, a second `TouchSensor` node senses a red "off" button that, when pressed, sets the timer’s stop time.
#VRML V2.0 utf8
#
# Start and stop buttons
#
Transform {
  translation -5.0 1.0 0.0
  children [
    DEF Start TouchSensor { }
    Shape {
      appearance Appearance {
        material Material { diffuseColor 0.0 1.0 0.0 }
      }
      geometry Box { size 2.0 1.0 0.25 }
    }
  ]
}
Transform {
  translation -5.0 -1.0 0.0
  children [
    DEF Stop TouchSensor { }
    Shape {
      appearance Appearance {
        material Material { diffuseColor 1.0 0.0 0.0 }
      }
      geometry Box { size 2.0 1.0 0.25 }
    }
  ]
}
#
# Spinning boxes
#
DEF SpinMe1 Transform {
  children Shape {
    appearance Appearance {
      material Material { diffuseColor 0.0 0.5 1.0 }
    }
    geometry Box { size 4.0 4.0 4.0 }
  }
}
DEF SpinMe2 Transform {
  children Shape {
    appearance Appearance {
      material Material { diffuseColor 1.0 0.0 1.0 }
    }
    geometry Box { size 3.9 3.9 3.9 }
  }
}
DEF SpinMe3 Transform {
  children Shape {
    appearance Appearance {
      material Material { diffuseColor 0.5 0.0 1.0 }
    }
    geometry Box { size 3.8 3.8 3.8 }
  }
}
DEF Clock TimeSensor {
    cycleInterval 2.0
    loop TRUE
    startTime 0.0
    stopTime 1.0
    # start time set on touch
}
DEF Spinner1 OrientationInterpolator {
    key [ 0.0, 1.0 ]
    keyValue [ 0.0 1.0 0.0 0.0, 0.0 1.0 0.0 1.57 ]
}
DEF Spinner2 OrientationInterpolator {
    key [ 0.0, 1.0 ]
    keyValue [ 1.0 0.0 0.0 0.0, 1.0 0.0 0.0 1.57 ]
}
DEF Spinner3 OrientationInterpolator {
    key [ 0.0, 1.0 ]
    keyValue [ 0.0 0.0 1.0 0.0, 0.0 0.0 1.0 1.57 ]
}
ROUTE Start.touchTime TO Clock.set_startTime
ROUTE Stop.touchTime TO Clock.set_stopTime
ROUTE Clock.fraction_changed TO Spinner1.set_fraction
ROUTE Clock.fraction_changed TO Spinner2.set_fraction
ROUTE Clock.fraction_changed TO Spinner3.set_fraction
ROUTE Spinner1.value_changed TO SpinMe1.set_rotation
ROUTE Spinner2.value_changed TO SpinMe2.set_rotation
ROUTE Spinner3.value_changed TO SpinMe3.set_rotation

Figure 7. Three boxes that start spinning when the green button is pressed, and stop when the red button is pressed
Click on the image to load the world.

As you experiment with the example in Figure 7, notice that the animation starts over from the beginning each time you press the green start button. Why doesn’t it continue from where it left off the last time you pressed the red stop button?

Animations in VRML 2.0 are typically controlled by the fractional time output of a TimeSensor node, like that used in Figure 7. Each time the timer starts, its fractional time output resets to 0.0 and the timer begins a new cycle. This behavior of a TimeSensor node insures that starting the sensor always starts an animation from the beginning, not somewhere in the middle.

You can create animation pause buttons, toggle buttons, and a variety of user interface widgets by wiring animation circuits using TouchSensor and TimeSensor nodes and VRML 2.0’s advanced Script node. The Script node enables you to write small program scripts in the Java, JavaScript, or VRMLScript programming languages. By writing your own program scripts you can gain access to advanced VRML 2.0 features and implement pause buttons, toggle buttons, and other behaviors not supported directly by the stock VRML 2.0 nodes. The advanced abilities of the Script node will be discussed in a future column.

Experimenting with time sensors
The TimeSensor node forms the foundation atop which virtually all VRML 2.0 animations are built. Each of the examples in this column, and last month’s, illustrate standard uses of TimeSensor nodes. To create more advanced timing effects, you can wire together multiple TimeSensor nodes in the same circuit.

Counting timer cycles
Each of the examples shown so far create one of two types of animations:

- Infinitely repeating animations that start and stop under viewer control
Animations that run a single cycle then stop automatically

What if you want an animation to run for four cycles then stop? How would you do it?

The **TimeSensor** node does not include a built-in feature to run for a preset number of cycles. However, you can create such a behavior using two **TimeSensor** nodes.

Recall that the **enabled** field of a **TimeSensor** node enables and disables the outputs of the timer. Recall also that the **TimeSensor** node’s **isActive** eventOut sends a **TRUE** event value when the timer starts, and a **FALSE** event value when the timer stops. If you route the **isActive** eventOut of one timer into the **enabled** field of a second timer, then the first timer can start and stop the second timer, and thereby start and stop any animation controlled by the second timer. If you set the cycle interval of the first timer to be an integer multiple of the cycle interval of the second timer, then the first timer acts like a cycle counter that only lets the second timer run for a selected number of cycles.

The VRML text in Figure 8 builds a bouncing blue ball controlled by two **TimeSensor** nodes configured so that the first timer starts and stops the second timer. When the blue ball is touched, the touch time of a **TouchSensor** node sets the start time for both timers. The first timer, configured as a cycle counter, starts immediately and enables the second timer. In response, the second timer starts and begins bouncing the blue ball. When the first timer reaches the end of its cycle, it stops and disables the second timer, which stops the animation. To let the ball bounce four times, the first timer’s cycle interval is set to be four times that of the second timer.

```vrml
#VRML V2.0 utf8
#
# Ball that bounces four times when touched
#
DEF BallTransform {
    # translation animated
    # scale animated
    children [ 
        DEF Touch TouchSensor [ ]
        Shape [ 
            appearance Appearance [ 
                material Material [ diffuseColor 0.0 0.7 1.0 ] 
            ]
            geometry Sphere [ ]
        ]
    ]
}
```
Figure 8. Two timers configured so that one timer starts and stops the other

Click on the image to load the world.
Creating periodic animations

A periodic animation is one that starts, stops, sleeps, then starts over again on a regular basis. The cuckoo movement of a cuckoo clock, for instance, is periodic: it runs once every hour and is dormant in between. You can create periodic animations using two TimeSensor nodes configured in a manner similar to that in Figure 8.

Recall that the cycleTime eventOut of a TimeSensor node sends the current time each time the sensor starts a new cycle. If you route this time into the startTime field of a second TimeSensor node, then the second timer will start each time the first node begins a new cycle.

The VRML text in Figure 9 uses two TimeSensor nodes to create a periodic leap-frog motion for two box shapes. A first timer, named PeriodicTimer, runs through 4.0 second long cycles, repeating indefinitely. The timer’s cycleTime eventOut is routed into a second timer, named LeapFrogTimer, that runs for a single 2.0 second long cycle each time it is started. The second timer controls two interpolators which, in turn, control the position of the two leap-frogging boxes.

Each time the PeriodicTimer starts a new 4.0 second long cycle, its cycleTime eventOut sets the start time of the LeapFrogTimer. That timer starts, runs for 2.0 seconds, causes the boxes to leap frog, then stops. In another 2.0 seconds, the PeriodicTimer finishes another 4.0 second cycle, sends another time out its cycleTime eventOut, and the whole thing repeats.
The animation in Figure 9 runs forever. You could add TouchSensor nodes to start the animation when either of the boxes are touched. To make the animation run only for a chosen number of leap-frogs, then stop, you could add a third TimeSensor node to automatically start and stop the
**PeriodicTimer** sensor by using exactly the same technique illustrated in the VRML text in Figure 8!

**Creating a stop watch**

By using multiple time and touch sensor techniques you can create complex interactive animated shapes. The VRML text in Figure 10 builds a stop watch with these characteristics:

- Hour, minute, and second hands animate continuously, always showing the current time of day in Pacific Standard Time
- A red sweep hand starts and stops when the viewer touches green and red buttons
- Every 15 minutes, a periodic animation puts on a show

To create the hour, minute, and second hand motion, three separate **TimeSensor** nodes tick through 60.0 second (1 minute), 3600.0 second (1 hour), and 43200.0 second (12 hour) cycles. Each hand timer is routed to an **OrientationInterpolator** node to rotate the appropriate hand.

A red stop watch sweep hand uses another **TimeSensor** node and **OrientationInterpolator** node. The timer’s start and stop time values are set by start and stop buttons, each sensed by a **TouchSensor** node.

A periodic animation runs every 15 minutes, controlled by a pair of **TimeSensor** nodes. The first timer controls the animation period, automatically starting the second timer every 900.0 seconds (15 minutes). The second timer controls **OrientationInterpolator** and **PositionInterpolator** nodes to spin and scale the clock.
DEF HourHand Transform {
  translation 0.0 1.5 0.6
  center 0.0 -1.5 0.6
  # animated rotation
  scale 1.0 0.7 1.0
  children [ USE Arm, USE ArrowHead ]
}

DEF SecondHand Transform {
  translation 0.0 1.5 0.6
  center 0.0 -1.5 0.6
  # animated rotation
  scale 0.6 1.0 0.6
  children [ USE Arm, USE ArrowHead ]
}

DEF SweepHand Transform {
  translation 0.0 1.9 0.6
  center 0.0 -1.9 0.6
  # animated rotation
  scale 0.6 1.0 0.6
  children Shape {
    appearance DEF Black Appearance {
      material Material { diffuseColor 1.0 0.0 0.0 }
    }
    geometry Cylinder {
      radius 0.17
      height 3.8
    }
  }
}

# Timers and interpolators to spin hands
# DEF SecondTimer TimeSensor {
#  cycleInterval 60.0  # 60 seconds per sweep
#  loop TRUE
#  startTime 0.0
#  stopTime -1.0
#}
# DEF MinuteTimer TimeSensor {
#  cycleInterval 3600.0  # 60*60 seconds per sweep
#  loop TRUE
#  startTime 0.0
#  stopTime -1.0
#}
# DEF HourTimer TimeSensor {
#  cycleInterval 43200.0  # 60*60*12 seconds per sweep
#  loop TRUE
#  startTime 28800.0  # Adjust for Pacific Standard Time
#  stopTime -1.0
#}
# DEF SecondSpinner OrientationInterpolator {
#  key [ 0.0, 0.5, 1.0 ]
#  keyValue [ 0.0 0.0 1.0 0.0, 0.0 0.0 1.0 -3.14, 0.0 0.0 1.0 -6.28 ]
#}
# DEF MinuteSpinner OrientationInterpolator {
#  key [ 0.0, 0.5, 1.0 ]
#  keyValue [ 0.0 0.0 1.0 0.0, 0.0 0.0 1.0 -3.14, 0.0 0.0 1.0 -6.28 ]
#}
# DEF HourSpinner OrientationInterpolator {
#  key [ 0.0, 0.5, 1.0 ]
#  keyValue [ 0.0 0.0 1.0 0.0, 0.0 0.0 1.0 -3.14, 0.0 0.0 1.0 -6.28 ]
#}
ROUTE SecondTimer.fraction_changed TO SecondSpinner.set_fraction
ROUTE MinuteTimer.fraction_changed TO MinuteSpinner.set_fraction
ROUTE HourTimer.fraction_changed TO HourSpinner.set_fraction
ROUTE SecondSpinner.value_changed TO SecondHand.set_rotation
ROUTE MinuteSpinner.value_changed TO MinuteHand.set_rotation
ROUTE HourSpinner.value_changed TO HourHand.set_rotation

# Timer and interpolators to spin stop watch hand
# DEF SweepTimer TimeSensor {
cycleInterval 60.0      # 60 seconds per sweep
loop TRUE
startTime 0.0
# start time set on start button press
stopTime 1.0
# stop time set on stop button press
}
DEF SweepSpinner OrientationInterpolator {
  key [ 0.0, 0.5, 1.0 ]
  keyValue [ 0.0 0.0 1.0 0.0,  0.0 0.0 1.0 -3.14,  0.0 0.0 1.0 -6.28 ]
}
ROUTE Start.touchTime TO SweepTimer.set_startTime
ROUTE Stop.touchTime  TO SweepTimer.set_stopTime
ROUTE SweepTimer.fraction_changed TO SweepSpinner.set_fraction
ROUTE SweepSpinner.value_changed TO SweepHand.set_rotation
#
# Timers and interpolators for quarter-hour animations
#
DEF QuarterHour TimeSensor {
  cycleInterval 900.0    # 60*15 seconds per action
  loop TRUE
  startTime 28800.0      # PST
  stopTime -1.0
}
DEF QuarterAnimation TimeSensor {
  cycleInterval 3.0
  loop FALSE
  startTime -1.0
  # start time set by quarter-hour clock
  stopTime 0.0
}
DEF QuarterSpinner OrientationInterpolator {
  key [ 0.0, 0.5, 1.0 ]
  keyValue [ 1.0 1.0 0.0 0.0,  1.0 1.0 0.0 -3.14,  1.0 1.0 0.0 -6.28 ]
}
DEF QuarterSquisher PositionInterpolator {
  key [ 0.0, 0.25, 0.5, 0.75, 1.0 ]
  keyValue [ 1.0 1.0 1.0, 0.1 3.0 1.2, 3.0 0.1 1.0, 0.3 2.0 1.2, 1.0 1.0 1.0, ]
}
ROUTE QuarterHour.cycleTime TO QuarterAnimation.set_startTime
ROUTE QuarterAnimation.fraction_changed TO QuarterSpinner.set_fraction
ROUTE QuarterAnimation.fraction_changed TO QuarterSquisher.set_fraction
ROUTE QuarterSpinner.value_changed TO StopWatch.set_rotation
ROUTE QuarterSquisher.value_changed TO StopWatch.set_scale

Figure 10. A stop watch with continuous animation, periodic animation, and animation started and stopped by
the viewer's touch

Click on the image to load the world.

Notice that the stop watch shows the correct time (if you live in the Pacific Standard Time (PST)
time zone)! How is this done?

Recall that a TimeSensor node uses start and stop times measured since 12:00 midnight,
Greenwich Mean Time (GMT), January 1st, 1970. So, a startTime field value of 0.0 starts a timer
at midnight on this date, and a value of 1.0 starts it 1 second later.

To start a timer at a specific time in the history or future of your world, compute the number of
seconds since 12:00 midnight, GMT, January 1st, 1970. There are 3600 seconds in an hour,
86,400 seconds in a day, 31,536,000 seconds in a year of 365 days, and so on.

Times computed in this manner are always in GMT. To convert to another time zone, add or
subtract the appropriate number of hours. Pacific Standard Time (PST), for instance, is eight hours
behind GMT.

The stop watch in Figure 10 uses these time calculations to start the stop watch hands moving at 8 hours after 12:00 midnight, GMT, January 1st, 1970. This insures that the timers are synchronized to PST, 8 hours delayed from GMT.

---

**Next in the VRML Technique column**

This month’s column concludes this series introducing VRML 2.0’s shape-building, animation, and interaction features. Next month I’ll take a look at what’s happening in the VRML industry. I’ll report on the recent WorldMovers and VRML ’97 conferences, highlight a few of the announcements made at those conferences, and provide some perspective on where the industry is going and what might happen next.

---

**Resources**

- A list of David Nadeau’s VRML Technique columns in *NetscapeWorld*
- VRML 2.0 browsers *NetscapeWorld’s* guide to finding and installing a VRML browser on your computer.
- VRML 2.0 glossary
- *NetscapeWorld’s* VRML vendors chart A handy reference of VRML browser and server companies including their plug-ins to Web browsers -- with updated items in bold to aid your review -- and links to all the vendors. [http://www.netsapeworld.com/netsapeworld/common/nw.vrmltable.html](http://www.netsapeworld.com/netsapeworld/common/nw.vrmltable.html)
- The UTF-8 character set sidebar accompanying the first VRML Technique column.

**Specifications**

- VRML 2.0 specification [http://vag.vrml.org/VRML2.0/FINAL/](http://vag.vrml.org/VRML2.0/FINAL/)

**Sites**

- The VRML Repository [http://www.sdsc.edu/vrml](http://www.sdsc.edu/vrml)
- VRML Architecture Group [http://vag.vrml.org](http://vag.vrml.org)

**About the author**

David R. Nadeau is a co-author of *The VRML 2.0 Sourcebook*, published by John Wiley & Sons and written with Andrea L. Ames and John L. Moreland. David is a staff researcher at the San Diego Supercomputer Center where he is a specialist in 3-D computer graphics, virtual reality, and scientific visualization. He is also the creator of *The VRML Repository*, a Web site providing extensive information on VRML software, documentation, and 3-D worlds.

⚠️ You can buy David R. Nadeau’s *The VRML 2.0 Sourcebook* at a 20% discount from Amazon.com Books.
How to view VRML 2.0

Finding and installing the right VRML browser for your computer

By David R. Nadeau

Table of contents

Obtaining a VRML 2.0 browser
Installing a VRML 2.0 browser
Installing DimensionX’s LiquidReality
Installing Intervista’s WorldView
Installing Netscape’s Live3D
Installing Silicon Graphics’ Cosmo Player
Installing Sony’s Community Place

Switching among VRML 2.0 browser plug-ins
Configuring your system

Obtaining a VRML 2.0 browser
DimensionX, Intervista, Newfire, Netscape, Silicon Graphics (SGI), and Sony each provide freely downloadable VRML 2.0 browser plug-ins for use with Netscape Navigator or Microsoft Internet Explorer.

DimensionX, Intervista, Netscape, SGI, and Sony browsers run on PCs with Windows 95. SGI also provides a version of their browser that runs on their Unix workstations. DimensionX provides versions of their browser that work on SGI, Sun, and Linux platforms. The latest version of Netscape Navigator, Netscape Communicator, or Microsoft Internet Explorer is required by most VRML browser plug-ins.

You can obtain information on Netscape Navigator and Communicator from Netscape’s web site at:

http://www.netscape.com

You can obtain information on Microsoft Internet Explorer from Microsoft’s web site at:

http://www.microsoft.com

Installing a VRML 2.0 browser
All VRML 2.0 browsers load and display VRML 2.0 worlds. Browsers differ in their user interfaces, drawing speed, image quality, documentation, and completeness of their VRML 2.0 feature support. To find the browser that’s right for you, download them all and try them out.

Installing DimensionX’s LiquidReality
Unlike other VRML 2.0 browsers, DimensionX’s LiquidReality VRML 2.0 browser is written primarily in Java and runs as a Java applet. You can download the LiquidReality Java applet and support files from DimensionX’s Web site at:
LiquidReality is currently available in a 1.0 beta release for PCs running Windows 95, Windows NT 4.0, and Linux, as well as Sun workstations running Solaris, and SGI workstations running IRIX.

*Note:* The beta release of LiquidReality does not yet support all VRML 2.0 features. See the product’s release notes for feature support details.

Detailed installation instructions for a variety of platforms are available at DimensionX’s Web site.

**Installing Intervista’s WorldView**

You can download the WorldView VRML 2.0 browser plug-in from Intervista’s Web site at:

[http://www.intervista.com](http://www.intervista.com)

WorldView is currently available in a beta release for PCs running Windows 95.

*Note:* The beta release of WorldView does not yet support all VRML 2.0 features. See the product’s release notes for feature support details.

To install WorldView, download the release from Intervista’s Web site, then double-click on the file to run the installation wizard to walk you through the rest of the installation procedure.

Intervista’s WorldView installation automatically installs the browser, plus Intel’s RSX II software. RSX provides advanced sound playback features used by WorldView to enhance the realism of sounds played within VRML 2.0 worlds.

Once installed, WorldView acts as a plug-in for Netscape Navigator. To load a VRML 2.0 world, open the world’s file or URL within Navigator. The WorldView plug-in is automatically invoked and the VRML world displayed within the Navigator window.

**Installing Netscape’s Live3D**

Netscape’s Communicator preview release includes the Live3D 2.0 browser plug-in. Unlike the prior Live3D 1.0, the new version supports VRML 2.0. You can download the Netscape Communicator preview release from Netscape’s Web site at:

[http://www.netscape.com](http://www.netscape.com)

Live3D is currently available in a beta release for PCs running Windows 95.

*Note:* The beta release of Live3D does not yet support all VRML 2.0 features. See the product’s release notes for feature support details.

To install Netscape Communicator, download the release from Netscape’s Web site, then double-click on the file to run the installation wizard to walk you through the rest of the installation procedure.

**Installing Silicon Graphics’ Cosmo Player**

You can download the Cosmo Player VRML 2.0 browser plug-in from Silicon Graphics’ (SGI’s)
Cosmo Player is currently available in a beta release for PCs running Windows 95 or Windows NT. Cosmo Player is also available in a full release for SGI workstations running IRIX 5.3 or IRIX 6.2.

*Note:* The beta and full releases of Cosmo Player do not yet support all VRML 2.0 features. See the product’s release notes for feature support details.

To install Cosmo Player on a PC, download the release from SGI’s Web site, then double-click on the file to run the installation wizard to walk you through the rest of the installation procedure.

SGI’s Cosmo Player PC installation automatically installs the browser, plus Intel’s RSX II software. RSX provides advanced sound playback features used by Cosmo Player to enhance the realism of sounds played within VRML 2.0 worlds.

To install Cosmo Player on an SGI Unix workstation, download the release tar archives from SGI’s Web site. SGI also recommends that you download and install 18 Mbytes of operating system patches. Using `tar`, extract the distribution files from the downloaded tar archive, then install the distribution using `inst` or `swmgr`. You will need the root password to install the files.

Once installed, Cosmo Player acts as a plug-in for Netscape Navigator. To load a VRML 2.0 world, open the world’s file or URL within Navigator. The Cosmo Player plug-in is automatically invoked and the VRML world displayed within the Navigator window.

**Installing Sony’s Community Place**

You can download the Community Place VRML 2.0 browser plug-in and helper application from Sony’s Web site at:

[http://vs.spiw.com/vs](http://vs.spiw.com/vs)

Community Place is currently available in a full 1.0 release for PCs running Windows 95 or Windows NT.

*Note:* The full release of Community Place does not yet support all VRML 2.0 features. See the product’s release notes for feature support details.

Sony provides two versions of Community Place: one that acts as a helper-application, and one that acts as a plug-in for Netscape Navigator 3.0. Both versions provide identical functionality. Most users will probably find that the plug-in version is more convenient since it can display VRML 2.0 worlds directly within the Navigator window. The helper-application version instead uses a separate application window for the display of VRML 2.0 worlds.

To install the Community Place helper-application or plug-in, download the release file then double-click the downloaded file to run a ZIP file self-extractor that extracts the distribution into a temporary folder of your choosing. Once extracted, double-click on `Setup.exe` to run the installation wizard to walk you through the rest of the installation procedure.

Once installed, the Community Place helper-application acts as a slave application for Netscape
Navigator. To load a VRML 2.0 world, open the world’s file or URL within Navigator. The Community Place helper-application is automatically invoked and the VRML world displayed in a separate application window.

The Community Place plug-in works as a plug-in for Netscape Navigator. To load a VRML 2.0 world, open the world’s file or URL within Navigator. The Community Place plug-in is automatically invoked and the VRML world displayed within the Navigator window.

Switching among VRML 2.0 browser plug-ins
Each time a VRML world is loaded into Netscape Navigator, the application looks for a VRML plug-in to display the world. If you have more than one VRML plug-in installed, only the first plug-in found by Navigator is used. If you want to install multiple VRML plug-ins and switch among them, you will need to trick Navigator into loading the one you want.

On a PC, Netscape Navigator plug-ins are stored as DLL (Dynamically Loaded Library) files in a plugins folder within the application’s folder. You can view your current set of plug-ins by following these steps:

- **Open the plugins folder for Netscape Navigator**

  Open the Netscape application folder on your hard disk. This is often found in your Program Files folder. Within the Netscape folder, open the Navigator folder (or Navigator Gold), then the Program folder, and finally the plugins folder.

- **Show all hidden files**

  Using the View menu on any file and folder window, select Options to bring up an options window. On the window’s View tab, click on Show all files, then click OK to close the window.
By showing all files, you reveal the hidden plug-in DLL files in the **plugins** folder.

If you’ve installed all of the VRML 2.0 plug-ins from Intervista, Netscape, SGI, and Sony, you should have a DLL file from each one in your **plugins** folder.

The table below shows the names of several VRML plug-in DLL files.

<table>
<thead>
<tr>
<th>DLL file</th>
<th>Plug-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>npcosmop.dll</td>
<td>SGI Cosmo Player</td>
</tr>
<tr>
<td>np13d32.dll</td>
<td>Netscape Live3D</td>
</tr>
<tr>
<td>npvscp.dll</td>
<td>Sony Community Place</td>
</tr>
<tr>
<td>npWorldView.dll</td>
<td>Intervista WorldView</td>
</tr>
</tbody>
</table>

When Netscape Navigator looks for a VRML plug-in to display a VRML world, it selects the first plug-in *alphabetically*. So, if you have all of the above plug-ins installed, SGI’s Cosmo Player will always be selected. The plug-ins from Netscape, Sony, and Intervista will be ignored.
You can trick Navigator into selecting one of the other VRML plug-ins by one of three methods:

- **Drag unwanted plug-ins to another folder**

  For each plug-in you *don’t want*, move the plug-in’s DLL file to another folder. For instance, you can create an *unplugged* folder within the Netscape Navigator *Program* folder. If you don’t want the Live3D plug-in loaded, drag its DLL file out of the *plugins* folder and into the *unplugged* folder:

  ![Unplugged folder](image)

- **Add an underscore to names of unwanted plug-ins**

  For each plug-in you *don’t want*, add an underscore ( _ ) to the front of the plug-in’s DLL file name. Leave unchanged the name of the VRML plug-in you do want.

  The added underscore in the names of unwanted plug-in DLL files changes the alphabetic sort order of the plug-ins. Plug-ins without underscores are sorted to the top of the list and are chosen by Navigator in preference to those with underscores. For instance, if you don’t want Cosmo Player, Live3D, or WorldView to load (leaving only Community Place), add underscores to their DLL file names.

  ![Plugins folder](image)

- **Use Sony’s plug-in chooser to deactivate unwanted plug-ins**

  Sony provides a plug-in chooser application which you can download from Sony’s Web site. To install the application, download the file *npc10.zip* (112 Kbytes) and unzip it into a new application folder. Double-click on the file *NpChooser.exe* to start the application and bring up a plug-in chooser window.

  The chooser window has an upper area that provides a scrolling list of content-types (also known as MIME types). If you select one of the content types, the lower part of the chooser window displays lists of active and inactive plug-ins for that content type. Clicking on the name of an active or inactive plug-in displays information about that plug-in in the area to the right.
To enable or disable VRML plug-ins, scroll through the content type list and look for one or more entries with VRML’s type code: `x-world/x-vrml`. Click on the content type to display active and inactive plug-ins in the lower part of the chooser window.

For each active plug-in you don’t want, click on the plug-in’s name in the active list, then click on the red down-arrow to slide the plug-in to the inactive list. The chooser application automatically adds an underscore ( `_` ) to the plug-in’s DLL name. Similarly, to activate an inactive plug-in, click on the plug-in’s name in the inactive list, then click on the red up-arrow to slide the plug-in to the active list.

In the future, choosing among different VRML plug-ins will probably be easier. A plug-in chooser like Sony’s is expected to be integrated into the next version of Netscape Navigator.

### Configuring your system

Once you have a VRML 2.0 browser installed, you should also adjust your screen settings to use 16-bit colors (also called High Color or 65535 colors). Do not use 8-bit colors (also called 256 colors), 24-bit colors or 32-bit colors (also called True colors). 8-bit colors give you too few colors to achieve smooth realistic shading when VRML worlds are drawn. 24-bit and 32-bit colors give you plenty of colors for shading, but the added colors require extra processing in your VRML 2.0 browser. That extra processing can significantly slow down the browser, reducing its interactivity.

On a PC running Windows 95 or Windows NT, open your Display control panel and select the Settings tab. Adjust the Color palette menu to select 65535 Colors (16-bit colors). Finally, click the OK button. On some systems, you may be prompted to restart your computer to make the changes take effect.
The UTF-8 character set

VRML 2.0’s international character set

By David R. Nadeau

To enable VRML 2.0 browsers to display any character in any of the world’s languages, VRML 2.0 uses the **UTF-8 Character Set Encoding** defined by the International Standards Organization (ISO) in the ISO 10646-1:1993 specification and the specification’s pDAM 1-5 extension. VRML’s use of this character set standard enables you to use VRML features to build shapes for any English alphabet character, as well as characters in Japanese, Arabic, Cyrillic, and other languages.

**UTF-8** is short for "**UCS Transformation Format 8**," and **UCS** is short for "**Universal Character Set**." Putting these together, UTF-8 is a computer-encoding scheme (transformation format) for storing characters in a file. The "8" in the UTF-8 name indicates that the basic unit of encoding is an 8-bit byte.

The UTF-8 character set encoding includes, as a subset, all of the characters found in the ASCII character set used by most computers. So, to put an "A" in a VRML 2.0 file, just type an "A." International characters not found on the standard computer keyboard may be entered by typing in their UTF-8 codes. This requires special features in your text editor or in a VRML world-building application.

*Note:* For maximum portability of your VRML worlds, restrict your use of UTF-8 characters to only those found on the computer keyboard. There are over 24,000 characters defined in the ISO 10646-1:1993 standard, but only 127 in the ASCII character set. Many VRML browsers will not support the full range of characters theoretically available within VRML 2.0. Additionally, because the UTF-8 encoding requires the use of 8-bit characters, instead of the more common 7-bit ASCII characters, many text editing applications will be unable to create UTF-8 characters or display them properly.

Feedback: nweditors@netscapeworld.com
URL: http://www.netscapeworld.com/netscapeworld/nw-12-1996/sidebars/utf8.html
Last updated: Wednesday, February 19, 1997
VRML 2.0 glossary

The key terms you need to know to get started with VRML

By David R. Nadeau

Appearance
A description of the coloration of a shape. Appearance is described by an Appearance node type. [see Appearance node type, material, Material node type, and shape]

Appearance node type
A node type used to describe the coloration of a shape. Node fields specify the shape material, texture, and texture transform (position, orientation, and scaling of the texture). [see appearance, material, and Material node type]

Axis
An imaginary line establishing a direction in 3-D space. Three axes, labeled X, Y, and Z, are typically used to indicate three orthogonal directions for a coordinate system. A rotation axis is used when specifying an orientation for a coordinate system. [see coordinate system and rotation axis]

Box node type
A geometry node type that builds a 3-D box or cube. A node field specifies the box width, height, and depth. [see geometry and Shape node type]

Child coordinate system
A coordinate system built within the child list of a parent coordinate system. As the parent coordinate system moves, orients, or scales, so does the child coordinate system. [see coordinate system, parent coordinate system, rotation, scaling, and translation]

Click
A press, and immediate release of a pointing device button (such as a mouse button), without movement of the cursor. [see drag, move, pointing device, and touch sensor node type]

Comment
An arbitrary note included in a VRML file. A comment begins with a number-sign (#) and extends to the line end. Comments are skipped by VRML browsers.

Cone node type
A geometry node type that builds a 3-D, upright cone. Node fields specify the cone height and bottom radius. [see geometry and Shape node type]

Coordinate system
A center point and set of orthogonal reference axes used as a reference for measuring distances and shape sizes. In a 3-D coordinate system, the axes are labeled X (side-to-side), Y (up-and-down), and Z (front-to-back). The center point at which the three axes cross is coordinate system origin. A new coordinate system is created by the Transform node type.
Cycle interval
The length of time, measured in seconds, that a TimeSensor node requires to vary its fractional time output from 0.0 to 1.0. [see fractional time, time, and TimeSensor node type]

Cylinder node type
A geometry node type that builds a 3-D, upright cylinder. Node fields specify the cylinder height and radius. [see geometry and Shape node type]

Data type
A description of a type of data, including floating-point numbers, integers, text strings, colors, and more. Every field, exposed field, eventIn, and eventOut of a node has a data type. Each event sent between eventOut and eventIn has a data type that matches that of the eventOut and eventIn. [see event, eventIn, eventOut, exposed field, field, node, and route]

Defined name
A name given to a node using the DEF syntax. [see node name]

Degrees
A system for measuring angles wherein a full circle is 360.0 degrees. A value in degrees can be converted to radians by this formula: radians = degrees * 3.142 / 180.0. [see radians and rotation angle]

Diffuse color
The basic color of a shape, resulting from the random scattering of light that falls on the shape. A diffuse color is specified in the diffuseColor field of a Material node. [see Material node type and RGB color]

Drag
A press of a pointing device button (such as a mouse button) followed by movement of the cursor and a later release of the button. [see click, move, pointing device, and touch sensor node type]

Emissive color
A glow color for a shape, resulting from the shape’s own emission of light. An emissive color is specified in the emissiveColor field of a Material node. [see Material node type and RGB color]

Event
A message sent from one node to another along an animation circuit route. Every event contains a value, with a data type, and a time-stamp. [see data type, eventIn, eventOut, route, and time-stamp]

EventIn
An input to a node used when wiring a route for an animation circuit. An eventIn has a name and a data type. When wiring an animation circuit route, the data type of the eventIn and eventOut on either end of the route must match. [see data type, event, eventOut, exposed field, field, node, and route]
EventOut
An output from a node used when wiring a route for an animation circuit. An eventOut has a name and a data type. When wiring an animation circuit route, the data type of the eventIn and eventOut on either end of the route must match. [see data type, event, eventIn, exposed field, field, node, and route]

Exposed field
A combination of a field, an eventIn, and an eventOut for a node. An exposed field has a field name, a field data type, and a value. For an exposed field named xyz, the associated eventIn and eventOut are named set_xyz and xyz_changed, respectively. [see data type, field, field value, eventIn, eventOut, and node]

Field
A node parameter that provides a shape dimension, color, or other form of node attribute. A field has a field name, a field data type, and a value. [see data type, exposed field, field value, and node]

Field value
A value, such as a number, given to a node’s field to specify a shape dimension or other node attribute. [see field and data type]

Fractional time
Fractional time is an abstract notion of time that indicates the start of an event with a fractional value of 0.0, and the end of the event with a value of 1.0. Intermediate fractional time values are computed as needed so that half-way through the event, the fractional time is 0.5, three-quarters through has a fractional time of 0.75, and so on. The TimeSensor node computes fractional times and binds their starting and ending values to selected start and stop times. If the time between start and stop times is 10 seconds, for example, then fractional time values will vary from 0.0 to 1.0, but take 10 seconds. If this interval is increased to 100 seconds, then fractional time values will still vary from 0.0 to 1.0, but now take 100 seconds to do so. Fractional times are typically used to control animations described by interpolator nodes. [see interpolator, time and TimeSensor node type]

Geometry
A description of the form, or structure of a shape. Geometry may be described by any of several geometry node types, including Box, Cone, Cylinder, and Sphere node types. [see Box node type, Cone node type, Cylinder node type, Sphere node type, shape, and Shape node type]

Instance
A repeated use of a node previously given a defined name. An instance of a node shares the same node type, fields, and field values as the original node given the defined name. A node is named by preceding the node type with DEF myName. A node is instanced by typing USE myName anywhere a node value can be used. [see node name and original]

Interpolator
A node that computes position, orientation, scale, and other types of animation values based upon a list of key values. Computed values are calculated by linearly interpolating between the key values. [see OrientationInterpolator node type and PositionInterpolator node type]

Material
A description of the overall color and transparency of a shape. Material is described by a Material node type. [see appearance, diffuse color, and emissive color]

**Material node type**
A node type that specifies a set of colors used to shade a shape. Node fields describe the diffuse color, emissive color, specular color, ambient intensity, and transparency of a shading material. [see appearance, diffuse color, and emissive color]

**Move**
Movement of a pointing device (such as a mouse) without a button held down. [see click, drag, pointing device, and touch sensor node type]

**Node**
A basic building-block used in VRML 2.0 world-building instructions. A VRML 2.0 file always has at least one node in it, and often contains hundreds or even thousands of nodes. Individual nodes build shapes, describe appearance, control animation, etc. Every node has a node type. The fields and exposed fields of a node are enclosed within curly-braces. [see node type]

**Node name**
A name given to a node so that the node may be repeatedly used (instanced) elsewhere within the same VRML file. Node names may be any sequence of letters and numbers, but may not start with a number or contain most punctuation characters. [see instance and original]

**Node type**
A description of a variety of node, including a node type name and a list of zero or more fields, exposed fields, eventIns, and eventOuts. VRML 2.0 supports 50+ built-in node types. Typical node types include those to build shapes, specify geometry, select appearance, choose sounds, and so on. [see eventIn, eventOut, exposed field, field, and node]

**OrientationInterpolator node type**
An interpolator node that computes rotation axis and angle values based upon a list of key values and fractional times. The rotation output of the interpolator is often routed into the rotation input of a Transform node. [see PositionInterpolator node type and Transform node type]

**Origin**
The center of a coordinate system; the point where the X, Y, and Z axes cross. [see axis and coordinate system]

**Original**
A node given a defined name so that it may be repeatedly used (instanced) later in the same VRML file. The original node’s node type, fields, and field values are re-used each time the node is instanced. A node is named by preceding the node type with DEF myName. A node is instanced by typing USE myName anywhere a node value can be used. [see instance and node name]

**Parent coordinate system**
A coordinate system with one or more shapes or coordinate systems built within it. Such child coordinate systems move, orient, and scale along with the parent coordinate system.
Pointing device
A device to enable the user to move a cursor about on the screen and perform move, click, and drag operations. Most computers use a mouse pointing device, but joysticks, trackballs, trackpads, and similar devices are equally usable. The user’s pointing device can be sensed by a TouchSensor node. [see click, drag, move, and touch sensor node type]

PositionInterpolator node type
An interpolator node that computes 3D positions, translations, or 3D scaling factors based upon a list of key values and fractional times. The output of the interpolator is often routed into the translation or scale inputs of a Transform node. [see OrientationInterpolator node type and Transform node type]

Radians
A system for measuring angles wherein a full circle is 2 PI = 6.28 radians. The system of measuring angles in radians is common in mathematics and science, though use of degrees is more common outside these fields. Angles measured in radians are used to specify rotation angles. A value in radians can be converted to degrees by this formula: degrees = radians * 180.0 / 3.142. [see degrees and rotation angle]

RGB color
A triple of floating-point numbers that specify the amount of red, green, and blue light to be mixed together to form a desired color. Each red, green, or blue amount is given as a value between 0.0 (none) and 1.0 (lots). RGB colors are used to specify colors for shape appearance, lighting, and more. [see appearance]

Rotation
Orientation of a coordinate system by spinning it about an axis by an angle. Rotation is controlled by the Transform node type. [see child coordinate system, parent coordinate system, rotation axis, rotation angle, scaling, translation, and Transform node type]

Rotation angle
An angular measurement used to indicate the amount by which to rotate a coordinate system about a rotation axis. Rotation angles are measured in radians. [see radians, rotation, rotation axis, and Transform node type]

Rotation axis
An imaginary line (vector) about which a coordinate system is turned. One endpoint of the line is always the origin of the coordinate system, while the second endpoint is any 3-D coordinate. [see axis, rotation, rotation angle, and Transform node type]

Route
A connection between an eventOut of one node and an eventIn of another. Routes form the wires of an animation circuit. Event values flow along a route, from eventOut to eventIn. [see event, eventIn, and eventOut]

Scaling
A change in the size of shapes within a coordinate system. Scaling increases or decreases shape size by scaling factors for the X, Y, and Z directions. Scaling is controlled by the Transform node type. [see child coordinate system, parent coordinate system, rotation,
Scaling factor
A positive multiplicative factor used to indicate the degree by which a coordinate system’s shapes are increased or decreased in size. Scaling factors between 0.0 and 1.0 decrease shape size, while those above 1.0 increase shape size. A scaling factor of 1.0 leaves shape size unchanged. [see scaling and Transform node type]

Scene graph
A family tree of coordinate systems and shapes that collectively describe a VRML world. The top-most item in the scene family tree is the world coordinate system. That coordinate system acts as the parent for one or more child coordinate systems and shapes. Those child coordinate systems may, in turn, be parents to further child coordinate systems and shapes. [see child coordinate system, coordinate system, parent coordinate system, shape, and world coordinate system]

Sensor
A node type that senses a change in the environment. Typical sensor nodes sense the passage of time, movement of the user’s cursor, the press of the user’s mouse button, the user’s proximity, collision of the user with a shape, and so forth. [see pointing device, TimeSensor node type, TouchSensor node type]

Shape
A 3-D object in a world, described by its geometry and its appearance. All VRML shapes are built using a Shape node type. [see appearance, geometry, and Shape node type]

Shape node type
A node type that builds a 3-D shape centered at the origin of the parent coordinate system. Node fields specify the geometry and appearance of the shape. [see appearance, coordinate system, geometry, origin, and parent coordinate system]

Sphere node type
A geometry node type that builds a 3-D ball. A node field specifies the ball radius. [see geometry and Shape node type]

Start time
The time at which an animation begins. An animation may be started at a specific time in the history or future of a virtual world. Alternately, animations may be started when a shape is touched, or when some other environment change is sensed. [see fractional time, sensor, stop time, time, TimeSensor node type, and TouchSensor node type]

Stop time
The time at which an animation ends. An animation may be stopped at a specific time in the history or future of a virtual world or allowed to run forever. Animations also may be stopped when a shape is touched, or when some other environment change is sensed. [see fractional time, sensor, start time, time, TimeSensor node type, and TouchSensor node type]

TimeSensor node type
A sensor node type that senses the passage of time. Node fields enable and disable sensor node event outputs, set the start and stop time for those outputs, indicate if the sensor should generate infinitely repeating cyclic outputs, and specify the duration of each cycle. Event
outputs include the current time and fractional times. **TimeSensor** node outputs are frequently routed into one or more interpolator nodes. **TimeSensor** nodes are often started and stopped using **TouchSensor** nodes. [see cycle interval, fractional time, interpolator, route, sensor, start time, stop time, time, and TouchSensor node type]

**Time**
VRML times are measured in seconds measured in seconds since 12:00 midnight, Greenwich Mean Time (GMT), January 1st, 1970. [see fractional time and TimeSensor node type]

**Time-stamp**
A time, measured in seconds, that indicates the moment at which an event was generated and sent along a route. [see event, route, and time]

**TouchSensor**
A sensor node type that senses motion and button presses on the user’s pointing device (such as a mouse). A node field enables and disables the sensor’s outputs. Event outputs include flags indicating when the cursor is over a sensed shape, when a button is pressed, and when a button is released. **TouchSensor** node outputs are often routed into **TimeSensor** nodes. [see pointing device, route, sensor, time, and TimeSensor node type]

**Transform node type**
A node type that creates a new coordinate system in which to build zero or more shapes. The new coordinate system is positioned (translated), oriented (rotated), and resized (scaled) based upon values specified in node fields. [see coordinate system, rotation, scaling, and translation]

**Translation**
Positioning of a coordinate system at a 3-D coordinate relative to the origin of a parent coordinate system. Translation is controlled by the Transform node type. [see child coordinate system, parent coordinate system, rotation, scaling, and Transform node type]

**UTF-8**
An international character set used in VRML 2.0 files. The ASCII characters of a standard computer keyboard form a subset of UTF-8.

**VRML**
An acronym for *Virtual Reality Modeling Language*. VRML is a rich text language for the description of 3-D interactive worlds. The original proposal from Silicon Graphics that led to the development of VRML 2.0 was titled *Moving Worlds*. [see VRML browser and world-builder]

**VRML browser**
A stand-alone helper application or Web browser plug-in that displays VRML worlds. [see world-builder, and the VRML Vendors chart for a list of VRML browsers and plug-ins]

**VRML file header**
The first line of every VRML file. The header line identifies the file as containing VRML content, indicates the version of the language used, and the character set of the file. VRML 1.0 files use the ASCII character set, while VRML 2.0 files use the UTF-8 character set. [see UTF-8]
**World-builder**

An application that enables VRML world authoring within an interactive 3-D drawing environment. [see VRML browser]

**Feedback:** nweditors@netscapeworld.com

**URL:** http://www.netscapeworld.com/netscapeworld/common/nw-vrmlglossary.html

**Last updated:** Wednesday, February 19, 1997