*Eurographics 98*
*Lisbon Portugal*

# *Introduction to VRML 97*

## Lecturer

**David R. Nadeau**

nadeau@sdsc.edu
http://www.sdsc.edu/~nadeau
San Diego Supercomputer Center
University of California at San Diego

## Tutorial notes sections

*Introduction to VRML 97*

# *Abstract*


VRML (the Virtual Reality Modeling Language) is an international 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.

*Introduction to VRML 97*

# *Preface*

Welcome to the *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. Included are 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://vrml.sdsc.edu)

## What's included in these notes

These tutorial notes primarily contain two types of information:

1. General information, such as this preface
2. Tutorial slides and examples

The tutorial slides are arranged as a sequence of 500+ hyper-linked pages containing VRML syntax notes, VRML usage comments, or images of sample VRML worlds. Clicking on a sample world's image loads the VRML world into your browser for you to examine yourself.

Clicking on a sample world's file name, shown underneath the image, loads into your browser a text page showing the VRML code itself. Using these links, or text editor, you can view the VRML code and see how a particular effect is created. 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. It is recommended that you invest in one of the VRML books on the market to get a more thorough coverage of the language. The book we recommend is one we co-authored:

**The VRML 2.0 Sourcebook**
**by Andrea L. Ames, David R. Nadeau, and John L. Moreland**
**published by John Wiley & Sons**

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.

| Version/Released | Comments |
|---|---|
| **VRML 1.0**<br>May 1995 | Begun in late 1994, the first version of VRML was largely based upon the *Open Inventor* file format developed by Silicon Graphics Inc. (SGI). The VRML 1.0 specification was completed in May 1995 and included support for shape building, lighting, and texturing.<br><br>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. |
| **VRML 1.0c**<br>January 1996 | 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. |
| **VRML 1.1**<br>canceled | 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.<br><br>No VRML 1.1 browsers exist. |
| **Moving Worlds**<br>January 1996 | 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. |
| **VRML 2.0**<br>August 1996 | 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.<br><br>While multiple VRML 2.0 browsers exist today, as of this writing, none are *complete*. All of the browsers are missing a few features. Fortunately, most of the missing features are obscure aspects of VRML. |
| **VRML 97** | In early 1997, efforts got under way to present the VRML 2.0 specification |

| | |
|---|---|
| December 1997 | 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 and its explanitory text finalized in September 1997. This specification was ratified by ISO in December 1997.<br><br>Most major VRML 2.0 browsers are now VRML 97 browsers. |

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 these tutorial notes were created

These tutorial notes were developed and tested on a PC with a Diamond Multimedia FireGL 1000 3D accelerator card, and on a Silicon Graphics High Impact UNIX workstation. HTML and VRML text was hand-authored using a text editor. In some cases Perl and C programs were used to automatically generate smooth surfaces and animation paths.

A Perl script, called `mktalk`, developed by John Moreland, was used to process raw tutorial notes text and produce the 500+ individual HTML files, one per tutorial slide.

HTML text was displayed using Netscape Navigator 4.04 on Silicon Graphics and PC systems and Microsoft Internet Explorer 4.01 on 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 24 point font for presentation during the 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.

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 Silicon Graphics Cosmo Player 2.0 for Netscape Navigator and Microsoft Internet Explorer.

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 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

Can you use these tutorial notes for your own purposes? The answer is:



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, please:

1. Give credit for the original material
2. Tell us since we like hearing about the use of the material!

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

## Contact

For bug reports, comments, and questions, please 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
http://www.sdsc.edu/~nadeau

# *Lecturer Information*

**David R. Nadeau**

| | |
|---|---|
| Title | Principal Scientist |
| Affiliation | San Diego Supercomputer Center (SDSC)<br>University of California, San Diego (UCSD) |
| Address | P.O. Box 85608<br>San Diego, CA 92186-9784<br><br>UPS, Fed Ex: 10100 Hopkins Dr.<br>La Jolla, CA 92093-0505 |
| Work phone | (619) 534-5062 |
| Fax phone | (619) 534-5152 |
| Email | nadeau@sdsc.edu |
| Home page | http://www.sdsc.edu/~nadeau |


Dave Nadeau is a principal scientist at the San Diego Supercomputer Center (SDSC), a national research center specializing in computational science and engineering, located on the campus of the University of California, San Diego (UCSD). Specializing in scientific visualization and virtual reality, he is the author of technical papers and articles on 3D graphics and VRML and is a co-author of two books on VRML (*The VRML Sourcebook* and *The VRML 2.0 Sourcebook*, published by John Wiley & Sons). He is the founder and lead librarian for *The VRML Repository* and *The Java3D Repository*, principal Web sites for information on VRML, Java3D, and related software.

Dave has taught VRML at multiple conferences including SIGGRAPH 96-97, WebNet 96-97, VRML 97-98, WMC/SCS 98, Eurographics 97, and Visualization 97. He was a co-chair for the *VRML Behavior Workshop* in October 1995, the first workshop on VRML behavior technology, and a co-chair for the *VRML 95* conference in December 1995, the first conference on VRML. He was on the program committees for VRML 97 and VRML 98 and is SDSC's representative to the VRML Consortium.

Dave holds a B.S. in Aerospace Engineering from the University of Colorado, Boulder, an M.S. in Mechanical Engineering from Purdue University, and is in the PhD program in Electrical and Computer Engineering at the University of California, San Diego.

# *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). Fortunately, most of the features not yet fully supported are fairly obscure.

All VRML examples in these tutorial notes have been extensively tested and are believed to be correct. 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, Cosmo Software's Cosmo Player for PCs, Macs, and Silicon Graphics UNIX workstations is the fastest, most complete, and most robust VRML 97 browser available. It is this browser that was used to test this tutorial's VRML 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 an image of the world is included 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 Cosmo Software (Silicon Graphics), support a derivative of JavaScript called *VRMLscript*. The language is essentially identical to JavaScript. Because of Cosmo Software's 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 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:

```
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 Cosmo Software 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 the examples to JavaScript simply by changing the tag "`vrmlscript:`" to "`javascript:`" like this:

```
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. Perhaps you should 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.

*Introduction to VRML 97*

# *Using the Java examples*

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, these tutorial notes include source Java files, but *not* compiled Java class files. 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 http://www.gamelan.com.

Once you have the VRML Java API class files and a Java compiler, you will need to compile the supplied Java files. Each platform and Java compiler is different. You'll have to consult your software's manuals.

Once compiled, place the `.class` files in the `examples` folder along with the other tutorial examples. 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 Java 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. Perhaps you should consider getting a different brand of 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 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, 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.

Introduction to VRML 97

# *Table of contents*

## Section 4 - Scripts and prototypes

# Welcome!

# *Introduction to VRML 97*

*Welcome to the tutorial!*

Dave Nadeau
San Diego Supercomputer Center
University of California at San Diego

# ***Schedule for the day***

| | |
|---|---|
| **Section 1** | Shapes, geometry, appearance |
| | *Break* |
| **Section 2** | Animation, sensors, geometry |
| | *Lunch* |
| **Section 3** | Textures, lights, environment |
| | *Break* |
| **Section 4** | Scripts, prototypes |

# ***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

# 5

# Introducing VRML

# *What is VRML?*

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

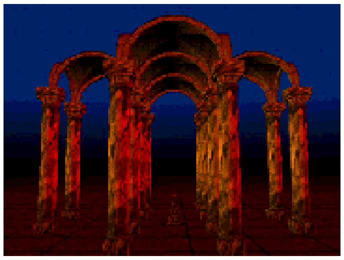
- VRML text files use a **`.wrl`** extension

# *What do I need to use VRML?*

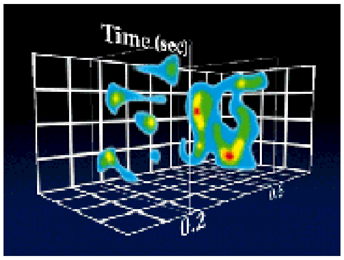
- You can view VRML files using a *VRML browser*:
  - A VRML helper-application
  - A VRML plug-in to an HTML browser

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

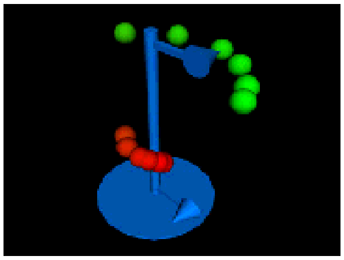
# *Examples*



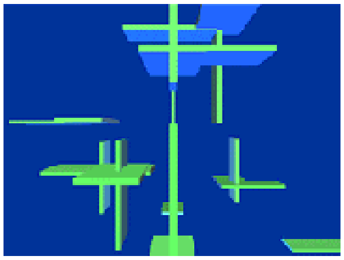
[ temple.wrl ]




[ cutplane.wrl ]



[ spiral.wrl ]



[ floater.wrl ]

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

| | |
|---|---|
| ● Fill Web page | [ boxes.wrl ] |
| ● Embed into Web page | [ boxes1.htm ] |
| ● Fill Web page frame | [ boxes2.htm ] |
| ● Embed into Web page frame | [ boxes3.htm ] |
| ● Embed multiple times | [ boxes4.htm ] |

# *What do I need to develop in VRML?*

- You can construct VRML files using:
  - A text editor
  - A world builder application
  - A 3D modeler and format translator
  - A shape generator (like a Perl script)

# *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 and animating user interface
    - Little need to learn VRML syntax

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

# *Should I use a 3D modeler and format translator?*

- Pros:
    - Very powerful drawing and animating features
    - Can make photo-realistic images too
- Cons:
    - May not support all VRML features
    - May not produce most efficient VRML
    - Not designed for VRML
    - Often a one-way path from 3D modeler into VRML
    - Easy to make shapes that are too complex

# *Should I use a shape generator?*

- Pros:
  - Easy way to generate complex shapes
    - Fractal mountains, logos, etc.
  - Generate VRML from CGI Perl scripts
  - Extend science applications to generate VRML

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

# *How do I get VRML software?*

- The VRML Repository at:

  http://vrml.sdsc.edu

  maintains uptodate information and links for:

| | |
|---|---|
| Browser software | Sound libraries |
| World builder software | Object libraries |
| File translators | Specifications |
| Image editors | Tutorials |
| Java authoring tools | Books |
| Texture libraries | *and more...* |

# 16

# Building a VRML world

# *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. . .

# *A sample VRML file*

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

# *Understanding the header*

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

- **`#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*

```
# A Cylinder
```

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

# ***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

# *Using node type names*

- Node type names are *case sensitive*
  - Each word starts with an upper-case character
  - The rest of the word is lower-case

- Some examples:

```
Appearance          ElevationGrid
Cylinder            FontStyle
Material            ImageTexture
Shape               IndexedFaceSet
```

# *Using fields and values*

```
Cylinder {
    height 2.0
    radius 1.5
}
```

- Fields describe node attributes
- Every field has:
  - A field name (**`height`**, **`radius`**, etc.)
  - A data type (float, integer, etc.)
  - A default value

# *Using field names*

- Field names are *case sensitive*
  - The first word starts with lower-case character
  - Each added word starts with upper-case character
  - The rest of the word is lower-case

- Some examples:

```
appearance          coordIndex
height              diffuseColor
material            fontStyle
radius              textureTransform
```

# *Using fields and values*

- Different node types have different fields

- Fields are optional
  - A default value is used if a field is not given

- Fields can be listed in any order
  - The order doesn't affect the node

# Summary

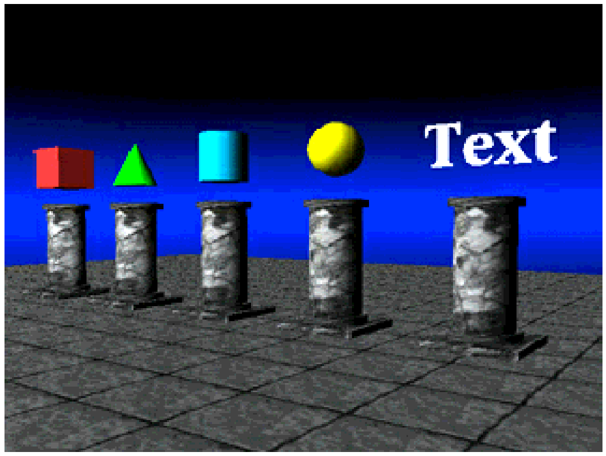
- The file header gives the version and encoding
- Nodes describe scene content
- Fields and values specify node attributes
- Everything is case sensitive

# 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*




[ prim.wrl ]

# *Syntax: Shape*

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

```
Shape {
    appearance . . .
    geometry   . . .
}
```

# *Specifying appearance*

- Shape appearance is described by *appearance* nodes

- For now, we’ll use nodes to create a shaded white appearance:

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

# *Specifying geometry*

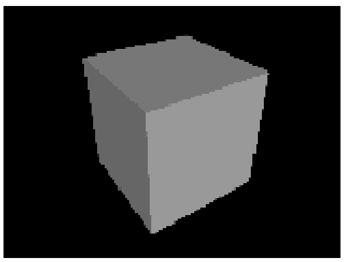
- Shape geometry is built with *geometry* nodes:

```
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
  - **`size`** - width, height, depth

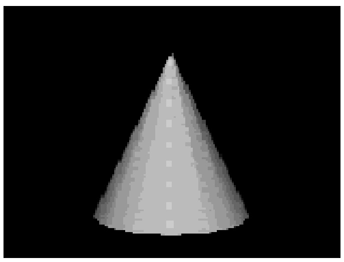


[ box.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Box {
        size 2.0 2.0 2.0
    }
}
```

# *Syntax: Cone*

- A **`Cone`** geometry node builds an upright cone
  - **`height`** and **`bottomRadius`** - cylinder size
  - **`bottom`** and **`side`** - parts on or off

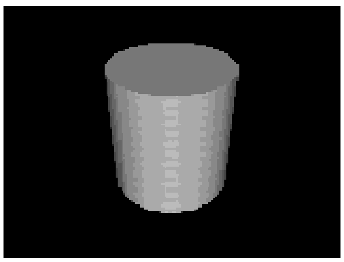


[ cone.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Cone {
        height 2.0
        bottomRadius 1.0
        bottom TRUE
        side TRUE
    }
}
```

# *Syntax: Cylinder*

- A **`Cylinder`** geometry node builds an upright cylinder
  - **`height`** and **`radius`** **-** cylinder size
  - **`bottom`**, **`top`**, and **`side`** **-** parts on or off

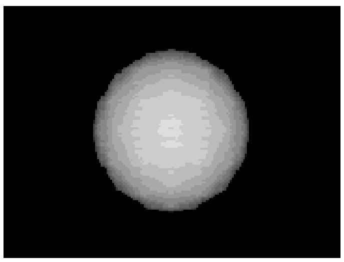


[ cyl.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Cylinder {
        height 2.0
        radius 1.0
        bottom TRUE
        top TRUE
        side TRUE
    }
}
```

# *Syntax: Sphere*

- A **`Sphere`** geometry node builds a sphere
  - **`radius`** - sphere radius

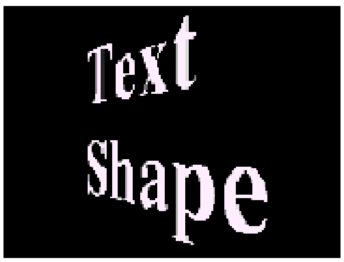


[ sphere.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Sphere {
        radius 1.0
    }
}
```

# *Syntax: Text*

- A **`Text`** geometry node builds text
  - **`string`** **-** text to build
  - **`fontStyle`** **-** font control




[ text.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string [ "Text",
                 "Shape" ]
        fontStyle FontStyle {
            style "BOLD"
        }
    }
}
```

# *Syntax: FontStyle*

- A **`FontStyle`** node describes a font
  - **`family`** **-** **`SERIF`**, **`SANS`**, or **`TYPEWRITER`**
  - **`style`** **-** **`BOLD`**, **`ITALIC`**, **`BOLDITALIC`**, or **`PLAIN`**


[ textfont.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
            family  "SERIF"
            style   "BOLD"
        }
    }
}
```

# *Syntax: FontStyle*

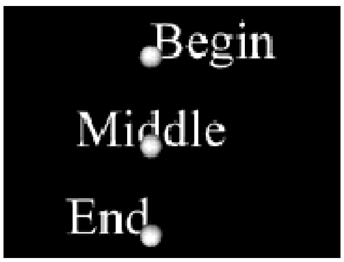
- A **`FontStyle`** node describes a font
  - **`size`** - character height
  - **`spacing`** - row/column spacing


[ textsize.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
            size    1.0
            spacing 1.0
        }
    }
}
```

# *Syntax: FontStyle*

- A **`FontStyle`** node describes a font
  - **`justify`** - **`FIRST`**, **`BEGIN`**, **`MIDDLE`**, or **`END`**

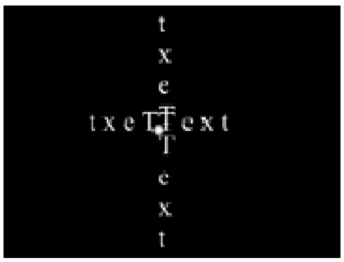



[ textjust.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
            justify "BEGIN"
        }
    }
}
```

# *Syntax: FontStyle*

- A **`FontStyle`** node describes a font
  - **`horizontal`** - horizontal or vertical
  - **`leftToRight`** and **`topToBottom`** - direction



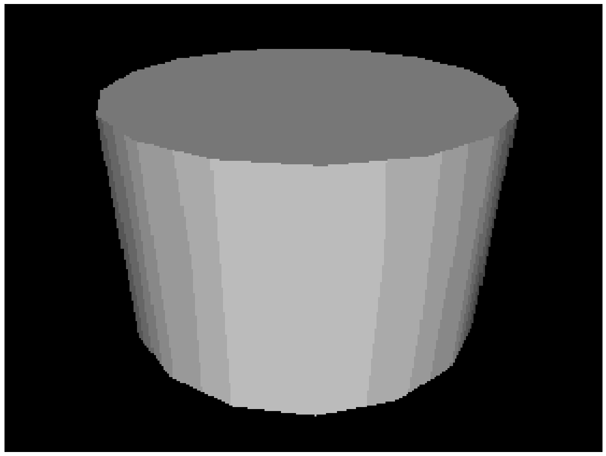
[ textvert.wrl ]

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
            horizontal  FALSE
            leftToRight TRUE
            topToBottom TRUE
        }
    }
}
```

# *Primitive shape example code*

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

# *Primitive shape example*



[ cylinder.wrl ]

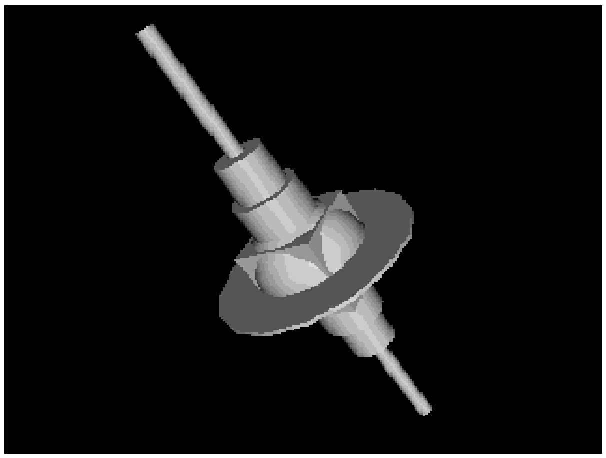
# *Building multiple shapes*

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

# *Multiple shapes file example code*

```
#VRML V2.0 utf8
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Box {
        size 1.0 1.0 1.0
    }
}
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Sphere {
        radius 0.7
    }
}
. . .
```

# *Multiple shapes file example*



[ space.wrl ]

# *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

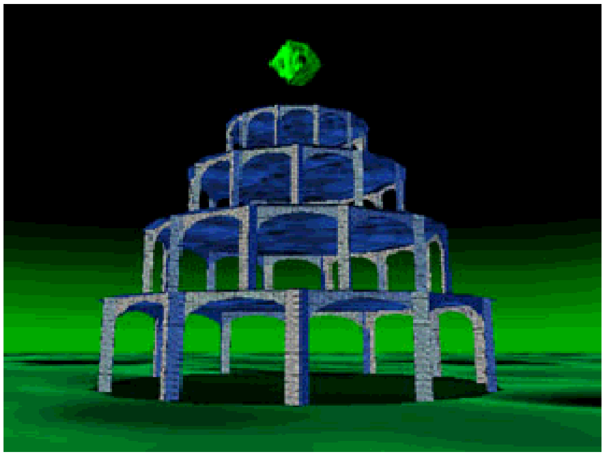
# Transforming shapes

# *Motivation*

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

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

# *Example*

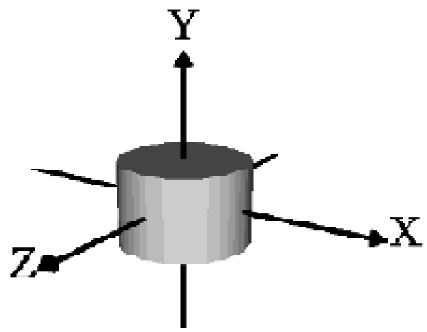


[ towers.wrl ]

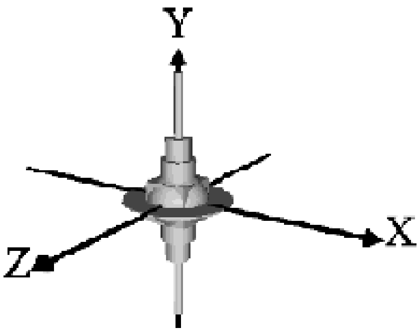
# Using coordinate systems

- 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

# *Visualizing a coordinate system*




a. XYZ axes and a simple shape




b. XYZ axes and a complex shape

# *Transforming a 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

# *Syntax: Transform*

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

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

# *Including children*

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

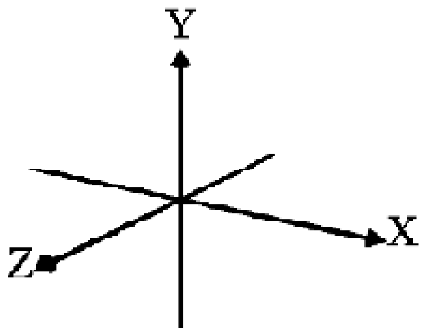
```
Transform {
    . . .
    children [
        Shape { . . . }
        Shape { . . . }
        Transform { . . . }
        . . .
    ]
}
```

# *Translating*

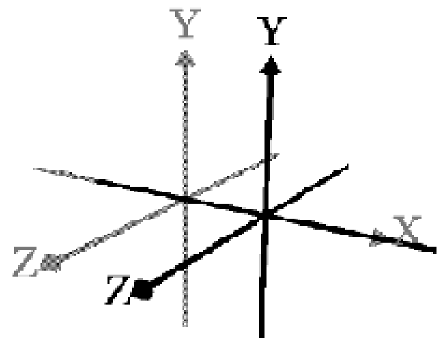
- *Translation* positions a coordinate system in X, Y, and Z

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

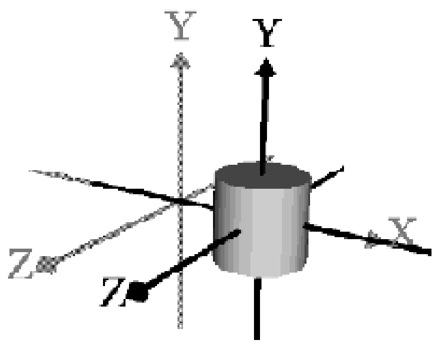
# ***Translating***




a. World coordinate system




b. New coordinate system, translated 2.0 units in X




c. Shape built in new coordinate system

# *Rotating*

- *Rotation* orients a coordinate system about a rotation axis by a rotation angle
  - Angles are measured in *radians*
    - **radians = degrees / 180.0 * 3.141**

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

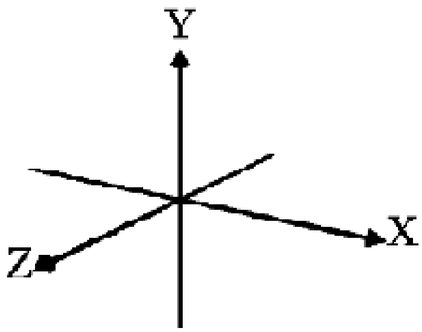
# *Specifying rotation axes*

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

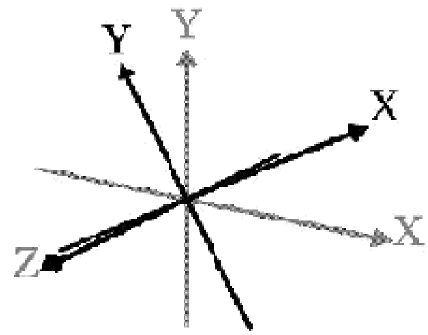
- Typical rotations are about the X, Y, or Z axes:

| **Rotate about** | **Axis** |
|---|---|
| X-Axis | `1.0 0.0 0.0` |
| Y-Axis | `0.0 1.0 0.0` |
| Z-Axis | `0.0 0.0 1.0` |

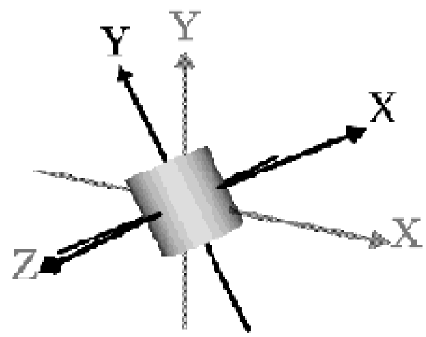
# *Rotating*




a. World coordinate system




b. New coordinate system, rotated 30.0 degrees around Z




c. Shape built in new coordinate system

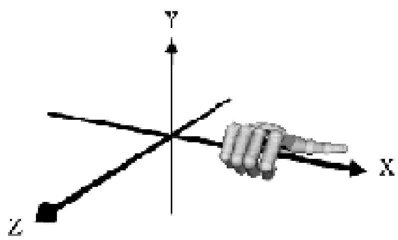
# *Using the Right-Hand Rule*

- Positive rotations are *counter-clockwise*

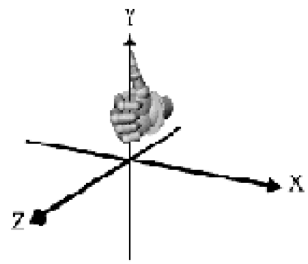
- To help remember positive and negative rotation directions:
  - Open your hand
  - Stick out your thumb
  - Aim your thumb in an axis *positive* direction
  - Curl your fingers around the axis

  - The curl direction is a *positive* rotation

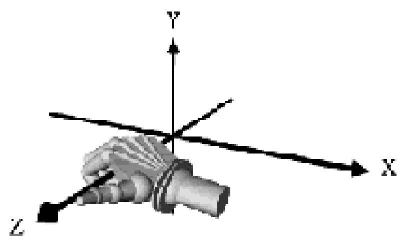
# *Using the Right-Hand Rule*




a. X-axis rotation




b. Y-axis rotation



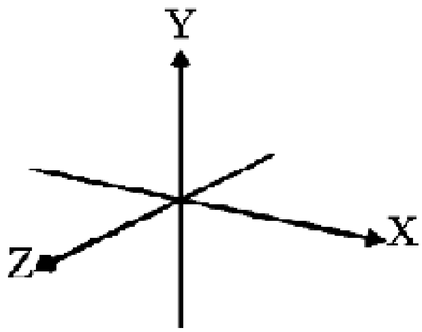

c. Z-axis rotation

# *Scaling*

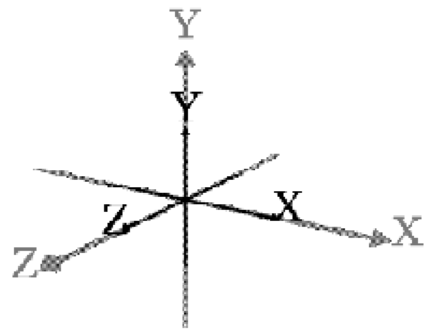
- *Scale* grows or shrinks a coordinate system by a scaling factor in X, Y, and Z

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

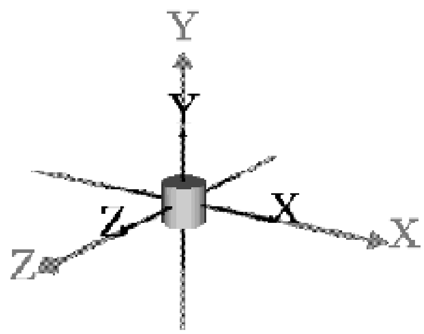
# *Scaling*




a. World coordinate system




b. New coordinate system, scaled by half




c. Shape built in new coordinate system

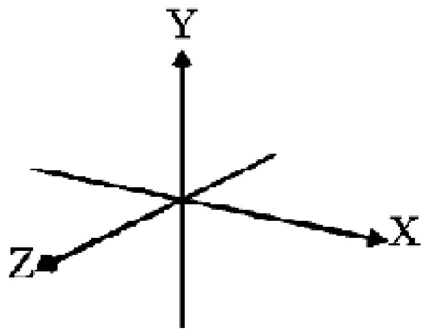
# *Scaling, rotating, and translating*

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

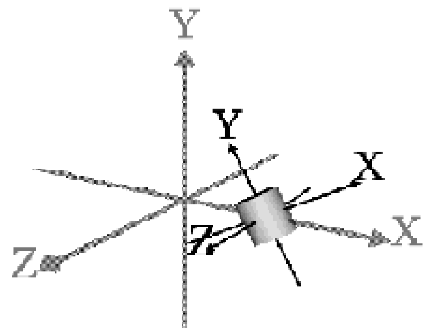
```
Transform {
    translation 2.0 0.0 0.0
    rotation 0.0 0.0 1.0  0.52
    scale 0.5 0.5 0.5
    children [ . . . ]
}
```

- Read operations *bottom-up*:
  - The children are scaled, rotated, then translated
  - Order is fixed, independent of field order

# *Scaling, rotating, and translating*




a. World coordinate system

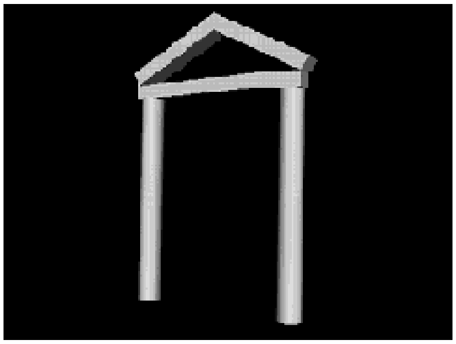



b. New coordinate system,
scaled by half,
rotated 30.0 degrees around Z,
and translated 2.0 units in X

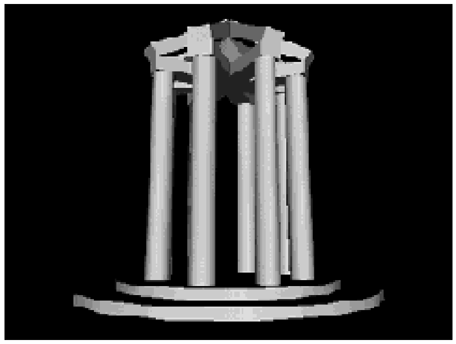
# *Transform group example code*

```
Transform {
    translation -2.0 -1.0 0.0
    children [
        Shape {
            appearance Appearance {
                material Material { }
            }
            geometry Cylinder {
                radius 0.3
                height 6.0
                top FALSE
            }
        }
    ]
}
. . .
```

# *Transform group example*



**[ arch.wrl ]**



**[ arches.wrl ]**

# *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`**

# Controlling appearance with materials

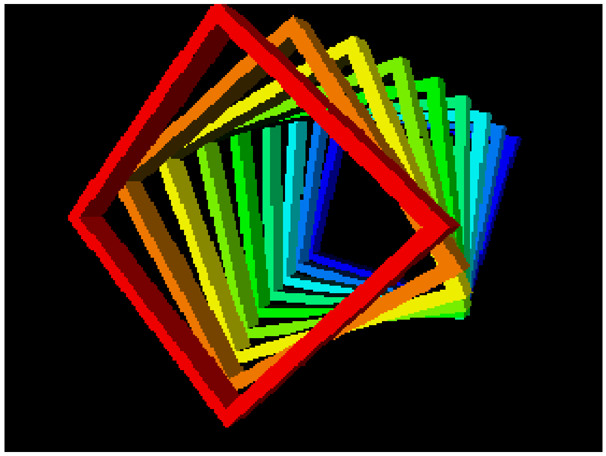
# *Motivation*

- 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 **`Shape`** nodes describe:
  - **`appearance`** - color and texture
  - **`geometry`** - form, or structure

```
Shape {
    appearance . . .
    geometry   . . .
}
```

# *Syntax: Appearance*

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

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

# *Syntax: Material*

- A **`Material`** node controls shape material attributes
  - **`diffuseColor`** - main shading color
  - **`emissiveColor`** - glowing color
  - **`transparency`** - opaque or not

```
Shape {
    appearance Appearance {
        material Material {
            diffuseColor  0.8 0.8 0.8
            emissiveColor 0.0 0.0 0.0
            transparency  0.0
        }
    }
    geometry . . .
}
```

# *Specifying colors*

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

| Color | Red | Green | Blue | Result |
|---|---|---|---|---|
| White | `1.0` | `1.0` | `1.0` | (white) |
| Red | `1.0` | `0.0` | `0.0` | (red) |
| Yellow | `1.0` | `1.0` | `0.0` | (yellow) |
| Cyan | `0.0` | `1.0` | `1.0` | (cyan) |
| Brown | `0.5` | `0.2` | `0.0` | (brown) |

# *Syntax: Material*

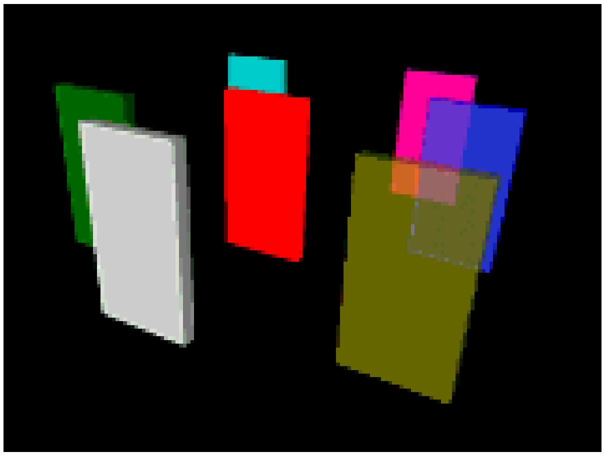
- A **`Material`** node also controls shape shininess
  - **`specularColor`** - highlight color
  - **`shininess`** - highlight size
  - **`ambientIntensity`** - ambient lighting effects

```
Shape {
    appearance Appearance {
        material Material {
            specularColor 0.71 0.70 0.56
            shininess 0.16
            ambientIntensity 0.4
        }
    }
    geometry . . .
}
```

# *Appearance example code*

```
Shape {
    appearance Appearance {
        material Material {
            diffuseColor  0.2 0.2 0.2
            emissiveColor 0.0 0.0 0.8
            transparency  0.25
        }
    }
    geometry Box {
        size 2.0 4.0 0.3
    }
}
. . .
```

# *Appearance example*



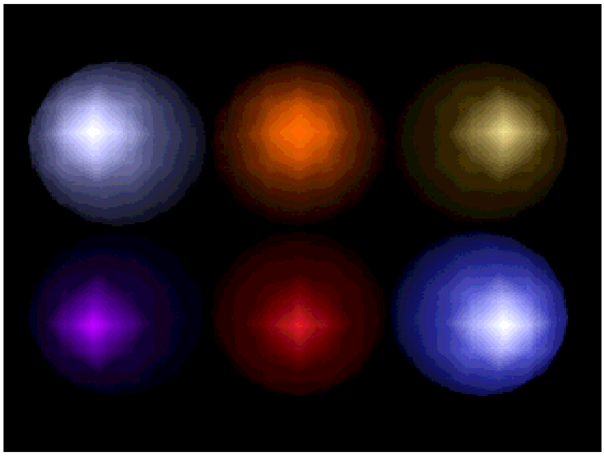
[ slabs.wrl ]

Controlling appearance with materials

# *Experimenting with shiny materials*

| **Description** | **ambient Intensity** | **diffuse Color** | **specular Color** | **shininess** |
|---|---|---|---|---|
| Aluminum | 0.30 | 0.30 0.30 0.50 | 0.70 0.70 0.80 | 0.10 |
| Copper | 0.26 | 0.30 0.11 0.00 | 0.75 0.33 0.00 | 0.08 |
| Gold | 0.40 | 0.22 0.15 0.00 | 0.71 0.70 0.56 | 0.16 |
| Metalic Purple | 0.17 | 0.10 0.03 0.22 | 0.64 0.00 0.98 | 0.20 |
| Metalic Red | 0.15 | 0.27 0.00 0.00 | 0.61 0.13 0.18 | 0.20 |
| Plastic Blue | 0.10 | 0.20 0.20 0.71 | 0.83 0.83 0.83 | 0.12 |

# *Shiny materials example*



[ shiny.wrl ]

# *Summary*

- The `Appearance` node controls overall shape appearance
- The `Material` node controls overall material properties including:
  - Shading color
  - Glow color
  - Transparency
  - Shininess
  - Ambient intensity

# Grouping nodes

# *Motivation*

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

```
Group       { . . . }
Switch      { . . . }
Transform   { . . . }
Billboard   { . . . }
Anchor      { . . . }
Inline      { . . . }
```

# *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
    - Children implicitly numbered from 0
    - A **`-1`** selects no children

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

# *Syntax: Transform*

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

```
Transform {
    translation 0.0 0.0 0.0
    rotation    0.0 1.0 0.0 0.0
    scale       1.0 1.0 1.0
    children [ . . . ]
}
```

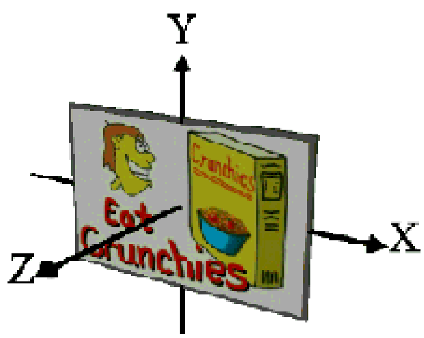
# *Syntax: Billboard*

- 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

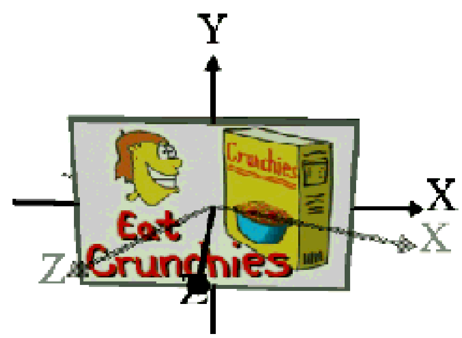
```
Billboard {
    axisOfRotation 0.0 1.0 0.0
    children [ . . . ]
}
```

# *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. Viewer moves to the right




b. Billboard automatically rotates to face viewer

# *Billboard rotation axes*

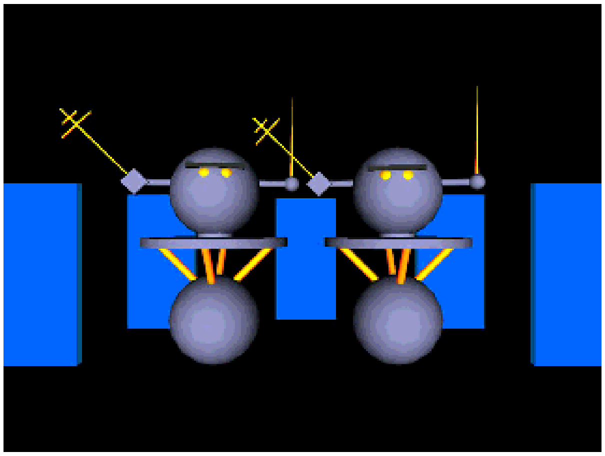
- A rotation axis limits rotation to spin about that axis
- A *zero* rotation axis enables rotation around any axis

| **Rotate about** | **Axis** |
|---|---|
| X-Axis | `1.0 0.0 0.0` |
| Y-Axis | `0.0 1.0 0.0` |
| Z-Axis | `0.0 0.0 1.0` |
| Any Axis | `0.0 0.0 0.0` |

# *Billboard group example code*

```
Billboard {
    # Y-axis
    axisOfRotation 0.0 1.0 0.0
    children [
        Shape { . . . }
        Shape { . . . }
        Shape { . . . }
        . . .
    ]
}
```

# *Billboard group example*



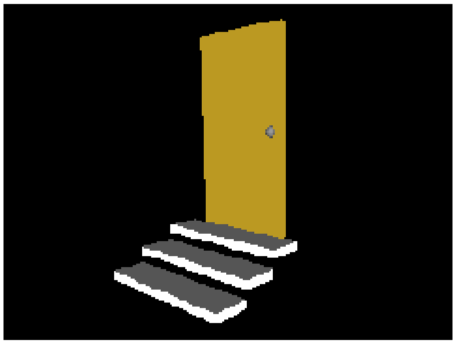
[ robobill.wrl ]

# *Syntax: Anchor*

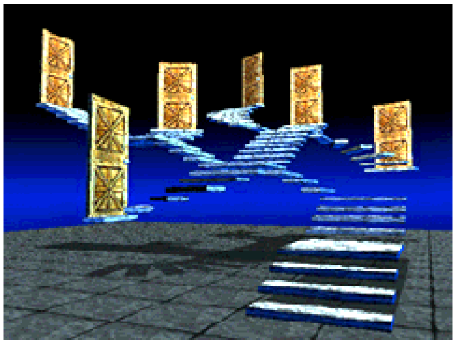
- An **`Anchor`** node creates a group that acts as a clickable anchor
  - *Every child* node in the group is displayed
  - Clicking any child follows a URL
  - A *description* names the anchor

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

# *Anchor example*



[ **anchor.wrl** ]
a. Click on door to go to...



[ **stairwy.wrl** ]
b. ...the stairway world

# *Syntax: Inline*

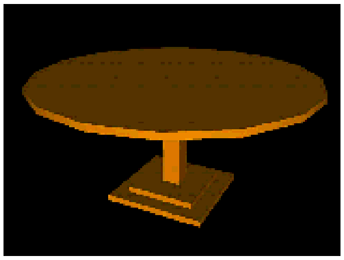
- An **`Inline`** node creates a special group from another VRML file's contents
  - Children read from file selected by a URL
  - *Every child* node in group is displayed

```
Inline {
    url "table.wrl"
}
```

# *Inline example code*

```
Inline { url "table.wrl" }
. . .
Transform {
    translation -0.95 0.0 0.0
    rotation 0.0 1.0 0.0 3.14
    children [
        Inline { url "chair.wrl" }
    ]
}
```

# *Inline example*



[ table.wrl ]

[ chair.wrl ]

[ dinette.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

# Naming nodes

# *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

```
Shape {
    appearance Appearance {
        material DEF RedColor Material {
            diffuseColor 1.0 0.0 0.0
        }
    }
    geometry . . .
}
```

# *Using DEF*

- `DEF` must be in upper-case
- 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

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

# *Using USE*

- **`USE`** must be in upper-case
- 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
  - You can only instance names defined in the same file

# *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...

# *Node names example code*

```
Inline { url "table.wrl" }
Transform {
    translation 0.95 0.0 0.0
    children DEF Chair Inline { url "chair.wrl" }
}
Transform {
    translation -0.95 0.0 0.0
    rotation 0.0 1.0 0.0 3.14
    children USE Chair
}
Transform {
    translation 0.0 0.0 0.95
    rotation 0.0 1.0 0.0 -1.57
    children USE Chair
}
Transform {
    translation 0.0 0.0 -0.95
    rotation 0.0 1.0 0.0 1.57
    children USE Chair
}
```

# *Node names example*

[ dinette.wrl ]

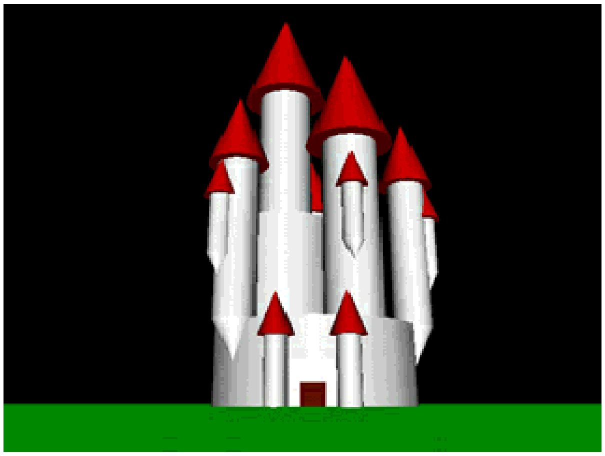
# *Summary*

- **`DEF`** names a node
- **`USE`** uses a named node

# Summary examples

# *A fairy-tale castle*

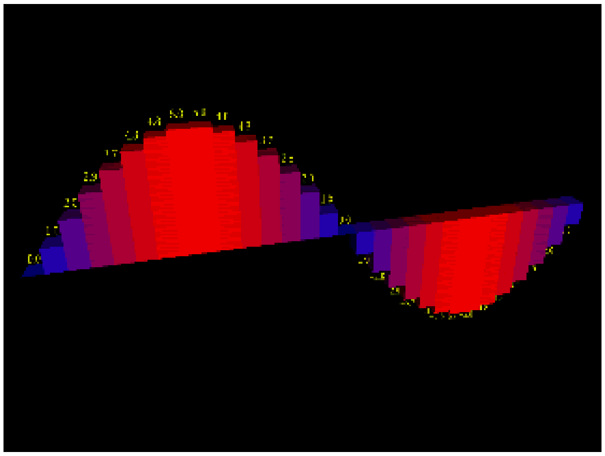
- **`Cylinder`** nodes build the towers
- **`Cone`** nodes build the roofs and tower bottoms



[ castle.wrl ]

# *A bar plot*

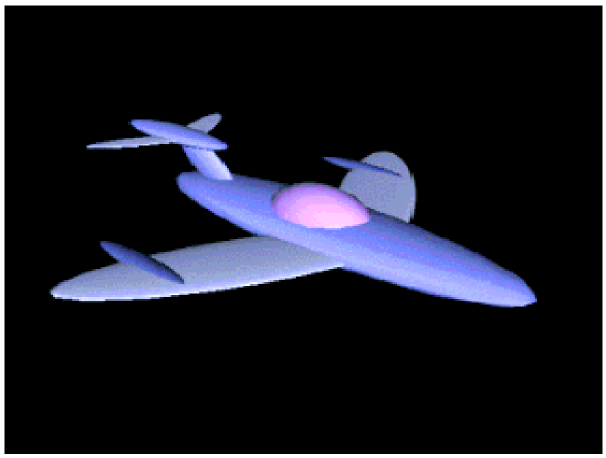
- **`Box`** nodes create the bars
- **`Text`** nodes provide bar labels
- **`Billboard`** nodes keep the labels facing the viewer



[ barplot.wrl ]

# *A simple spaceship*

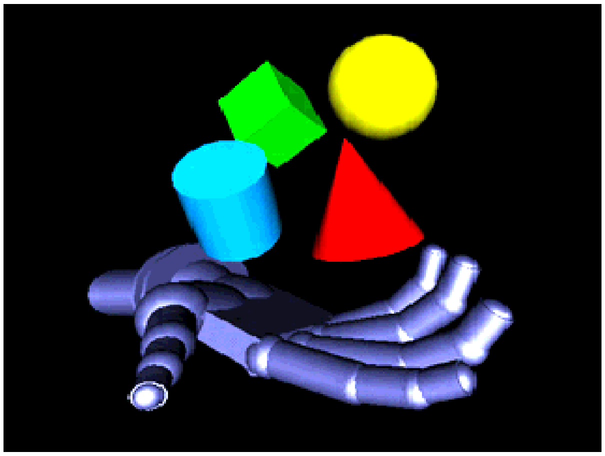
- **`Sphere`** nodes make up all parts of the ship
- **`Transform`** nodes scale the spheres into ship parts



[ space2.wrl ]

# *A juggling hand*

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



[ hand.wrl ]

# Introducing animation

# *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

# *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)

# *Using animation circuits*

- 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

# *Routing events*

- To set up an animation circuit, you need three things:

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

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

  3. A route connecting them

# *Using node inputs and outputs*

- 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*

# *Sample inputs*

- A **`Transform`** node has these eventIns:
  - **`set_translation`**
  - **`set_rotation`**
  - **`set_scale`**

- A **`Material`** node has these eventIns:
  - **`set_diffuseColor`**
  - **`set_emissiveColor`**
  - **`set_transparency`**

# *Sample outputs*

- An **`OrientationInterpolator`** node has this eventOut:
  - **`value_changed`** to send rotation values
- A **`PositionInterpolator`** node has this eventOut:
  - **`value_changed`** to send position (translation) values
- A **`TimeSensor`** node has this eventOut:
  - **`time`** to send time values

# *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
```

- **`ROUTE`** and **`TO`** must be in upper-case

# *Event data types*

- Sender and receiver event data types must match!
- Data types have names with a standard format, such as:
  **`SFString`**, **`SFRotation`**, or **`MFColor`**

| Character | Values |
|---|---|
| 1 | **`S`**: Single value<br>**`M`**: Multiple values |
| 2 | Always an **`F`** |
| remainder | Name of data type, such as **`String`**, **`Rotation`**, or **`Color`** |

# *Event data types*

| Data type | Meaning |
|---|---|
| `SFBool` | Boolean, true or false value |
| `SFColor`, `MFColor` | RGB color value |
| `SFFloat`, `MFFloat` | Floating point value |
| `SFImage` | Image value |
| `SFInt32`, `MFInt32` | Integer value |
| `SFNode`, `MFNode` | Node value |

# *Event data types*

| Data type | Meaning |
|---|---|
| **`SFRotation`**, **`MFRotation`** | Rotation value |
| **`SFString`**, **`MFString`** | Text string value |
| **`SFTime`** | Time value |
| **`SFVec2f`**, **`MFVec2f`** | XY floating point value |
| **`SFVec3f`**, **`MFVec3f`** | XYZ floating point value |

# *Following naming conventions*

- Most nodes have *exposedFields*

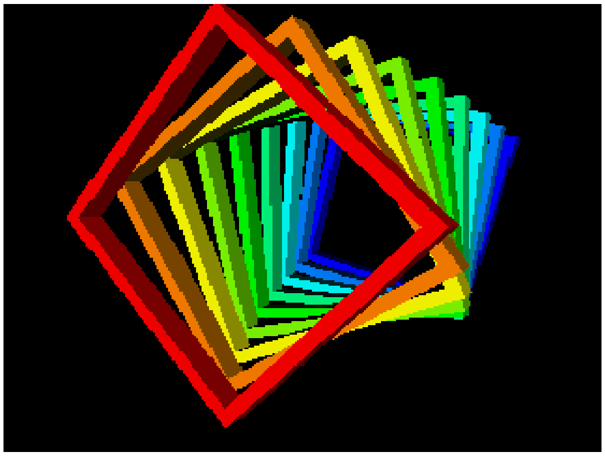
- If the exposed field name is **`xxx`**, then:
  - **`set_xxx`** is an *eventIn* to set the field
  - **`xxx_changed`** is an *eventOut* that sends when the field changes
  - The **`set_`** and **`_changed`** sufixes are optional but recommended for clarity

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

# *Animation example code*

```
DEF Touch TouchSensor { }

DEF Timer1 TimeSensor { . . . }

DEF Rot1 OrientationInterpolator { . . . }

DEF Frame1 Transform {
    children [
        Shape { . . . }
    ]
}

ROUTE Touch.touchTime TO Timer1.set_startTime
ROUTE Timer1.fraction_changed TO Rot1.set_fraction
ROUTE Rot1.value_changed TO Frame1.set_rotation
```

# *Animation example*



[ colors.wrl ]

# *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

# *Summary*

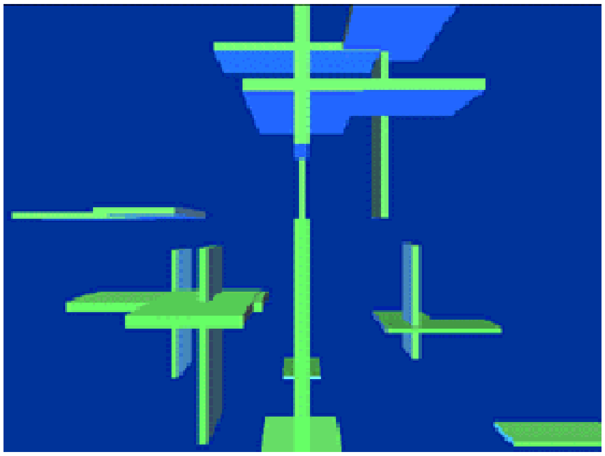
- Connect senders to receivers using routes
- *eventIns* are inputs, and *eventOuts* are outputs
- A route names the *sender.eventOut*, and the *receiver.eventIn*
  - Data types must match
- You can have multiple routes into or out of a node

# Animating transforms

# *Motivation*

- 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

# *Example*



[ floater.wrl ]

# *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

# *Using fractional time*

- Fractional time events give a number from 0.0 to 1.0
    - When the sensor starts, it outputs a 0.0
    - At the end of a *cycle*, it outputs a 1.0
    - The number of seconds between 0.0 and 1.0 is controlled by the *cycle interval*
- The sensor can loop forever, or run through only one cycle and stop

# *Syntax: TimeSensor*

- A **`TimeSensor`** node generates events based upon time
    - **`startTime`** and **`stopTime`** **-** when to run
    - **`cycleInterval`** **-** how long a cycle is
    - **`loop`** **-** whether or not to repeat cycles

```
TimeSensor {
    cycleInterval 1.0
    loop FALSE
    startTime 0.0
    stopTime 0.0
}
```

# *Using timers*

- To create a continuously running timer:

  **`loop TRUE`**

  **`stopTime`** <= **`startTime`**

- When stop time <= start time, stop time is ignored

# *Using timers*

- To run until the stop time:

  **`loop TRUE`**

  **`stopTime`** > **`startTime`**

- To run one cycle then stop:

  **`loop FALSE`**

  **`stopTime`** <= **`startTime`**

# *Using timers*

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

# *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

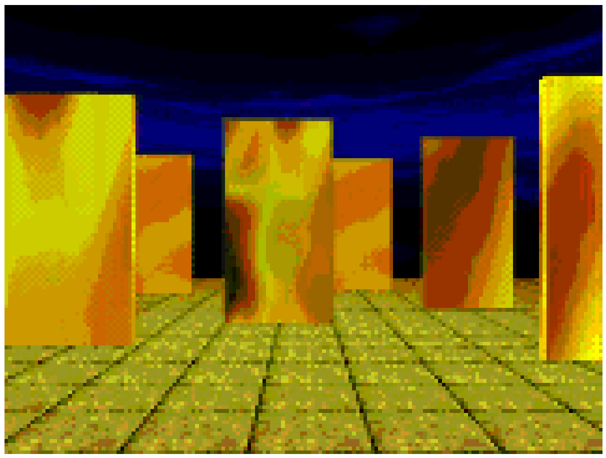
- The **`fraction_changed`** output event:
  - Outputs values from 0.0 to 1.0 during a cycle
  - Resets to 0.0 at the start of each cycle

# *Time sensor example code*

```
Shape {
    appearance Appearance {
        material DEF Monolith1Facade Material {
            diffuseColor 0.2 0.2 0.2
        }
    }
    geometry Box { size 2.0 4.0 0.3 }
}
DEF Monolith1Timer TimeSensor {
    cycleInterval 4.0
    loop FALSE
    startTime 0.0
    stopTime  0.1
}

ROUTE Monolith1Touch.touchTime
   TO Monolith1Timer.set_startTime
ROUTE Monolith1Timer.fraction_changed
   TO Monolith1Facade.set_transparency
```

Animating transforms

# *Time sensor example*



[ monolith.wrl ]

# *Converting time to position*

- 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

# *Interpolating positions*

- Each key position along a path has:
  - A *key value* (such as a position)
  - A *key* fractional time

- Interpolation fills in values between your key values:

| Fractional Time | Position |
|---|---|
| **0.0** | **0.0 0.0 0.0** |
| *0.1* | *0.4 0.1 0.0* |
| *0.2* | *0.8 0.2 0.0* |
| . . . | . . . |
| **0.5** | **4.0 1.0 0.0** |
| . . . | . . . |

# *Syntax: PositionInterpolator*

- A **`PositionInterpolator`** node describes a position path
  - **`key`** - key fractional times
  - **`keyValue`** - key positions

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

- Typically route into a **`Transform`** node's **`set_translation`** input

# *Using position interpolator inputs and outputs*

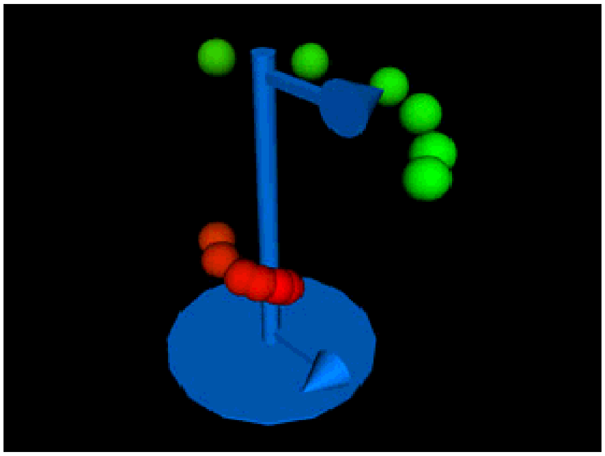
- The **`set_fraction`** input:
  - Sets the current fractional time along the key path

- The **`value_changed`** output:
  - Outputs the position along the path each time the fraction is set

# *Position interpolator example code*

```
DEF Particle1 Transform { . . . }
DEF Timer1 TimeSensor {
    cycleInterval 12.0
    loop TRUE
}
DEF Position1 PositionInterpolator {
    key  [ 0.0, . . . ]
    keyValue [ 0.0 0.0 0.0, . . .]
}
ROUTE Timer1.fraction_changed TO Position1.set_fraction
ROUTE Position1.value_changed TO Particle1.set_translation
```

# *Position interpolator example*



[ spiral.wrl ]

# *Using other types of interpolators*

| | |
|---|---|
| **Animate position** | `PositionInterpolator` |
| **Animate rotation** | `OrientationInterpolator` |
| **Animate scale** | `PositionInterpolator` |
| **Animate color** | `ColorInterpolator` |
| **Animate transparency** | `ScalarInterpolator` |

# *Syntax: OrientationInterpolator*

- A **`OrientationInterpolator`** node describes an orientation path
  - **`key`** - key fractional times
  - **`keyValue`** - key rotations (axis and angle)

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

- Typically route into a **`Transform`** node's **`set_rotation`** input

# *Syntax: PositionInterpolator*

- A **`PositionInterpolator`** node describes a position *or scale* path
  - **`key`** - key fractional times
  - **`keyValue`** - key positions (or scales)

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

- Typically route into a **`Transform`** node's **`set_scale`** input

# *Syntax: ColorInterpolator*

- **`ColorInterpolator`** node describes a color path
  - **`key`** - key fractional times
  - **`keyValue`** - key colors (red, green, blue)

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

- Typically route into a **`Material`** node's **`set_diffuseColor`** or **`set_emissiveColor`** inputs

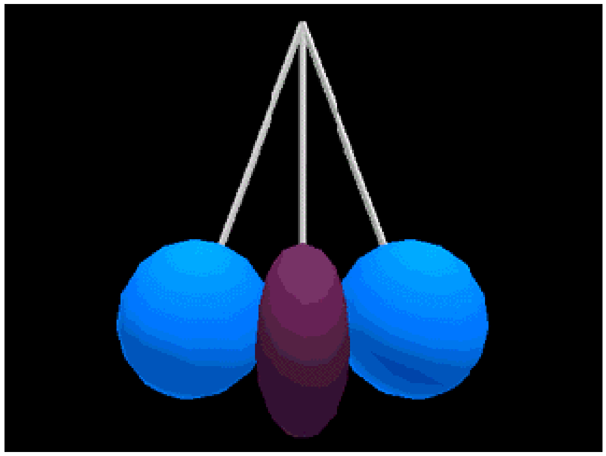
# *Syntax: ScalarInterpolator*

- **`ScalarInterpolator`** node describes a scalar path
  - **`key`** - key fractional times
  - **`keyValue`** - key scalars (used for anything)

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

- Often route into a **`Material`** node's **`set_transparency`** input

# *Other interpolators example*



[ squisher.wrl ]

# *Summary*

- 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

# *Summary*

- 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

Sensing viewer actions

# *Motivation*

- 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

# *Using action sensors*

- 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

# *Sensing shapes*

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

# *Syntax: TouchSensor*

- 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

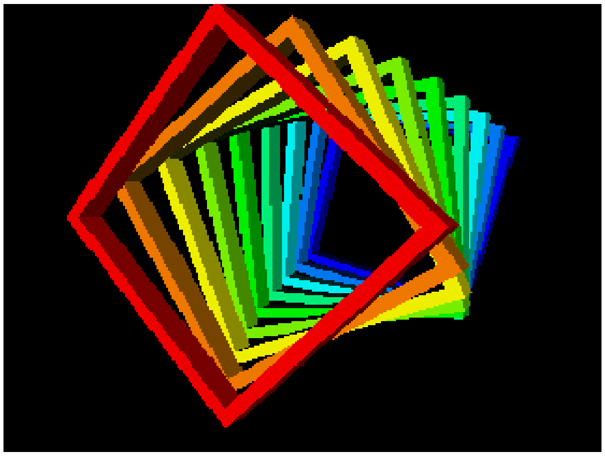
```
Transform {
    children [
        DEF Touched TouchSensor { }
        Shape { . . . }
        . . .
    ]
}
```

# *Touch sensor example code*

```
DEF Touch TouchSensor { }
DEF Timer1 TimeSensor { . . . }
DEF Rot1 OrientationInterpolator { . . . }
DEF Frame1 Transform {
    children [
        Shape { . . . }
    ]
}

ROUTE Touch.touchTime TO Timer1.set_startTime
ROUTE Timer1.fraction_changed TO Rot1.set_fraction
ROUTE Rot1.value_changed TO Frame1.set_rotation
```

Sensing viewer actions

# *Touch sensor example*



[ colors.wrl ]

# *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 Rotator  SphereSensor { }
        DEF RotateMe Transform { . . . }
    ]
}
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 Rotator  CylinderSensor { }
        DEF RotateMe Transform { . . . }
    ]
}
ROUTE Rotator.rotation_changed TO RotateMe.set_rotation
```

# *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

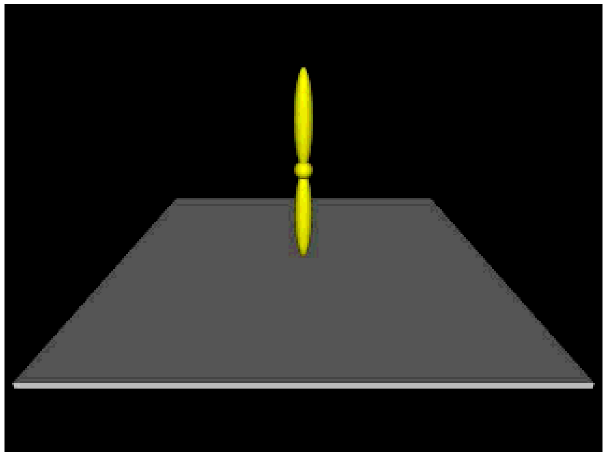
```
Transform {
    children [
        DEF Mover  PlaneSensor { }
        DEF MoveMe Transform { . . . }
    ]
}
ROUTE Mover.translation_changed TO MoveMe.set_translatior
```

# *Using multiple sensors*

- Multiple sensors can sense the same shape *but*. . .
  - If sensors are in the same group:
    - They all respond

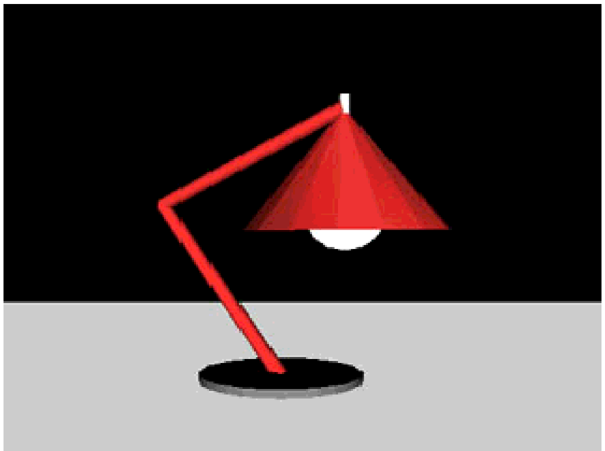
  - If sensors are at different depths in the hierarchy:
    - The deepest sensor responds
    - The other sensors do not respond

# *Multiple sensors example*



[ nested.wrl ]

# *Multiple sensors example*



[ lamp.wrl ]

# *Summary*

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

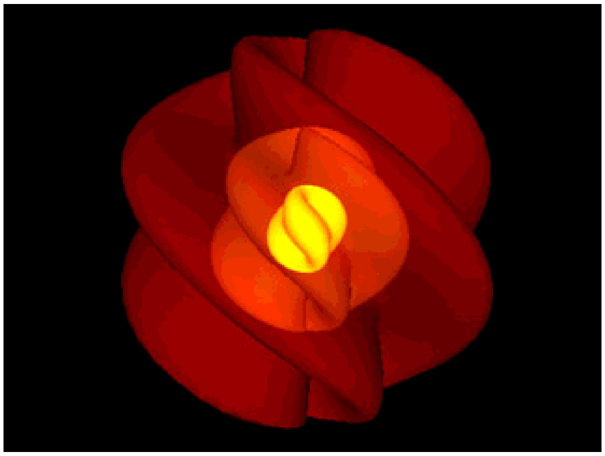
- Sensors convert viewer actions into events to
    - Start and stop animations
    - Orient shapes
    - Position shapes

# Building shapes out of points, lines, and faces

# *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

# *Example*



[ isosurf.wrl ]

# *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

# *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 coordinate-based shapes using geometry nodes:
  - **`PointSet`**
  - **`IndexedLineSet`**
  - **`IndexedFaceSet`**

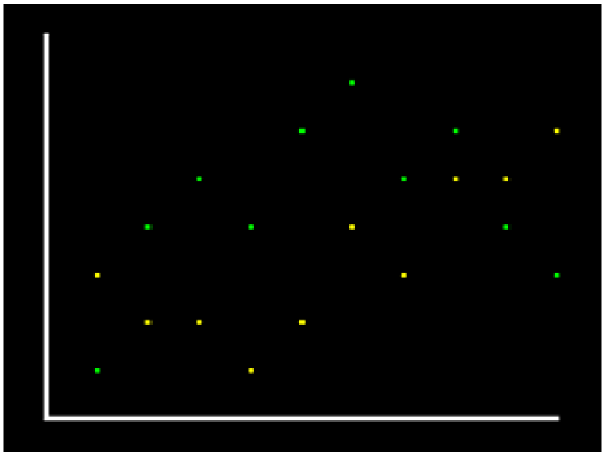
- For all three nodes, use a **`Coordinate`** node as the value of the **`coord`** field

# *Syntax: PointSet*

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

```
Shape {
    appearance Appearance { . . . }
    geometry PointSet {
        coord Coordinate {
            point [  . . .  ]
        }
    }
}
```

# *Point set example*



[ ptplot.wrl ]

# *Syntax: IndexedLineSet*

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

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

  - List coordinate indexes in the **`coordIndex`** field of the **`IndexedLineSet`** node

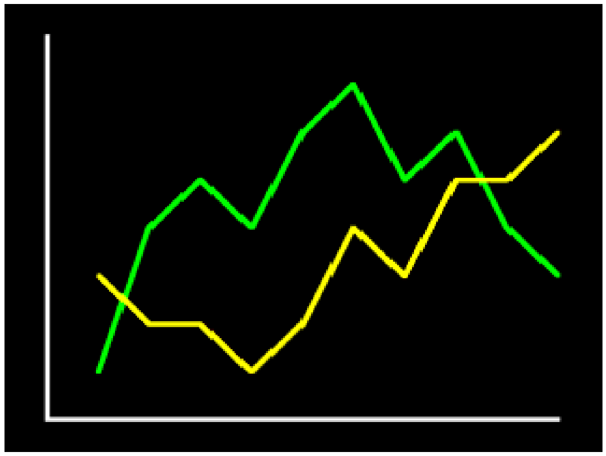
# *Using line set coordinate index lists*

- A line is drawn between pairs of coordinate indexes
  - **`-1`** marks a break in the line
  - A line is *not* automatically drawn from the last index back to the first

```
coordIndex [ 1, 0, 3, 8, -1, 5, 9, 0 ]
```

| | |
|---|---|
| `1, 0, 3, 8,` | **Draw line from 1 to 0 to 3 to 8** |
| `-1,` | **End line, start next** |
| `5, 9, 0` | **Draw line from 5 to 9 to 0** |

# *IndexedLineSet example*



[ lnplot.wrl ]

# *Syntax: IndexedFaceSet*

- An **`IndexedFaceSet`** geometry node creates geometry out of *faces*
  - A flat *face* (polygon) is drawn using an outline specified by coordinate indexes

```
Shape {
    appearance Appearance { . . . }
    geometry IndexedFaceSet {
        coord Coordinate {
            point [  . . .  ]
        }
        coordIndex [ . . . ]
    }
}
```

# *Using face set coordinate index lists*

- To build a face shape
  - Make a list of coordinates, using their indexes
  - List coordinate indexes in the **`coordIndex`** field of the **`IndexedFaceSet`** node

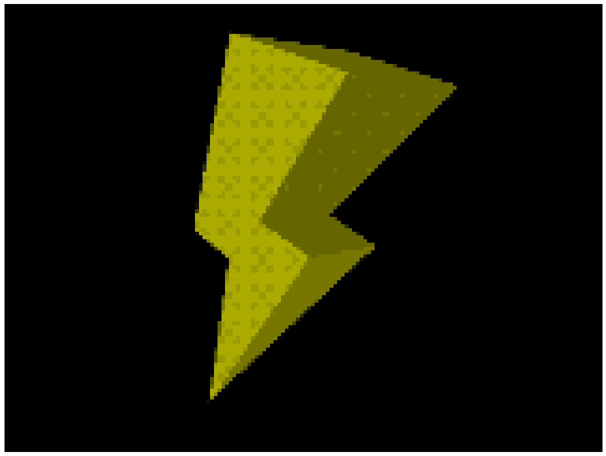
# Using face set coordinate index lists

- A triangle is drawn connecting sequences of coordinate indexes
  - `-1` marks a break in the sequence
  - Each face *is* automatically closed, connecting the last index back to the first

```
coordIndex [ 1, 0, 3, 8, -1, 5, 9, 0 ]
```

| | |
|---|---|
| `1, 0, 3, 8` | **Draw face from 1 to 0 to 3 to 8 to 1** |
| `-1,` | **End face, start next** |
| `5, 9, 0` | **Draw face from 5 to 9 to 0 to 5** |

# *IndexedFaceSet example*



[ lightng.wrl ]

# *Syntax: IndexedFaceSet*

- An **`IndexedFaceSet`** geometry node creates geometry out of *faces*
  - **`solid`** - shape is solid
  - **`ccw`** - faces are counter-clockwise
  - **`convex`** - faces are convex

```
Shape {
    appearance Appearance { . . . }
    geometry IndexedFaceSet {
        coord Coordinate { . . . }
        coordIndex [ . . . ]
        solid TRUE
        ccw TRUE
        convex TRUE
    }
}
```

# *Using shape control*

- A *solid* shape is one where the insides are never seen
  - If never seen, don't attempt to draw them
  - When **`solid TRUE`**, the *back* sides (inside) of faces are not drawn

- The front of a face has coordinates in *counter-clockwise order*
  - When **`ccw FALSE`**, the other side is the front

- Faces are assumed to be convex
  - When **`convex FALSE`**, concave faces are automatically broken into multiple convex faces

# *Syntax: CoordinateInterpolator*

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

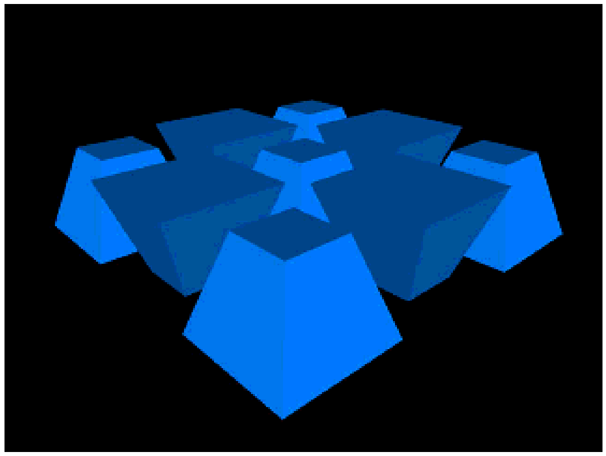
```
CoordinateInterpolator {
    key [ 0.0, . . . ]
    keyValue [ 0.0 1.0 0.0, . . . ]
}
```

- Typically route into a **`Coordinate`** node's **`set_point`** input

# *Interpolating coordinate lists*

- A **`CoordinateInterpolator`** node interpolates *lists* of coordinates
  - Each output is a *list* of coordinates
  - If **`n`** output coordinates are needed for **`t`** fractional times:
    - **`n × t`** coordinates are needed in the key value list

# Coordinate interpolator example



[ wiggle.wrl ]

# *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

# *Summary*

- 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

# *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

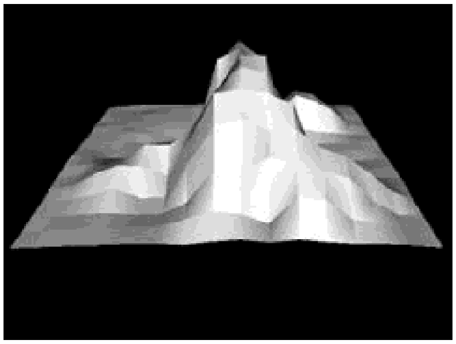
- The `CoordinateInterpolator` node converts times to coordinates

# Building elevation grids

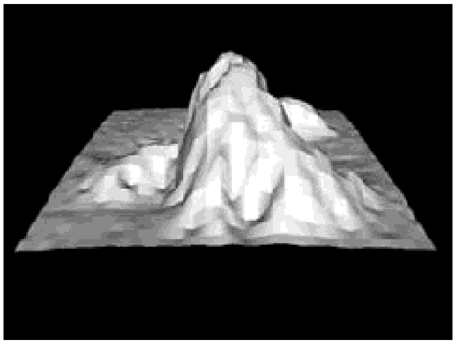
# *Motivation*

- 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

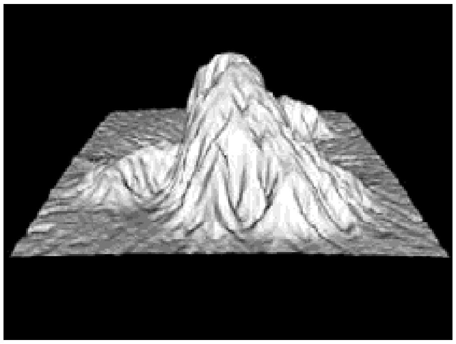
# *Example*



**[ 16 x 16: mount16.wrl ]**



**[ 32 x 32: mount32.wrl ]**



**[ 128 x 128: mount128.wrl ]**

# *Syntax: ElevationGrid*

- An **`ElevationGrid`** geometry node creates terrains
  - **`xDimension`** and **`zDimension`** - grid size
  - **`xSpacing`** and **`zSpacing`** - row and column distances

```
Shape {
    appearance Appearance { . . . }
    geometry ElevationGrid {
        xDimension 3
        zDimension 2
        xSpacing   1.0
        zSpacing   1.0
        . . .
    }
}
```

# *Syntax: ElevationGrid*

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

```
Shape {
    appearance Appearance { . . . }
    geometry ElevationGrid {
        . . .
        height [
            0.0, -0.5, 0.0,
            0.2,  4.0, 0.0
        ]
    }
}
```

# *Syntax: ElevationGrid*

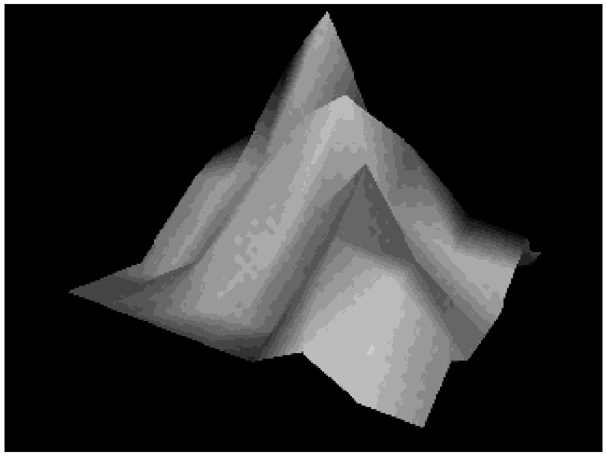
- An **`ElevationGrid`** geometry node creates terrains
  - **`solid`** - shape is solid
  - **`ccw`** - faces are counter-clockwise

```
Shape {
    appearance Appearance { . . . }
    geometry ElevationGrid {
        . . .
        solid TRUE
        ccw TRUE
    }
}
```

# Elevation grid example code

```
Shape {
  appearance Appearance { . . . }
  geometry ElevationGrid {
    xDimension 9
    zDimension 9
    xSpacing   1.0
    zSpacing   1.0
    solid FALSE
    height [
      0.0, 0.0, 0.5, 1.0, 0.5, 0.0, 0.0, 0.0, 0.0,
      0.0, 0.0, 0.0, 0.0, 2.5, 0.5, 0.0, 0.0, 0.0,
      0.0, 0.0, 0.5, 0.5, 3.0, 1.0, 0.5, 0.0, 1.0,
      0.0, 0.0, 0.5, 2.0, 4.5, 2.5, 1.0, 1.5, 0.5,
      1.0, 2.5, 3.0, 4.5, 5.5, 3.5, 3.0, 1.0, 0.0,
      0.5, 2.0, 2.0, 2.5, 3.5, 4.0, 2.0, 0.5, 0.0,
      0.0, 0.0, 0.5, 1.5, 1.0, 2.0, 3.0, 1.5, 0.0,
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0, 1.5, 0.5,
      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.0,
    ]
  }
}
```

# *Elevation grid example*



[ mount.wrl ]

# *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*

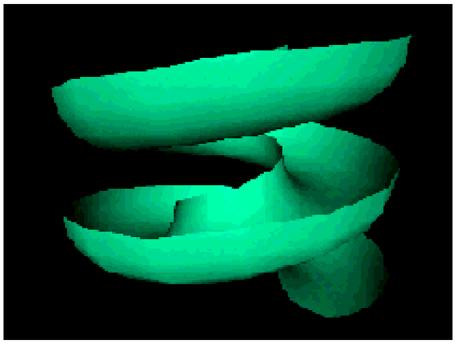
- Extruded shapes are very common
  - Tubes, pipes, bars, vases, donuts
  - Other tricky uses...

- You can build extruded shapes using an **`IndexedFaceSet`** node

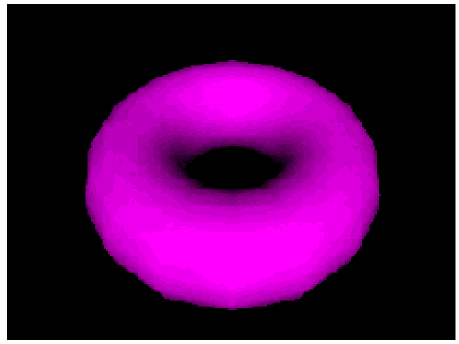
- You can build extruded shapes more easily and efficiently using an **`Extrusion`** node

Building extruded shapes

# *Examples*



[ slide.wrl ]



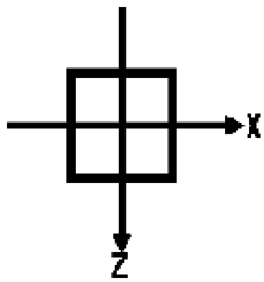
[ donut.wrl ]

# *Creating extruded shapes*

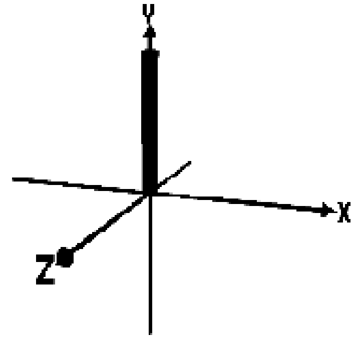
- Extruded shapes are described by
  - A 2-D *cross-section*
  - A 3-D *spine* along which to sweep the cross-section

- 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*

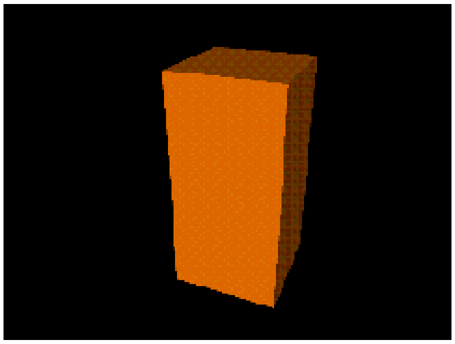
# *Extruding along a straight line*




a. Square cross-section

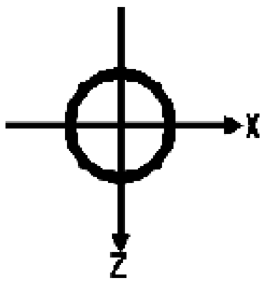



b. Straight spine

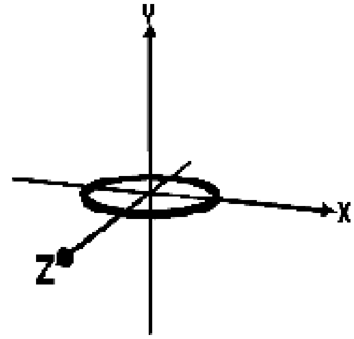


c. Resulting extrusion

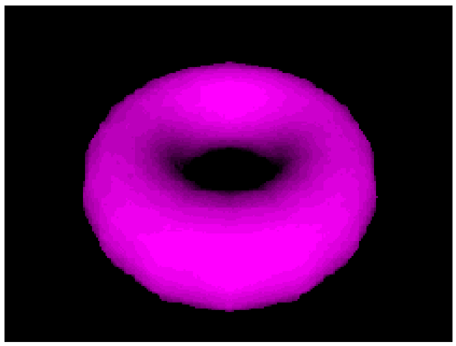
# *Extruding around a circle*




a. Circular cross-section

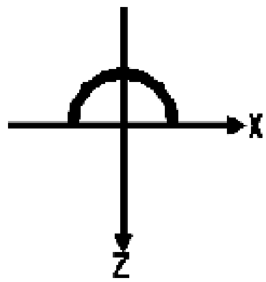



b. Circular spine

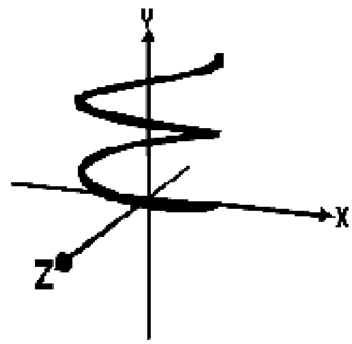


c. Resulting extrusion

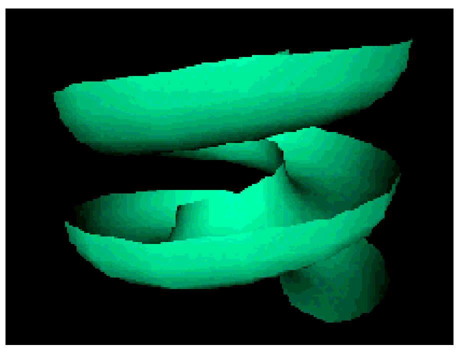
# *Extruding along a helix*




a. Half-circle cross-section




b. Helical spine



c. Resulting extrusion

# *Syntax: Extrusion*

- An **`Extrusion`** geometry node creates extruded geometry
  - **`cross-section`** - 2-D cross-section
  - **`spine`** - 3-D sweep path
  - **`endCap`** and **`beginCap`** - cap ends

```
Shape {
    appearance Appearance { . . . }
    geometry Extrusion {
        crossSection [ . . . ]
        spine [ . . .  ]
        endCap TRUE
        beginCap TRUE
        . . .
    }
}
```

# *Syntax: Extrusion*

- An **`Extrusion`** geometry node creates extruded geometry
  - **`solid`** - shape is solid
  - **`ccw`** - faces are counter-clockwise
  - **`convex`** - faces are convex

```
Shape {
    appearance Appearance { . . . }
    geometry Extrusion {
        . . .
        solid TRUE
        ccw TRUE
        convex TRUE
    }
}
```

# *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

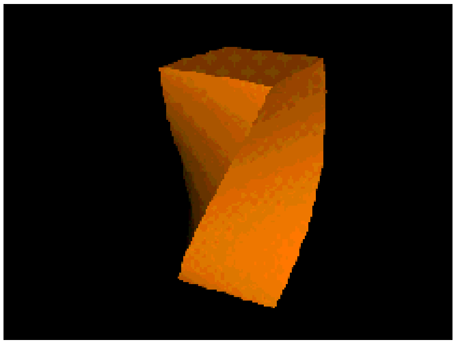
# *Syntax: Extrusion*

- An **`Extrusion`** geometry node creates geometry using
  - **`scale`** - cross-section scaling per spine point
  - **`orientation`** - cross-section rotation per spine point

```
Shape {
    appearance Appearance { . . . }
    geometry Extrusion {
        . . .
        scale [ . . . ]
        orientation [ . . . ]
    }
}
```

# *Sample extrusions with scale and rotation*

**[ horn.wrl ]**



**[ bartwist.wrl ]**

# *Summary*

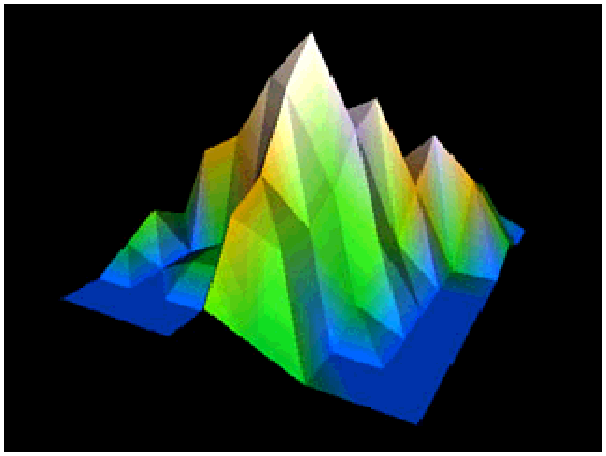
- An **`Extrusion`** node efficiently creates extruded shapes
- The **`crossSection`** field specifies the cross-section
- The **`spine`** field specifies the sweep path
- The **`scale`** and **`orientation`** fields specify scaling and rotation at each spine point

# Controlling color on coordinate-based geometry

# *Motivation*

- The `Material` node gives an entire shape the same color
- You can provide colors for individual parts of a shape using a `Color` node

# *Example*



[ cmount.wrl ]

Controlling color on coordinate-based geometry

# *Syntax: Color*

- A **`Color`** node contains a list of RGB values (similar to a **`Coordinate`** node)

```
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

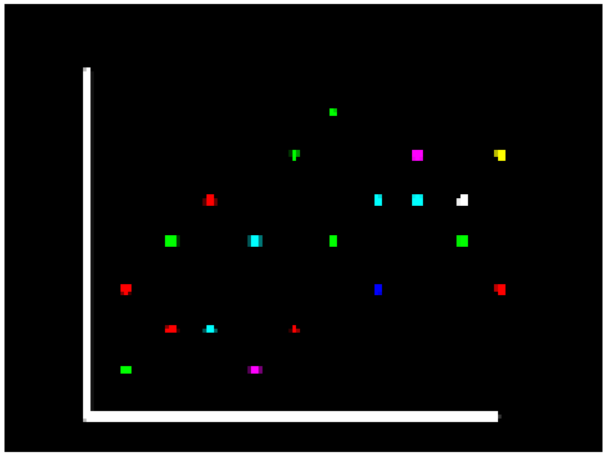
# *Syntax: PointSet*

- A **`PointSet`** geometry node creates geometry out of *points*
  - **`color`** - provides a list of colors
  - Always binds one color to each point, in order

```
Shape {
    appearance Appearance { . . . }
    geometry PointSet {
        coord Coordinate { . . . }
        color Color { . . . }
    }
}
```

Controlling color on coordinate-based geometry

# *PointSet example*



[ scatter.wrl ]

Controlling color on coordinate-based geometry

# *Syntax: IndexedLineSet*

- An **`IndexedLineSet`** geometry node creates geometry out of lines
  - **`color`** - list of colors
  - **`colorIndex`** - selects colors from list
  - **`colorPerVertex`** - control color binding

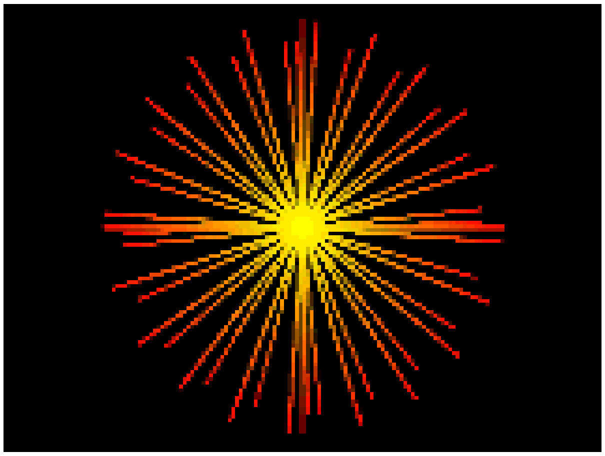
```
Shape {
    appearance Appearance { . . . }
    geometry IndexedLineSet {
        coord Coordinate { . . . }
        coordIndex [ . . . ]
        color Color { . . . }
        colorIndex [ . . . ]
        colorPerVertex TRUE
    }
}
```

# ***Controlling color binding for line sets***

- 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 color on coordinate-based geometry

# *IndexedLineSet example*



[ burst.wrl ]

# *Syntax: IndexedFaceSet*

- An **`IndexedFaceSet`** geometry node creates geometry out of faces
  - **`color`** - list of colors
  - **`colorIndex`** - selects colors from list
  - **`colorPerVertex`** - control color binding

```
Shape {
    appearance Appearance { . . . }
    geometry IndexedFaceSet {
        coord Coordinate { . . . }
        coordIndex [ . . . ]
        color Color { . . . }
        colorIndex [ . . . ]
        colorPerVertex TRUE
    }
}
```

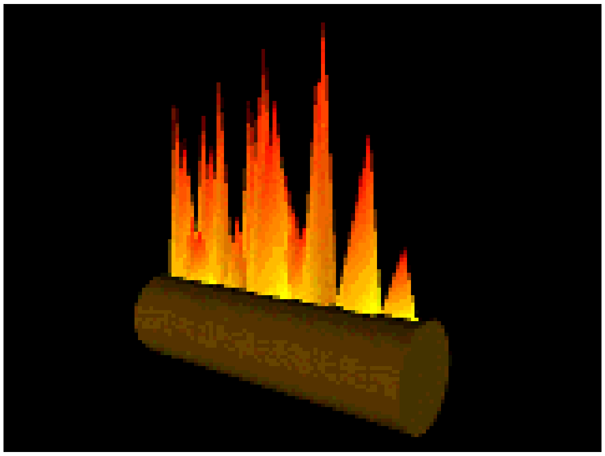
# *Controlling color binding for face sets*

- 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)

Controlling color on coordinate-based geometry

# *IndexedFaceSet example*



[ log.wrl ]

# *Syntax: ElevationGrid*

- An **`ElevationGrid`** geometry node creates terrains
  - **`color`** - list of colors
  - **`colorPerVertex`** - control color binding
  - Always binds one color to each grid point or square, in order

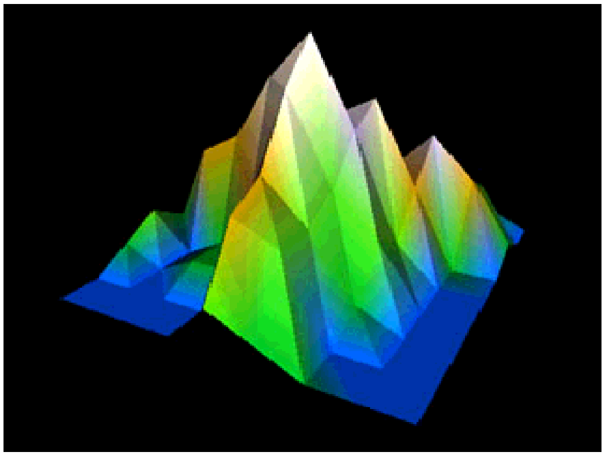
```
Shape {
    appearance Appearance { . . . }
    geometry ElevationGrid {
        . . .
        height [ . . . ]
        color Color { . . . }
        colorPerVertex TRUE
    }
}
```

# *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 color on coordinate-based geometry

# *ElevationGrid example*



[ cmount.wrl ]

# *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

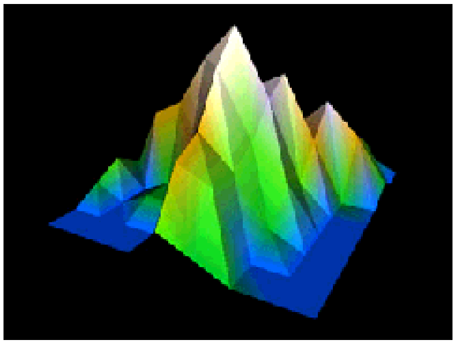
# Controlling shading on coordinate-based geometry

# *Motivation*

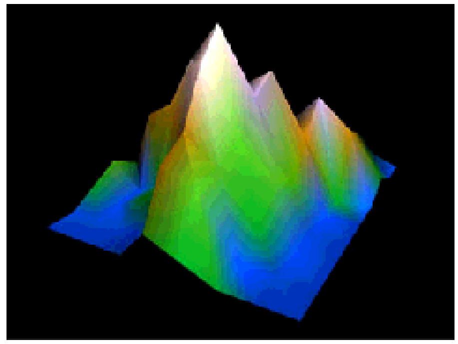
- When shaded, the faces on a shape are obvious
- To create a smooth shape you can use a large number of small faces
  - Requires lots of faces, disk space, memory, and drawing time
- Instead, use *smooth shading* to create the illusion of a smooth shape, but with a small number of faces

Controlling shading on coordinate-based geometry

# *Examples*



[ cmount.wrl ]
a. No smooth shading



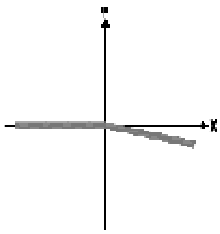
[ cmount2.wrl ]
b. With smooth shading

# *Controlling shading using the crease angle*

- By default, faces are drawn with faceted shading
- You can enable smooth shading using the **`creaseAngle`** field for
  - **`IndexedFaceSet`**
  - **`ElevationGrid`**
  - **`Extrusion`**

# *Selecting crease angles*

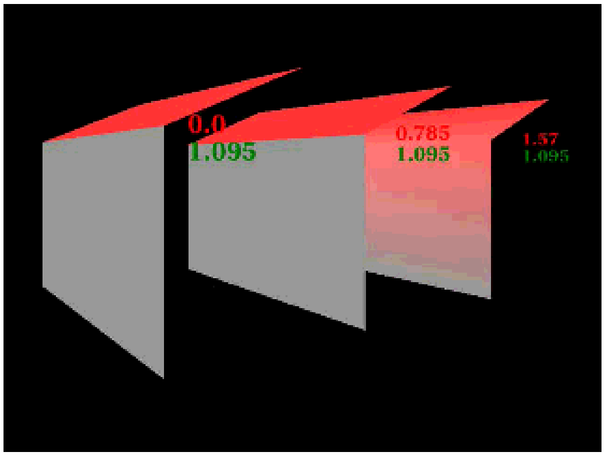
- A *crease angle* is a threshold angle between two faces



- **If face angle >= crease angle, use facet shading**
- **If face angle < crease angle, use smoot shading**

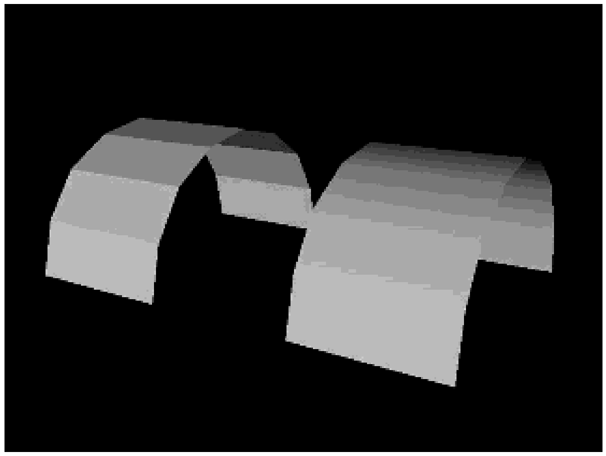
Controlling shading on coordinate-based geometry

# *Crease angle example*




[ creangle.wrl ]

# *Crease angle example*



[ hcyl.wrl ]
Left has crease angle = 0 (faceted),
Right has crease angle = 1.571 (smooth)

Controlling shading on coordinate-based geometry

# *Using normals*

- A *normal vector* indicates the direction a face is facing
  - If it faces a light, the face is shaded bright

- By defualt, normals are automatically generated by the VRML browser
  - You can specify your own normals with a **`Normal`** node

  - Usually automatically generated normals are good enough

Controlling shading on 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, . . . ]
}
```

- Normals can be given for **`IndexedFaceSet`** and **`ElevationGrid`** nodes

Controlling shading on coordinate-based geometry

# *Syntax: IndexedFaceSet*

- An **`IndexedFaceSet`** geometry node creates geometry out of faces
  - **`normal`** - list of normals
  - **`normalIndex`** - selects normals from list
  - **`normalPerVertex`** - control normal binding

```
Shape {
    appearance Appearance { . . . }
    geometry IndexedFaceSet {
        coord Coordinate { . . . }
        coordIndex [ . . . ]
        normal Normal { . . . }
        normalIndex [ . . . ]
        normalPerVertex TRUE
    }
}
```

# *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)

Controlling shading on coordinate-based geometry

# *Syntax: ElevationGrid*

- An **`ElevationGrid`** geometry node creates terrains
  - **`normal`** - list of normals
  - **`normalPerVertex`** - control normal binding
  - Always binds one normal to each grid point or square, in order

```
Shape {
    appearance Appearance { . . . }
    geometry ElevationGrid {
        height [ . . . ]
        normal Normal { . . . }
        normalPerVertex TRUE
    }
}
```

# *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

Controlling shading on coordinate-based geometry

# *Syntax: NormalInterpolator*

- A **`NormalInterpolator`** node describes a normal set
  - **`keys`** - key fractions
  - **`values`** - key normal lists (X,Y,Z lists)
  - Interpolates *lists* of normals, similar to the **`CoordinateInterpolator`**

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

- Typically route into a **`Normal`** node's **`set_vector`** input

# *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

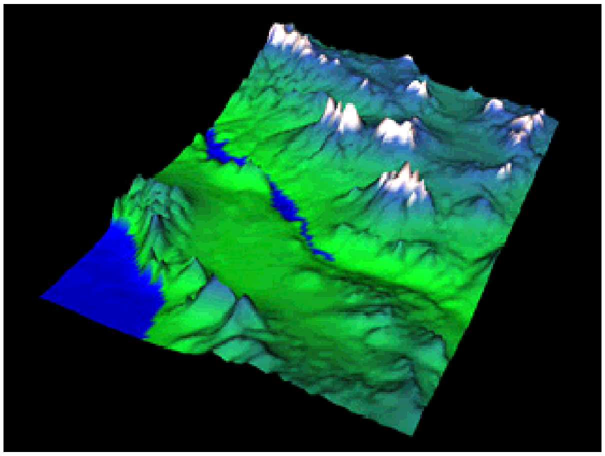
- The **`normalPerVertex`** field selects normal per face/grid square or normal per coordinate

- The **`NormalInterpolator`** node converts times to normals

# Summary examples

# *A terrain*

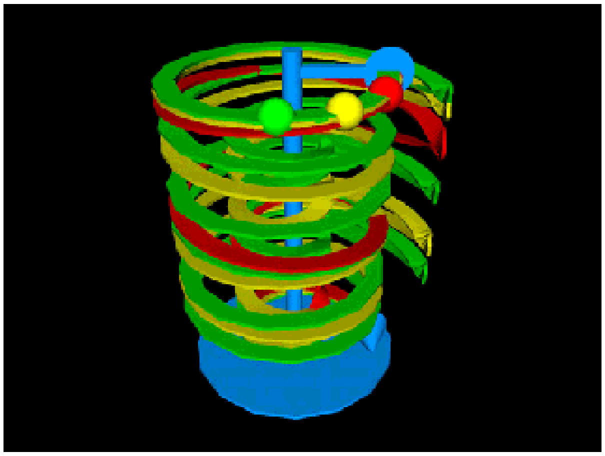
- An **`ElevationGrid`** node creates a terrain
- A **`Color`** node provides terrain colors



[ land.wrl ]

# *Particle flow*

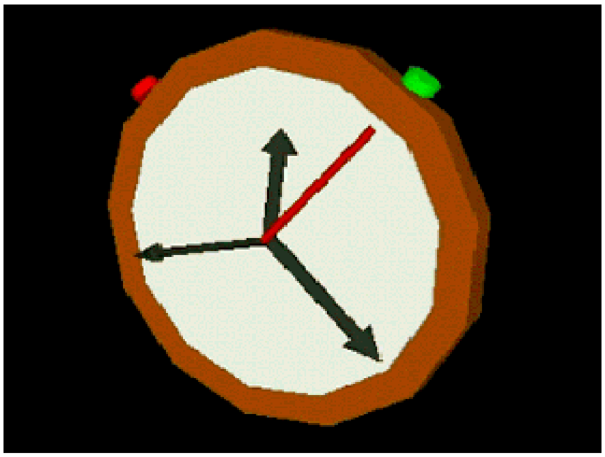
- Multiple **`Extrusion`** nodes trace particle paths
- Multiple **`PositionInterpolator`** nodes define particle animation paths
- Multiple **`TimeSensor`** nodes clock the animation using different starting times



[ espiralm.wrl ]

# *A real-time clock*

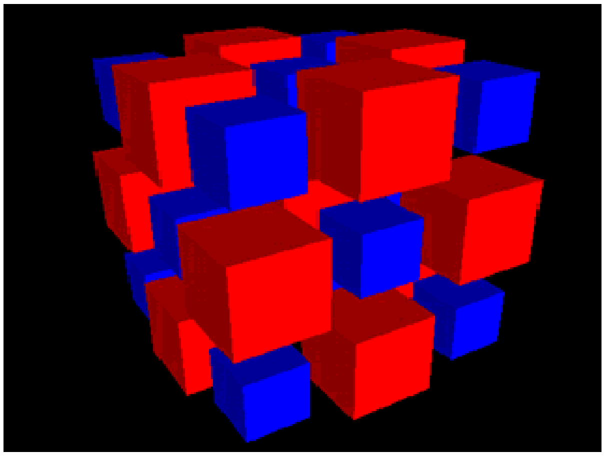
- A set of **`TimeSensor`** nodes watch the time
- A set of **`OrientationInterpolator`** nodes spin the clock hands



[ stopwtch.wrl ]

# *A timed timer*

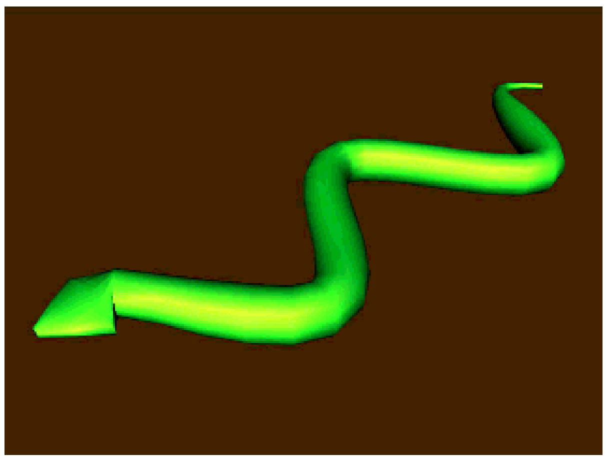
- A first **`TimeSensor`** node clocks a second **`TimeSensor`** node to create a periodic animation



[ timetime.wrl ]

# *A morphing snake*

- A **`CoordinateInterpolator`** node animates the spine of an **`Extrusion`** node



[ snake.wrl ]

# Mapping textures

# *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*

# *Using image textures*

- Image texture
  - Uses a single image from a file in one of these formats:

| | |
|---|---|
| **`GIF`** | • 8-bit lossless compressed images<br>• 1 transparency color<br>• Usually a poor choice for texture mapping |
| **`JPEG`** | • 8-bit thru 24-bit lossy compressed images<br>• No transparency support<br>• An adequate choice for texture mapping |
| **`PNG`** | • 8-bit thru 24-bit lossless compressed images<br>• 8-bit transparency per pixel<br>• Best choice |

# *Using pixel textures*

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

  - The image is encoded using *hex*
    - Up to 10 bytes per pixel
    - *Very* inefficient
    - Only useful for very small textures
      - Stripes
      - Checkerboard patterns

# *Using movie textures*

- Movie texture
  - A movie from an MPEG-1 file

  - The movie plays back on the textured shape
    - Problematic in some browsers

# *Syntax: Appearance*

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

```
Shape {
    appearance Appearance {
        material Material { . . . }
        texture ImageTexture { . . . }
    }
    geometry . . .
}
```

# *Syntax: ImageTexture*

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

```
Shape {
    appearance Appearance {
        material Material { }
        texture ImageTexture {
            url "wood.jpg"
        }
    }
    geometry . . .
}
```

# *Syntax: PixelTexture*

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

```
Shape {
    appearance Appearance {
        material Material { }
        texture PixelTexture {
            image 2 1 3
                0xFFFF00 0xFF0000
        }
    }
    geometry . . .
}
```

# *Syntax: MovieTexture*

- 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)

```
Shape {
    appearance Appearance {
        material Material { }
        texture MovieTexture {
            url "movie.mpg"
            loop TRUE
            speed 1.0
            startTime 0.0
            stopTime 0.0
        }
    }
    geometry . . .
}
```

# *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
- If there is *no* **`Material`** node, the texture is applied *emissively*

# Colorizing textures

a. Grayscale wood texture

b. Six wood colors from one colorized texture

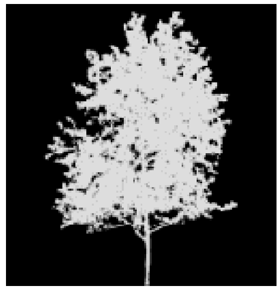
# *Using transparent textures*

- Texture images can include *color* and *transparency* values for each pixel
  - Pixel transparency is also known as *alpha*

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

# *Transparent texture example*

a. Color portion of tree texture



b. Transparency portion of tree texture

# *Transparent texture example*

[ treewall.wrl ]

# *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

# Controlling how textures are mapped

# *Motivation*

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

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

# *Working through the texturing process*

- Imagine the texture image is a big piece of rubbery cookie dough

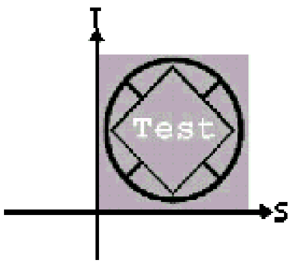
- Select a texture image piece
  - Define the shape of a cookie cutter
  - Position and orient the cookie cutter
  - Stamp out a piece of texture dough

- Stretch the rubbery texture cookie to fit a face

Controlling how textures are mapped

# *Using texture coordinate system*

- Texture images (the dough) are in a *texture coordinate system*

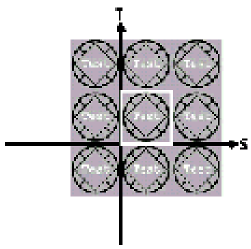



***S*** **direction is horizontal**
***T*** **direction is vertical**
**(0,0) at lower-left**
**(1,1) at upper-right**

Controlling how textures are mapped

# *Specifying texture coordinates*

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

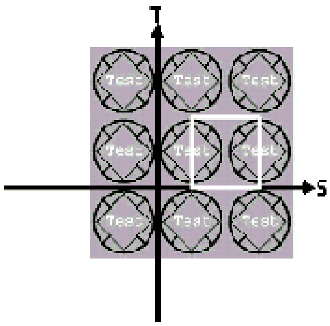


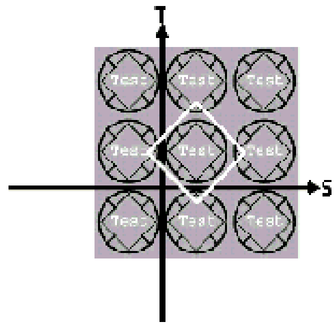

```
0.0 0.0,
1.0 0.0,
1.0 1.0,
0.0 1.0
```

Controlling how textures are mapped

# *Applying texture transforms*

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





Controlling how textures are mapped

# *Texturing a face*

- Bind the texture to a face (stretch the cookie and stick it)

# *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!

# *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
  - **`texCoord`** and **`texCoordIndex`** - specify texture pieces

```
Shape {
    appearance Appearance { . . . }
    geometry IndexedFaceSet {
        coord Coordinate { . . . }
        coordIndex [ . . . ]
        texCoord TextureCoordinate { . . . }
        texCoordIndex [ . . . ]
    }
}
```

# *Syntax: ElevationGrid*

- An **`ElevationGrid`** geometry node creates terrains
  - **`texCoord`** - specify texture pieces
  - Automatically generated texture coordinate indexes

```
Shape {
    appearance Appearance { . . . }
    geometry ElevationGrid {
        height [ . . . ]
        texCoord TextureCoordinate { . . . }
    }
}
```

# *Syntax: Appearance*

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

```
Shape {
    appearance Appearance {
        material Material { . . . }
        texture ImageTexture { . . . }
        textureTransform TextureTransform { . . . }
    }
    geometry . . .
}
```

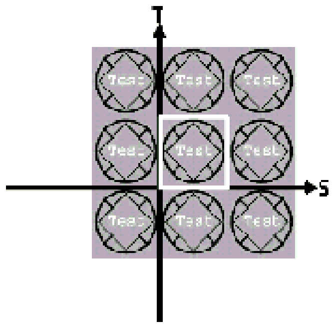
# *Syntax: TextureTransform*

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

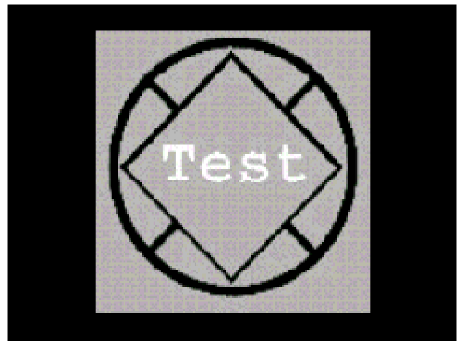
```
Shape {
    appearance Appearance {
        material Material { . . . }
        texture ImageTexture { . . . }
        textureTransform TextureTransform {
            translation 0.0 0.0
            rotation    0.0
            scale       1.0 1.0
        }
    }
}
```

Controlling how textures are mapped

# *No texture transform example*

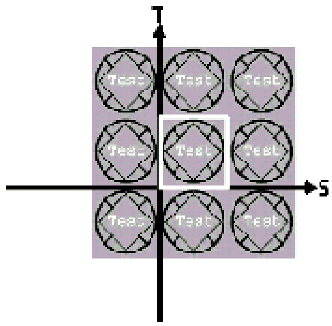



a. Texture in texture space

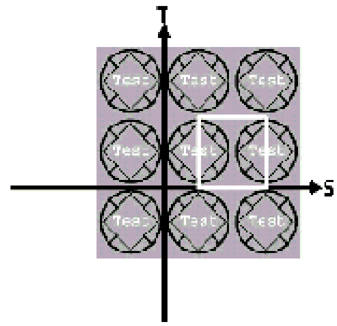



b. Texture on shape

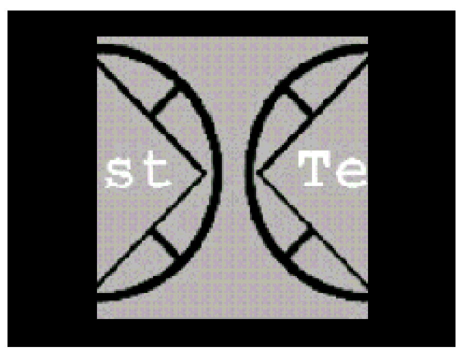
# *Texture translation example*




a. Texture in texture space



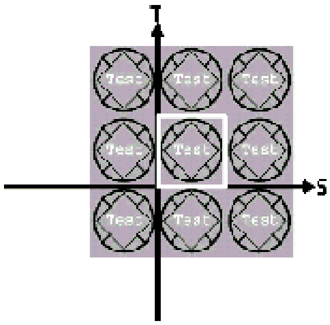

b. Translated cookie cutter



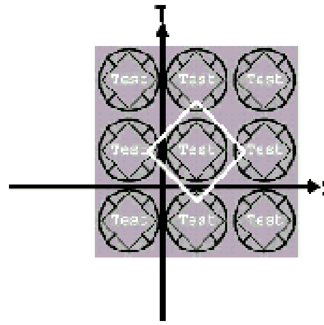

c. Texture on shape

Controlling how textures are mapped

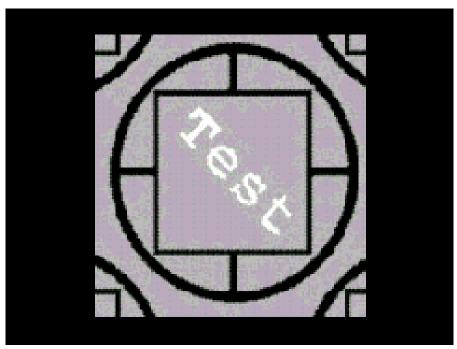
# ***Texture rotation example***




a. Texture in texture space



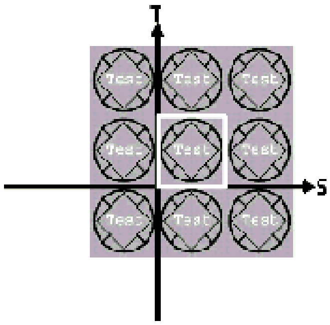

b. Rotated cookie cutter



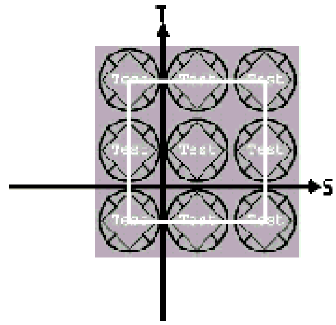

c. Texture on shape

Controlling how textures are mapped

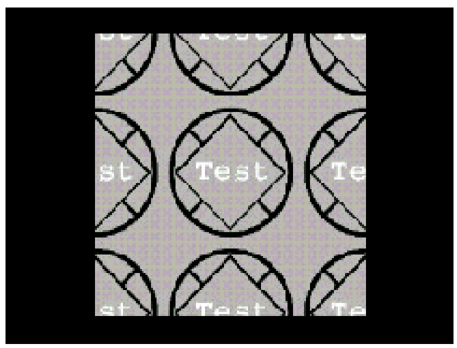
# *Texture scale example*




a. Texture in texture space




b. Scaled cookie cutter




c. Texture on shape

Controlling how textures are mapped

# *Texture coordinates example*

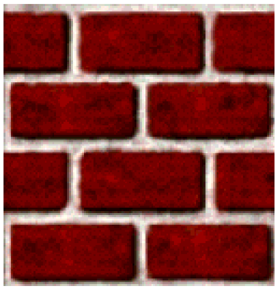
a. Texture image

[ cookie.wrl ]

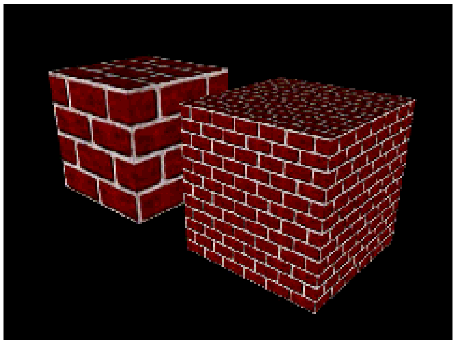
b. Texture on shapes

Controlling how textures are mapped

# *Texture scale example*



a. Texture image



[ brickb.wrl ]

b. Texture on shape

# *Scaling, rotating, and translating*

- *Scale*, *Rotate*, and *Translate* a texture cookie cutter one after the other

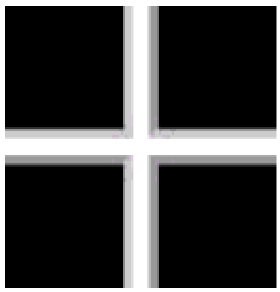
```
Shape {
    appearance Appearance {
        material Material { . . . }
        texture ImageTexture { . . . }
        textureTransform TextureTransform {
            translation 0.0 0.0
            rotation .785
            scale 8.5 8.5
        }
    }
}
```

# *Scaling, rotating, and translating*

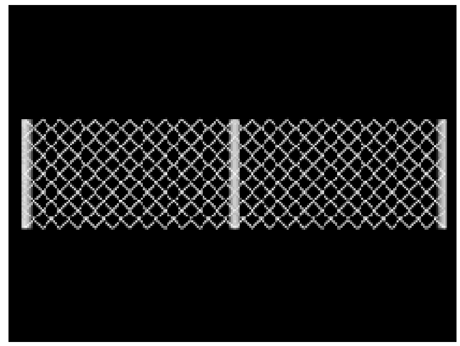
- Read texture transform operations *top-down*:
  - The cookie cutter is translated, rotated, then scaled
  - Order is fixed, independent of field order
  - This is the *reverse* of a **`Transform`** node

- This is a significant difference between VRML 2.0 and ISO VRML 97
  - VRML 2.0 uses scale, rotate, translate order
  - ISO VRML 97 uses translate, rotate, scale order

# *Texture scale and rotation example*



a. Texture image



[ fence.wrl ]
b. Texture on shape

# *Summary*

- Texture images are in a texture coordinate system
- Texture coordinates and indexes describe a texture cookie cutter
- Texture transforms translate, rotate, and scale place the cookie cutter
- Texture indexes bind the cut-out cookie texture to a face on a shape

# Lighting your world

# *Motivation*

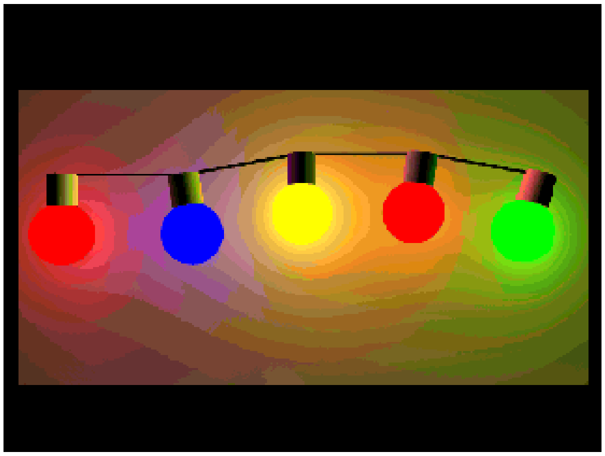
- By default, you have one light in the scene, attached to your head

- 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

# *Example*



# *Using types of lights*

- Theer 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

# *Using common lighting features*

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

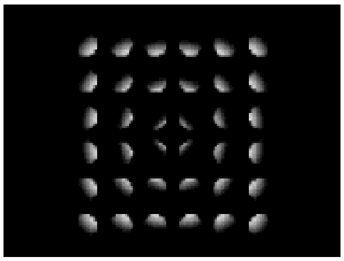
# *Using common lighting features*

- 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

# *Syntax: PointLight*

- A **`PointLight`** node illuminates radially from a point

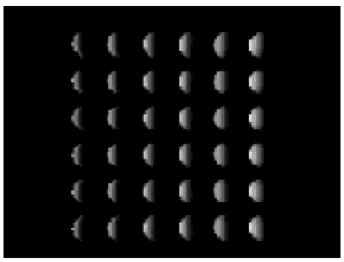


```
PointLight {
    location 0.0 0.0 0.0
    intensity 1.0
    color 1.0 1.0 1.0
}
```

[ pntlite.wrl ]

# *Syntax: DirectionalLight*

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

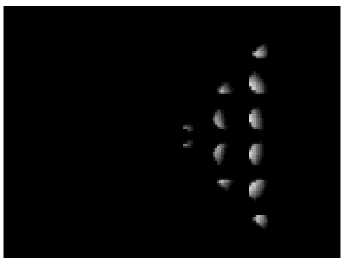


[ dirlite.wrl ]

```
DirectionalLight {
    direction 1.0 0.0 0.0
    intensity 1.0
    color 1.0 1.0 1.0
}
```

# *Syntax: SpotLight*

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



[ sptlite.wrl ]

```
SpotLight {
    location  0.0 0.0 0.0
    direction 1.0 0.0 0.0
    intensity 1.0
    color 1.0 1.0 1.0
    cutOffAngle 0.785
}
```

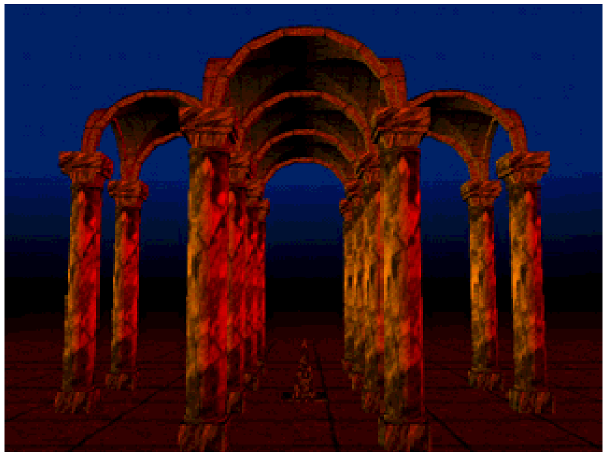
# *Syntax: SpotLight*

- 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

```
SpotLight {
    . . .
    cutOffAngle 0.785
    beamWidth   0.52
}
```

# *Example*



[ temple.wrl ]

# *Summary*

- 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*

- 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

# ***Using the background components***

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

- The sky sphere and ground hemisphere are shaded with a color gradient

- The panorama box is texture mapped with six images

# *Using the background components*

- Transparent parts of the ground hemisphere reveal the sky sphere
- Transparent parts of the panorama box reveal the ground and sky
- 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

# *Syntax: Background*

- A **`Background`** node describes background colors
  - **`skyColor`** and **`skyAngle`** - sky gradation
  - **`groundColor`** and **`groundAngle`** - ground gradation

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

# *Using sky angles and colors*

- The first sky color is at the north pole
- The remaining sky colors are at given sky angles
  - The maximum angle is 180 degrees = 3.1415 radians
- The last color smears on down to the south pole

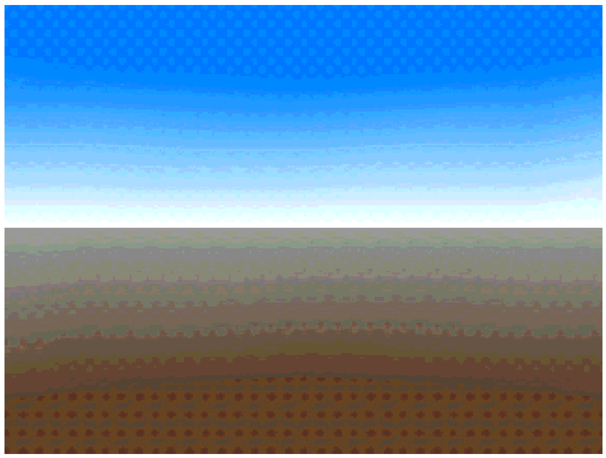
# *Using ground angles and colors*

- The first ground color is at the south pole
- The remaining ground colors are at given ground angles
  - The maximum angle is 90 degrees = 1.5708 radians
- After the last color, the rest of the hemisphere is transparent

# *Background example code*

```
Background {
    skyColor [
        0.0 0.2 0.7,
        0.0 0.5 1.0,
        1.0 1.0 1.0
    ]
    skyAngle [ 1.309, 1.571 ]
    groundColor [
        0.1 0.10 0.0,
        0.4 0.25 0.2,
        0.6 0.60 0.6,
    ]
    groundAngle [ 1.309, 1.571 ]
}
```

# *Background example*



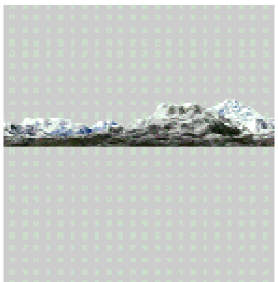
[ back.wrl ]

# *Syntax: Background*

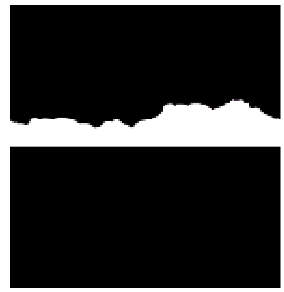
- A **`Background`** node describes background images
  - **`frontUrl`**, etc - texture image URLs for box

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

# ***Background image example***



a. Color portion of mountains texture

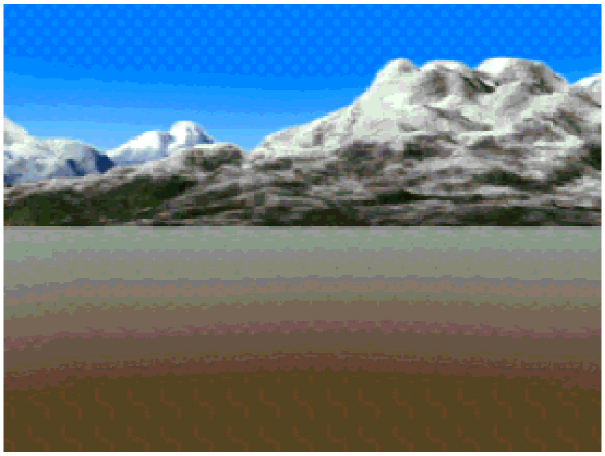


b. Transparency portion of mountains texture

# *Background image example code*

```
Background {
    skyColor [
        0.0 0.2 0.7,
        0.0 0.5 1.0,
        1.0 1.0 1.0
    ]
    skyAngle [ 1.309, 1.571 ]
    groundColor [
        0.1 0.10 0.0,
        0.4 0.25 0.2,
        0.6 0.60 0.6,
    ]
    groundAngle [ 1.309, 1.571 ]
    frontUrl "mountns.png"
    backUrl  "mountns.png"
    leftUrl  "mountns.png"
    rightUrl "mountns.png"
    # no top or bottom images
}
```

# *Background image example*



[ back2.wrl ]

# *Summary*

- Backgrounds describe:
  - Ground and sky color gradients on ground hemisphere and sky sphere

  - Panorama images on a panorama box

- The viewer can look around, but never get closer to the background

# Adding fog

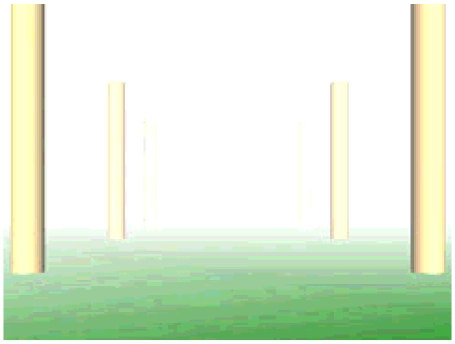
# *Motivation*

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

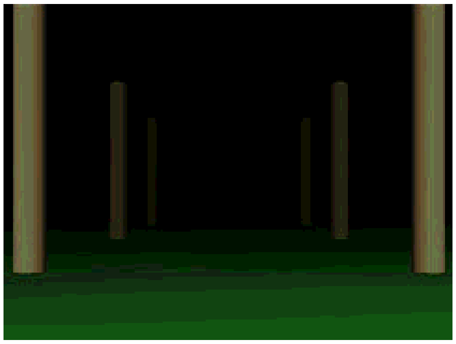
- The further the viewer can see, the more you have to model and draw

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

# *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

# *Selecting a fog color*

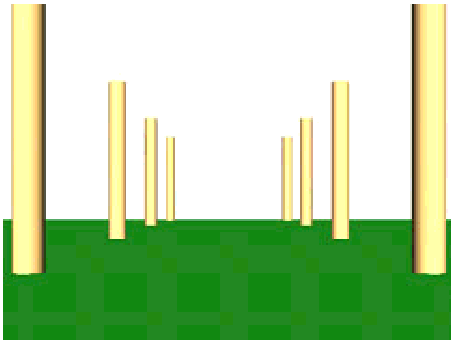
- Fog has a *fog color*
  - White is typical, but black, red, etc. also possible
- *Shapes* are faded to the fog color with distance
- The background is unaffected
  - For the best effect, make the background the fog color

# *Syntax: Fog*

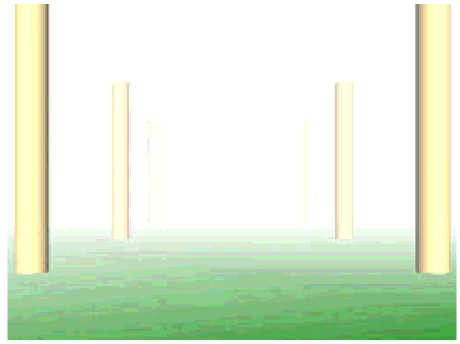
- A **`Fog`** node creates colored fog
  - **`color`** - fog color
  - **`fogType`** - **`LINEAR`** or **`EXPONENTIAL`**
  - **`visibilityRange`** - maximum visibility limit

```
Fog {
    color 1.0 1.0 1.0
    fogType "LINEAR"
    visibilityRange 10.0
}
```

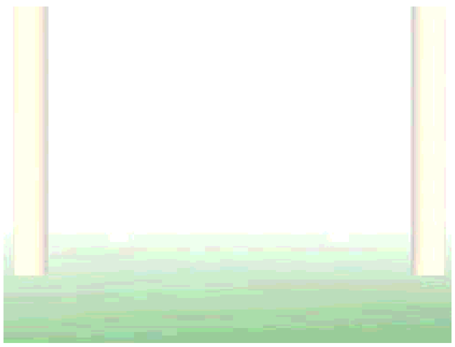
# *Several fog samples*



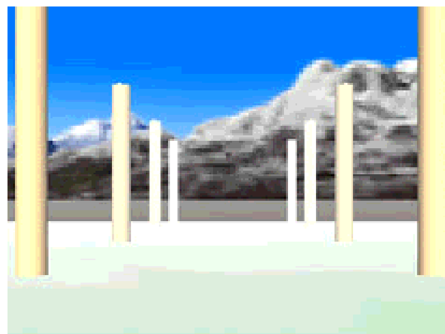
[ fog1.wrl ]
a. No fog



[ fog2.wrl ]
b. Linear fog, visibility range 30.0



[ fog3.wrl ]
c. Exponential fog, visibility range 30.0



[ fog5.wrl ]
c. Linear fog with a background (don’t do this!)

# *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

# Adding sound

# *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

```
Sound {
    source AudioClip {
        url "myfile.wav"
        pitch 1.0
        startTime 0.0
        stopTime  0.0
        loop FALSE
    }
}
```

# *Syntax: MovieTexture*

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

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

# *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

# *Syntax: Sound*

- A **`Sound`** node describes a sound emitter
  - **`source`** - **`AudioClip`** or **`MovieTexture`** node
  - **`location`** and **`direction`** - emitter placement

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

# *Syntax: Sound*

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

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

# *Syntax: Sound*

- A **`Sound`** node describes a sound emitter
  - **`minFront`**, **`minBack`** **-** inner ellipsoid
  - **`maxFront`**, **`maxBack`** **-** outer ellipsoid

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

# *Setting the sound range*

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

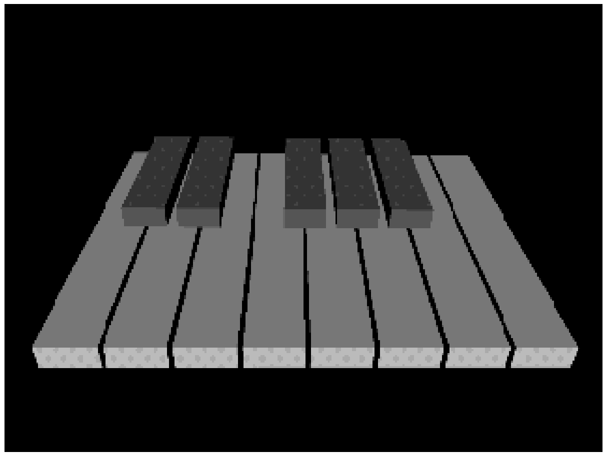
- **`AudioClip`** node:
  - **`loop FALSE`**
  - Set **`startTime`** from a sensor node

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

# *Triggered sound example code*

```
Group {
    children [
        Shape { . . . }
        DEF C4 TouchSensor { }
        Sound {
            source DEF PitchC4 AudioClip {
                url "tone1.wav"
                pitch 1.0
            }
            maxFront 100.0
            maxBack 100.0
        }
    ]
}
ROUTE C4.touchTime TO PitchC4.set_startTime
```

# *Triggered sound example*



[ kbd.wrl ]

# *Creating continuous localized sounds*

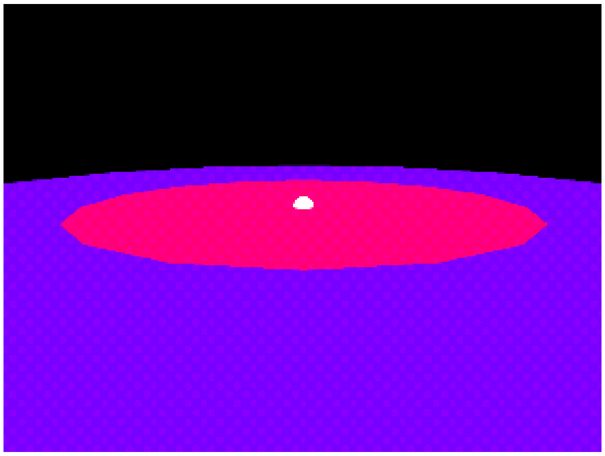
- **`AudioClip`** node:
  - **`loop TRUE`**
  - **`startTime 0.0`** (default)
  - **`stopTime 0.0`** (default)

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

# *Continuous localized sound example code*

```
Sound {
    source AudioClip {
        url "willow1.wav"
        loop TRUE
        startTime 1.0
        stopTime 0.0
    }
    minFront 5.0
    minBack  5.0
    maxFront 10.0
    maxBack  10.0
}
Transform {
    translation 0.0 -1.65 0.0
    children [
        Inline { url "sndmark.wrl" }
    ]
}
```

# *Continuous localized sound example*



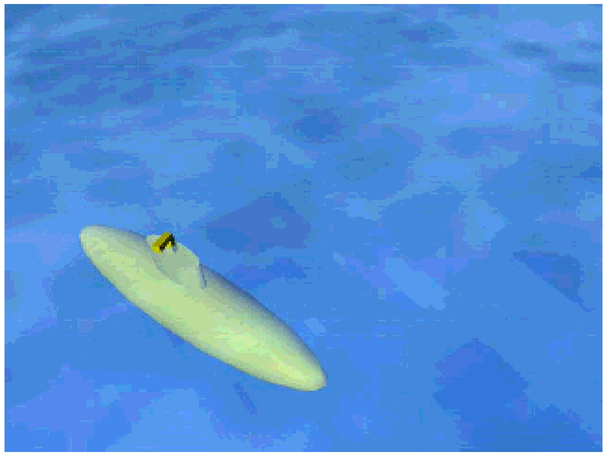
[ ambient.wrl ]

# *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)

# *Multiple sounds example*



[ subworld.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

# Controlling the viewpoint

# *Motivation*

- By default, the viewer enters a world at (0.0, 0.0, 10.0)
- You can provide your own preferred view points
  - Select the entry point position
  - Select favorite views for the viewer
  - Name the views for a browser menu

# *Creating viewpoints*

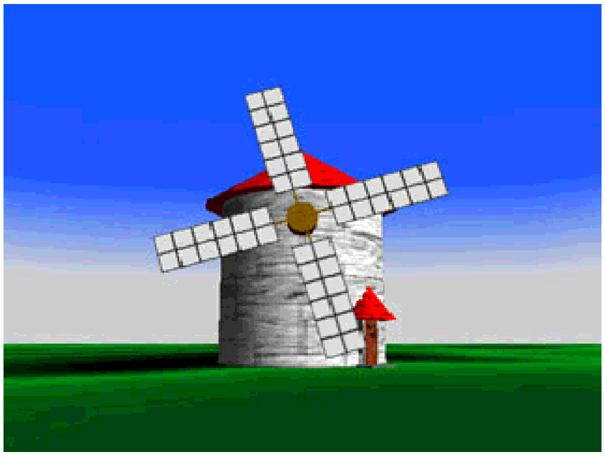
- Viewpoints specify a desired location, an orientation, and a camera field of view lens angle
- Viewpoints can be transformed using a **`Transform`** node
- The first viewpoint found in a file is the entry point

# *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"
}
```

# *Multiple viewpoints example*



[ windmill.wrl ]

# *Summary*

- Specify favorite viewpoints in **`Viewpoint`** nodes
- The first viewpoint in the file is the entry viewpoint

# Controlling navigation

# *Motivation*

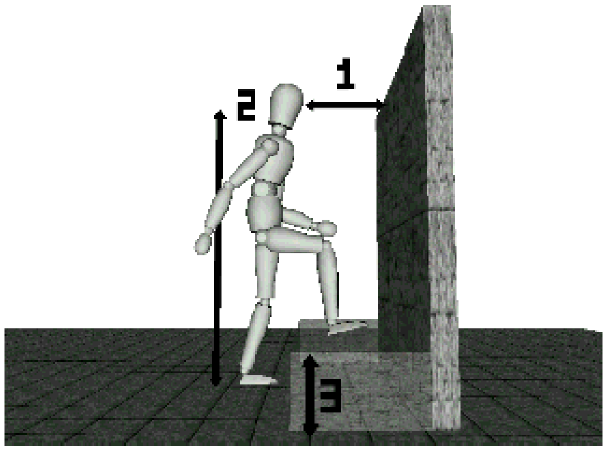
- Different types of worlds require different styles of navigation
  - Walk through a dungeon
  - Fly through a cloud world
  - Examine shapes in a CAD application
- You can select the navigation type
- You can describe the size and speed of the viewer's *avatar*

# *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 the headlight*

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

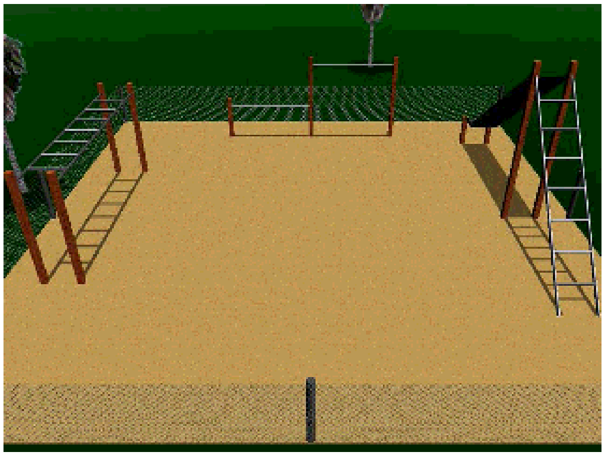
- You can turn this headlight on and off
  - Most browsers provide a menu option to control the headlight
  - You can also control the headlight with the **`NavigationInfo`** node

# *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

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

# NavigationInfo example



[ playyard.wrl ]

# *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

# Sensing the viewer

# *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

# *Sensing the viewer*

- 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

# *Syntax: VisibilitySensor*

- A **`VisibilitySensor`** node senses if the viewer sees or stops seeing 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

```
DEF VisSense VisibilitySensor {
    center 0.0 0.0 0.0
    size   14.0 14.0 14.0
}
ROUTE VisSense.enterTime TO Clock.set_startTime
```

# *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

```
DEF ProxSense ProximitySensor {
    center 0.0 0.0 0.0
    size   14.0 14.0 14.0
}
ROUTE ProxSense.enterTime TO Clock.set_startTime
```

# *Syntax: ProximitySensor*

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

```
DEF ProxSense ProximitySensor { . . . }

ROUTE ProxSense.position_changed TO PetRobotFollower.set_
```

# *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

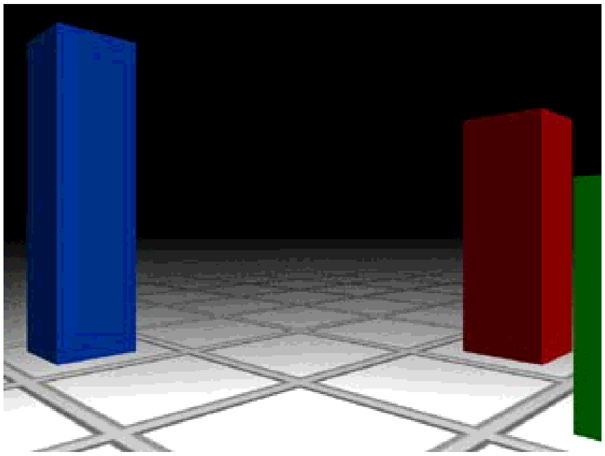
- A collision group with a proxy shape, but no children, creates an invisible collidable shape
  - Windows and invisible railings
  - Invisible world limits

# *Syntax: Collision*

- A **`Collision`** grouping node senses if the viewer collides with group shapes
  - **`collide`** - enable/disable sensor
  - **`proxy`** - simple shape to sense instead of children
  - **`children`** - children to sense
  - **`collideTime`** - sends time on collision

```
DEF Collide Collision {
    collide TRUE
    proxy Shape { geometry Box { . . . } }
    children [ . . . ]
}
ROUTE Collide.collideTime TO OuchSound.set_startTime
```

# *Proximity sensors and collision groups example*



[ prox2.wrl ]

# *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

# *Summary*

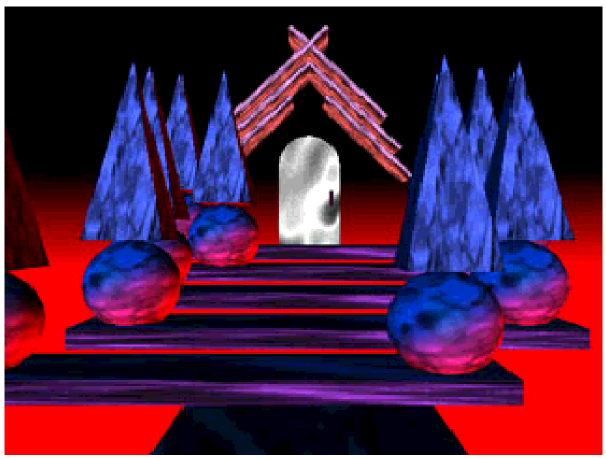
- A `Collision` grouping node checks if the viewer has run into a shape
  - The shapes are defined by the group's children or a proxy

  - Collision time is sent on contact

# Summary examples

# *A doorway*

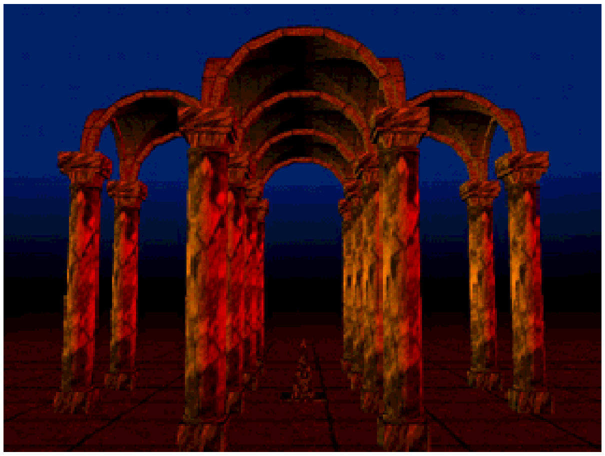
- A set of **`ImageTexture`** nodes add marble textures
- Lighting nodes create dramatic lighting
- A **`Fog`** node fades distant shapes
- A **`ProximitySensor`** node controls animation



[ doorway.wrl ]

# ***A mysterious temple***

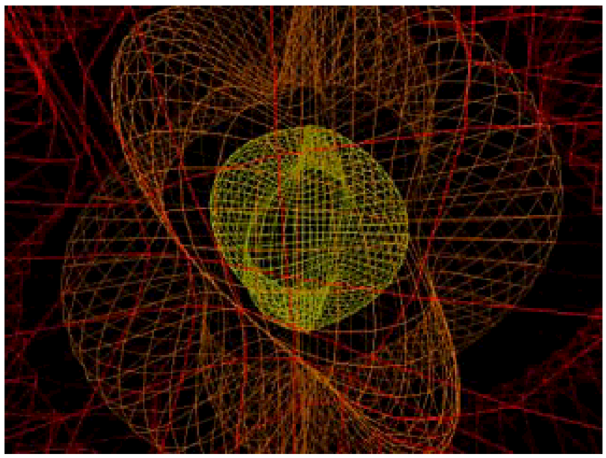
- A **`Background`** node creates a sky gradient
- A **`Sound`** node creates a spatialized sound effect
- A set of **`Viewpoint`** nodes provide standard views



[ temple.wrl ]

# *Depth-cueing using fog*

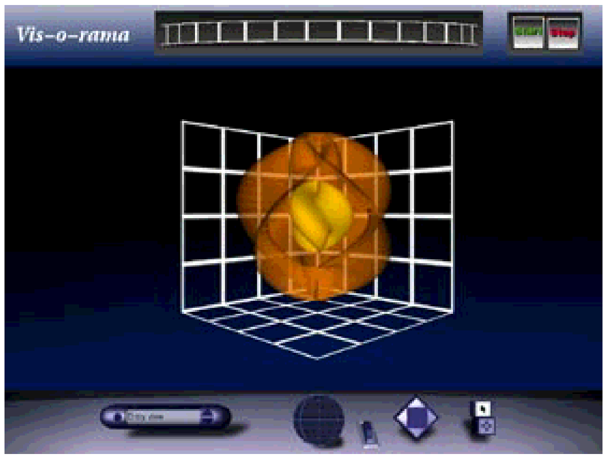
- Multiple **`IndexedLineSet`** nodes create wireframe isosurfaces
- A **`Fog`** node with black fog fades out distant lines for depth-cueing



[ isoline.wrl ]

# *A heads-up display*

- A **`ProximitySensor`** node tracks the viewer and moves a panel at each step
- The panel contains shapes and sensors to control the content




[ hud.wrl ]

# Controlling detail

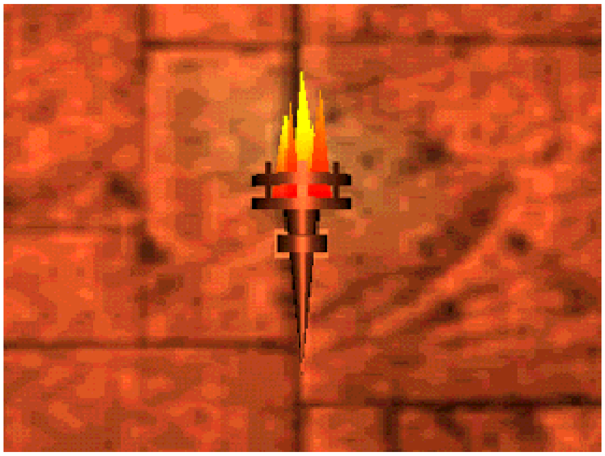
# *Motivation*

- 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

# *Example*



[ prox1.wrl ]

# *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 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

# *Syntax: LOD*

- An **`LOD`** grouping node creates a group of shapes describing different levels (versions) of the same shape
  - **`center`** - the center of the shape
  - **`range`** - a list of level switch ranges
  - **`level`** - a list of shape levels

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

# *Choosing detail ranges*

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

```
range [ 5.0, 10.0 ]
```

shape center —— **5.0** —— **10.0** ——

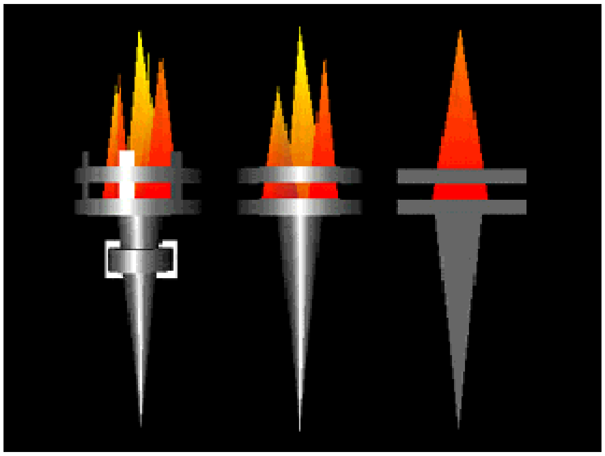
| Viewer <= 5.0 | Viewer <= 10.0 | Viewer > 10.0 |
|---|---|---|
| Show 1st level | Show 2nd level | Show 3rd level |

# *Optimizing a shape*

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

- Lower detail levels should use simpler geometry, fewer textures, and no text

# *Detail levels example*

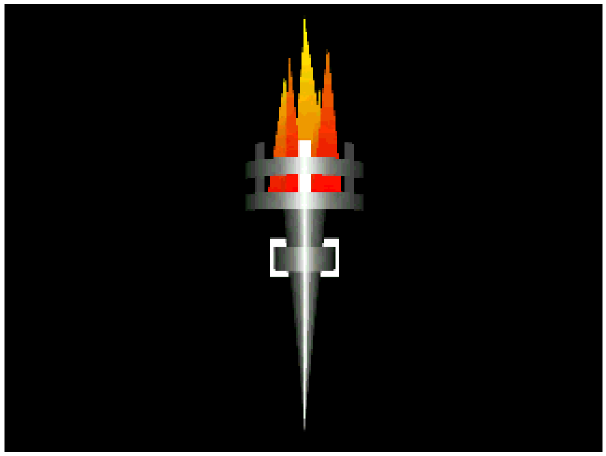


[ torches3.wrl ]

# *Level of detail example code*

```
LOD {
    center 0.0 0.0 0.0
    range [ 7.0, 10.0 ]
    level [
        Inline { url "torch1.wrl" }
        Inline { url "torch2.wrl" }
        Inline { url "torch3.wrl" }
    ]
}
```

# *Level of detail example*



[ torches.wrl ]

# *Summary*

- Increase performance by making multiple levels of shapes
  - High detail for close up viewing
  - Lower detail for more distant viewing

- Group the levels in an **`LOD`** node
  - Ordered from high detail to low detail
  - Ranges to select switching distances

# Introducing script use

Introducing script use

# *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

# *A word about scripting languages*

- The VRML specification doesn't *require* scripting language support
  - Most browsers support JavaScript et al
  - Many browsers support Java

- VRMLScript = JavaScript = ECMAScript
  - JavaScript is nothing like Java
  - *VRMLScript* is Cosmo Software's limited JavaScript
  - The ISO VRML specification calls for *ECMAScript*, the ECMA version of JavaScript

# *Syntax: Script*

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

```
DEF Bouncer Script {
    url "bouncer.class"
or...
    url "bouncer.js"
or...
    url "javascript: ..."
or...
    url "vrmlscript: ..."
}
```

# *Defining the program script interface*

- A **`Script`** node also declares the program script interface
  - **`field`**, **`eventIn`**, and **`eventOut`** - inputs and outputs
    - Each has a name and data type
    - Fields have an initial value

```
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed
}
```

# *Data types*

| **Data type** | **Meaning** |
|---|---|
| `SFBool` | Boolean, true or false value |
| `SFColor`, `MFColor` | RGB color value |
| `SFFloat`, `MFFloat` | Floating point value |
| `SFImage` | Image value |
| `SFInt32`, `MFInt32` | Integer value |
| `SFNode`, `MFNode` | Node value |

# *Data types*

| Data type | Meaning |
|---|---|
| `SFRotation`, `MFRotation` | Rotation value |
| `SFString`, `MFString` | Text string value |
| `SFTime` | Time value |
| `SFVec2f`, `MFVec2f` | XY floating point value |
| `SFVec3f`, `MFVec3f` | XYZ floating point value |

# *Program script example code*

```
DEF Clock TimeSensor { . . . }

DEF Ball Transform { . . . }

DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed
    url "vrmlscript: . . ."
}

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

Introducing script use

# *Program script example*

[ 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*

- 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*, *VRMLscript*, and other languages
  - JavaScript is easier to program
  - Java is more powerful
  - VRMLscript is essentially JavaScript

# *Declaring a program script interface*

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

```
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed
    url "javascript: . . ."
or...
    url "vrmlscript: . . ."
}
```

# *Initializing a program script*

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

```
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

```
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 ) {
    . . .
}
```

# *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
  - Multi-value data types are arrays

```
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
  - Multi-value data types are arrays

```
lastval = value_changed[0];  // get last event

value_changed[0] = newval;   // send new event
```

# *JavaScript script example code*

- Create a *Bouncing ball interpolator* that computes a gravity-like vertical bouncing motion from a fractional time input

- Nodes needed:

```
DEF Ball Transform { . . . }
DEF Clock TimeSensor { . . . }
DEF Bouncer Script { . . . }
```

# *JavaScript script example code*

- Script fields needed:
  - Bounce height

```
DEF Bouncer Script {
    field SFFloat bounceHeight 3.0
    . . .
}
```

# *JavaScript script example code*

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

```
DEF Bouncer Script {
    . . .
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed
    . . .
}
```

# *JavaScript script example code*

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

# *JavaScript script example code*

- Event processing actions needed:
  - **`set_fraction`** eventIn function

```
DEF Bouncer Script {
    . . .
    url "vrmlscript:
        function set_fraction( frac, tm ) {
            . . .
        }"
}
```

# JavaScript script example code

- 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 script example code*

```
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    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;  // X
            value_changed[1] = y;    // Y
            value_changed[2] = 0.0;  // Z
        }"
}
```

# *JavaScript script example code*

- 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
```

# *JavaScript script example code*

```
DEF Ball Transform {
    children [
        Shape {
            appearance Appearance {
                material Material {
                    ambientIntensity 0.5
                    diffuseColor 1.0 1.0 1.0
                    specularColor 0.7 0.7 0.7
                    shininess 0.4
                }
                texture ImageTexture { url "beach.jpg" }
                textureTransform TextureTransform { scale 2.
            }
            geometry Sphere { }
        }
    ]
}
DEF Clock TimeSensor {
    cycleInterval 2.0
    startTime 1.0
    stopTime 0.0
    loop TRUE
}
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    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;  // X
            value_changed[1] = y;    // Y
            value_changed[2] = 0.0;  // Z
        }"
}
ROUTE Clock.fraction_changed TO Bouncer.set_fraction
```

```
ROUTE Bouncer.value_changed  TO Ball.set_translation
```

# *JavaScript script example*

[ bounce1.wrl ]

# *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 script turns on at 1st touch, off at 2nd
  - A **`TouchSensor`** node can supply touch events

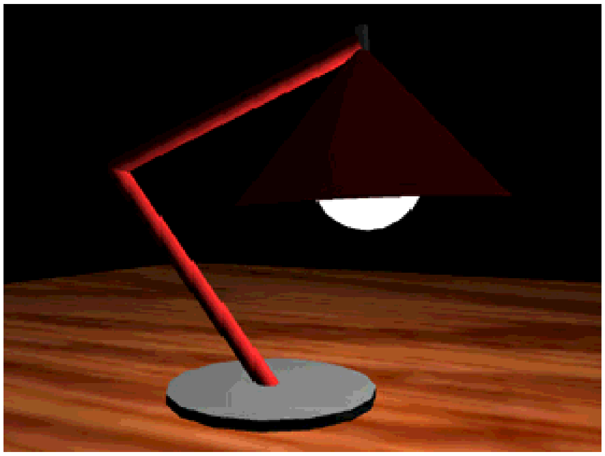
```
DEF Toggle Script {
    field    SFBool on TRUE
    eventIn  SFBool set_active
    eventOut SFBool on_changed
    url "vrmlscript:
        function set_active( b, ts ) {
            // ignore button releases
            if ( b  == FALSE ) return;

            // toggle on button presses
            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

```
DEF LightSwitch TouchSensor { }
DEF LampLight SpotLight { . . . }

DEF Toggle Script { . . . }

ROUTE LightSwitch.isActive TO Toggle.set_active
ROUTE Toggle.on_changed    TO LampLight.set_on
```

# *Using a toggle switch*



[ lamp2a.wrl ]

# *Building a color selector*

- The turns lamp 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

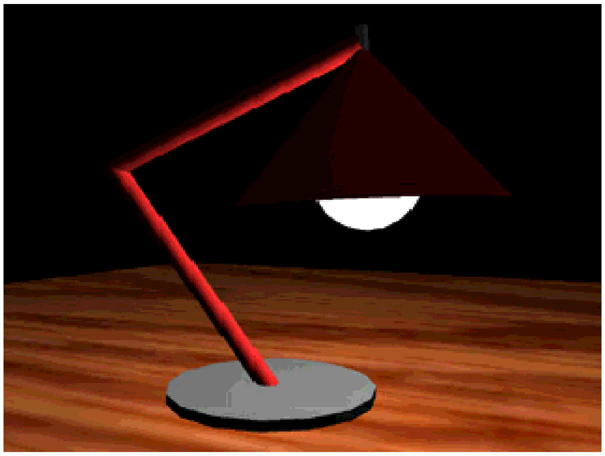
```
DEF ColorSelector Script {
    field    SFColor onColor  1.0 1.0 1.0
    field    SFColor offColor 0.0 0.0 0.0
    eventIn  SFBool  set_selection
    eventOut SFColor color_changed
    url "vrmlscript:
        function set_selection( b, ts ) {
            if ( b == TRUE )
                color_changed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

# Using a color selector

- Use the color selector to change the lamp bulb color

```
DEF LightSwitch TouchSensor { }
DEF LampLight SpotLight { . . . }
DEF BulbMaterial Material { . . . }

DEF Toggle Script { . . . }
DEF ColorSelector Script { . . . }

ROUTE LightSwitch.isActive TO Toggle.set_active
ROUTE Toggle.on_changed    TO LampLight.set_on
ROUTE Toggle.on_changed    TO ColorSelector.set_selectior
ROUTE ColorSelector.color_changed TO BulbMaterial.set_emi
```

# *Using a color selector*



[ lamp2.wrl ]

# *Summary*

- The **`initialize`** and **`shutdown`** functions are called at load and unload
- An eventIn function is called when an event is received
- Functions can get field values and send event outputs

# Writing program scripts with Java

# Motivation

- Compared to JavaScript/VRMLscript, Java enables:
  - Better modularity
  - Better data structures
  - Potential for faster execution
  - Access to the network

- For simple tasks, use JavaScript/VRMLscript
- 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

```
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed

    url "bounce2.class"
}
```

# *Importing packages for the Java class*

- The program script file must import the VRML packages:
  - Supplied by the VRML browser vendor

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

# *Creating the Java class*

- The program script must define a public class that extends the **`Script`** class

```
public class bounce2
    extends Script
{
    . . .
}
```

# *Initializing a program script*

- The optional **`initialize`** method is called when the script is loaded

```
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

```
public void shutdown ( ) {
    . . .
}
```

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

# *Responding to events*

- The **`processEvent`** method is called each time an event is received, passing an **`Event`** object containing the event's
  - value
  - time stamp

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

# *Accessing fields from Java*

- Each interface field can be read and written
  - Call **`getField`** to get a field object

    ```
    obj = (SFFloat) getField( "bounceHeight" );
    ```

  - Call **`getValue`** to get a field value

    ```
    lastval = obj.getValue( );
    ```

  - Call **`setValue`** to set a field value

    ```
    obj.setValue( newval );
    ```

# *Accessing eventOuts from Java*

- Each interface eventOut can be read and written
  - Call **`getEventOut`** to get an eventOut object

```
obj = (SFVec3f) getEventOut( "value_changed" );
```

  - Call **`getValue`** to get the last event sent

```
lastval = obj.getValue( );
```

  - Call **`setValue`** to send an event

```
obj.setValue( newval );
```

# *Java script example code*

- Create a *Bouncing ball interpolator* that computes a gravity-like vertical bouncing motion from a fractional time input

- Nodes needed:

```
DEF Ball Transform { . . . }
DEF Clock TimeSensor { . . . }
DEF Bouncer Script { . . . }
```

# *Java script example code*

- Give it the same interface as the JavaScript example

```
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed

    url "bounce2.class"
}
```

# *Java script example code*

- Imports and class definition needed:

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

public class bounce2
    extends Script
{
    . . .
}
```

# *Java script example code*

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

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

# *Java script example code*

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

```
public void initialize( )
{
    SFFloat obj = (SFFloat) getField( "bounceHeight" );
    bounceHeight = (float) obj.getValue( );
    value_changedObj = (SFVec3f) getEventOut( "value_char
}
```

# *Java script example code*

- Shutdown actions needed:
    - None - all work done in `processEvent` method

# *Java script example code*

- Event processing actions needed:
  - **`processEvent`** event method

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

# *Java script example code*

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

# *Java script example code*

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

    float y = (float)(4.0 * bounceHeight * frac * (1.0 - fra

    float[] changed = new float[3];
    changed[0] = (float) 0.0;
    changed[1] = y;
    changed[2] = (float) 0.0;
    value_changedObj.setValue( changed );
}
```

# Java script example code

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

public class bounce2
    extends Script
{
    private float bounceHeight;
    private SFVec3f value_changedObj;

    public void initialize( )
    {
        // Get the fields and eventOut
        SFFloat floatObj = (SFFloat) getField( "bounceHeight
        bounceHeight     = (float)   floatObj.getValue( );
        value_changedObj = (SFVec3f) getEventOut( "value_cha
    }

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

        float y = (float)(4.0 * bounceHeight * frac * (1.0 -

        float[] changed = new float[3];
        changed[0] = (float)0.0;
        changed[1] = y;
        changed[2] = (float)0.0;
        value_changedObj.setValue( changed );
    }
}
```

# *Java script example code*

- 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
```

# *Java script example code*

```
DEF Ball Transform {
    children [
        Shape {
            appearance Appearance {
                material Material {
                    ambientIntensity 0.5
                    diffuseColor 1.0 1.0 1.0
                    specularColor 0.7 0.7 0.7
                    shininess 0.4
                }
                texture ImageTexture { url "beach.jpg" }
                textureTransform TextureTransform { scale 2.
            }
            geometry Sphere { }
        }
    ]
}
DEF Clock TimeSensor {
    cycleInterval 2.0
    startTime 1.0
    stopTime 0.0
    loop TRUE
}
DEF Bouncer Script {
    field    SFFloat bounceHeight 3.0
    eventIn  SFFloat set_fraction
    eventOut SFVec3f value_changed

    url "bounce2.class"
}
ROUTE Clock.fraction_changed TO Bouncer.set_fraction
ROUTE Bouncer.value_changed  TO Ball.set_translation
```

# *Java script example*

[ bounce2.wrl ]

# *Summary*

- The `initialize` and `shutdown` methods are called at load and unload
- The `processEvent` method is called when an event is received
- Methods can get field values and send event outputs

# Accessing the browser from JavaScript and Java

# *Motivation*

- You can create scripts that request the VRML browser to:
  - Report it’s name and version number
  - Return its current frame rate
  - Add and delete routes
  - Load VRML from a URL or text string
  - Replace the current world

# *Using the Browser object*

- To control the browser, use the **`Browser`** object
  - Available in both Java and JavaScript program scripts

- To call a **`Browser`** function type

```
Browser.function
```

where ***`function`*** is the name of a browser function

# *Using Browser information functions*

- Browser information functions:
  - ***`string`*** **`Browser.getName( )`**
    - Get the name of the VRML browser
  - ***`string`*** **`Browser.getVersion( )`**
    - Get the VRML browser's version
  - ***`string`*** **`Browser.getWorldURL( )`**
    - Get the URL of the current world
  - ***`void`*** **`Browser.setDescription(`** ***`string`*** **`text )`**
    - Set the world's description

# *Using Browser information functions*

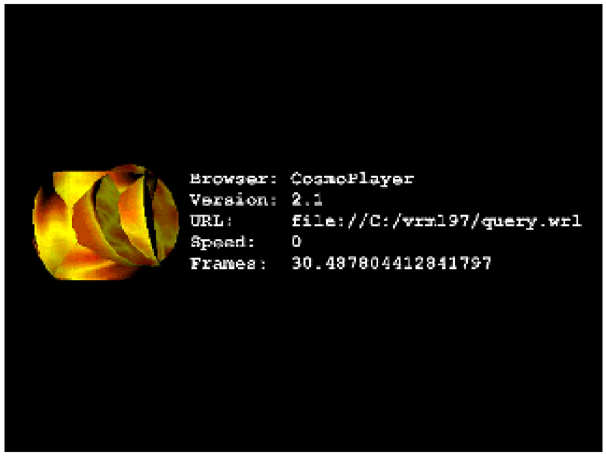
- Browser information functions:
  - ***`numeric`*** **`Browser.getCurrentSpeed( )`**
    - Get the VRML browser’s drawing speed
  - ***`numeric`*** **`Browser.getCurrentFrameRate( )`**
    - Get the VRML browser’s frame rate

# *Browser information functions example code*

- Query browser information and send it as a string
- Use a **`Text`** node to display the string

```
DEF Introspect Script {
    eventIn  SFTime   trigger
    eventOut MFString message
    url "vrmlscript:
        function update( ) {
            message.length = 5;
            message[0] = 'Browser: ' + Browser.getName( );
            message[1] = 'Version: ' + Browser.getVersion( )
            message[2] = 'URL:     ' + Browser.getWorldURL(
            message[3] = 'Speed:   ' + Browser.getCurrentSpe
            message[4] = 'Frames:  ' + Browser.getCurrentFra
        }
        function initialize( ) {
            update( );
        }
        function trigger( t, ts ) {
            update( );
        }"
}
```

# *Browser information functions example*




[ query.wrl ]

# *Using Browser route functions*

- Browser route functions:
  - ***void* Browser.addRoute(**
    ***node* fromNode, *string* fromOut,**
    ***node* toNode, *string* toIn )**
    - Create a route between two nodes
  - ***void* Browser.deleteRoute(**
    ***node* fromNode, *string* fromOut,**
    ***node* toNode, *string* toIn )**
    - Remove a route between two nodes

# *Using Browser content creation functions*

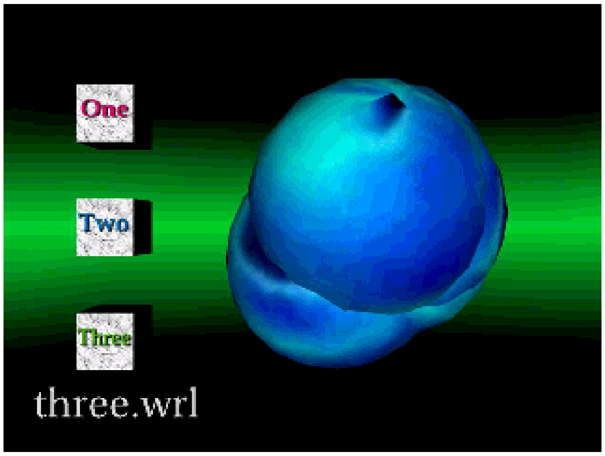
- Browser content creation functions:
  - ***void*** **`Browser.replaceWorld(`**
    **`    `*`node`*` newNode )`**
    - Replace the world with a new node
  - ***node*** **`Browser.createVrmlFromString(`**
    **`    `*`string`*` text )`**
    - Create a new node from VRML text
  - ***void*** **`Browser.createVrmlFromURL(`**
    **`    `*`string`*` url,`**
    **`    `*`node`*` notifyNode, `*`string`*` notifyIn )`**
    - Load VRML text from a URL, then notify a node by sending the loaded node to it’s notify eventIn

# *Browser content creation functions example code*

- Receive a URL on an input, load it, and output the results
- Use a **Group** node to hold the results

```
DEF Loader Script {
    field    SFNode   myself USE Loader
    field    MFString lastUrl ""
    eventIn  MFString loadUrl
    eventIn  MFNode   vrmlLoaded
    eventOut MFNode   node_changed
    eventOut MFString string_changed
    url "vrmlscript:
        function loadUrl( str, ts ) {
            lastUrl = str;
            Browser.createVrmlFromURL( str, myself, ’vrmlLoa
            string_changed[0] = ’Loading...’;
        }
        function vrmlLoaded( nd, ts ) {
            node_changed = nd;
            string_changed[0] = lastUrl[0];
        }"
}
```

# *Browser content creation functions example*




[ loader.wrl ]

# *Summary*

- Scripts can access the VRML browser to:
  - Get information including the browser name, version, speed, and current URL
  - Add and delete routes
  - Load VRML content into the current world, or replace it

# 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*

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

```
PROTO Robot [
    field SFColor eyeColor  1.0 0.0 0.0
    . . .
] {
    . . .
}
```

# *Defining prototype bodies*

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

```
PROTO Robot [
    . . .
] {
    Transform {
        children [ . . . ]
    }
}
```

# *Using new nodes*

- Once defined, a prototyped node can be used like any other node

```
Robot {
    eyeColor   0.0 1.0 0.0
    metalColor 0.6 0.6 0.8
    rodColor   1.0 1.0 0.0
}
```

# *Using prototypes*

- The **`PROTO`** interface declares items you can use within the body
  - A **`PROTO`** is like a JavaScript function
  - An interface item is like a JavaScript function argument

- For example:
  - Create a **`PROTO`** for a **`Robot`** node
  - Give the **`Robot`** node an **`eyeColor`** field
  - Use that **`eyeColor`** in the **`PROTO`** body to set the color of each robot eye

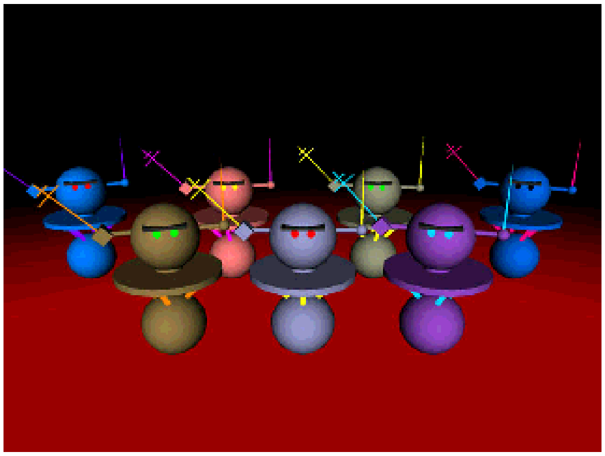
# *Syntax: IS*

- The **IS** syntax uses an interface item (argument) in the **PROTO** body
  - Like an assignment statement
  - Assigns an interface field or eventIn to a body
  - Assigns a body eventOut to interface

# IS example code

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

# *IS example*



[ robot.wrl ]

# *Using IS*

| | May `IS` to . . . | | | |
|---|---|---|---|---|
| Interface | Fields | Exposed fields | EventIns | EventOuts |
| Fields | yes | yes | no | no |
| Exposed fields | no | yes | no | no |
| EventIns | no | yes | yes | no |
| EventOuts | no | yes | no | yes |

# *Using IS*

- For example, you may say that:
  - A body node's field **`IS`** an interface field
    - Such as the Robot's eye color

  - A body node's eventIn **`IS`** an interface eventIn
    - Such as a Robot's **`turnOn`** event used to set a **`TimeSensor set_startTime`**

  - A body node's eventOut **`IS`** an interface eventOut
    - Such as a Robot's **`blasterFire`** eventOut from an AI script!

# *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

# *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

# *Prototype example code*

- 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

# *Prototype example code*

- Fields needed:
  - Bounce height
  - Bounce time

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

# *Prototype example code*

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

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

Creating new node types

# *Prototype example*

[ bounce3.wrl ]

# *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

# *Syntax: EXTERNPROTO*

- Prototypes are typically in a separate *external* file, referenced by an **EXTERNPROTO**
  - *name*, *fields*, *events* - as from **PROTO**, minus initial values
  - *url* - the URL of the prototype file
  - *#name* - name of **PROTO** in file

```
EXTERNPROTO BouncingBall [
    field SFFloat bounceHeight
    field SFTime  bounceTime
] "bounce4.wrl#BouncingBall"
```

# Summary

- **PROTO** declares a new node type and defines its node body
- **EXTERNPROTO** declares a new node type, specified by URL

# Providing information about your world

# *Motivation*

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

# *Syntax: WorldInfo*

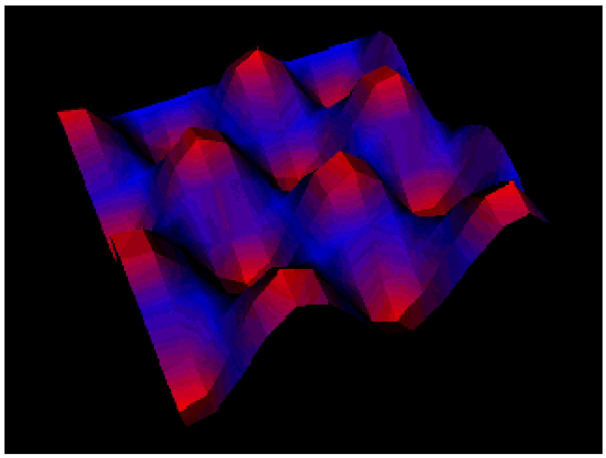
- A **`WorldInfo`** node provides title and description information for your world
  - **`title`** - the name for your world
  - **`info`** - any additional information

```
WorldInfo {
    title "My Masterpiece"
    info  [ "Copyright (c) 1997 Me." ]
}
```

# Summary examples

# *An animated switch*

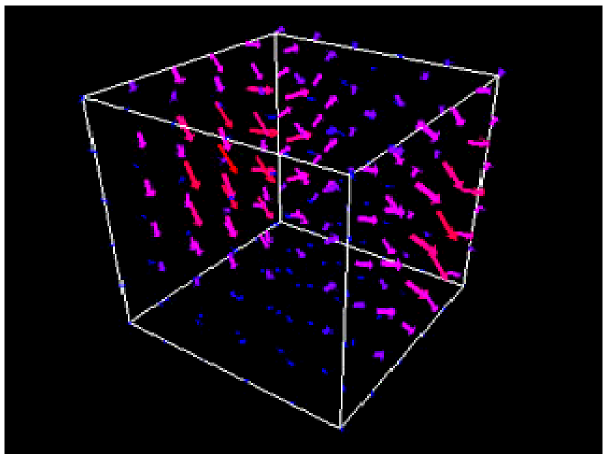
- A **`Switch`** node groups together a set of elevation grids
- A **`Script`** node converts fractional times to switch choices



[ animgrd.wrl ]

# *A vector node for vector fields*

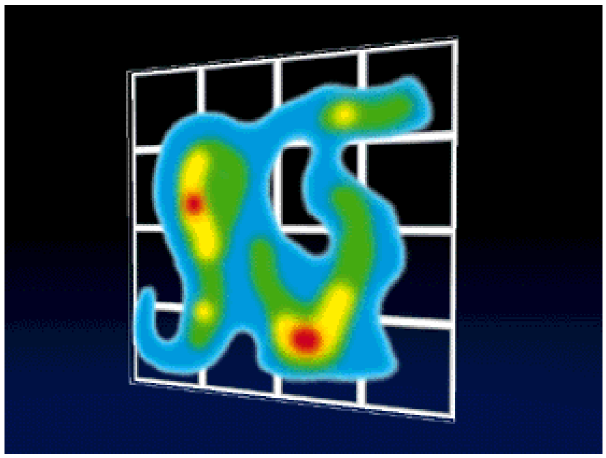
- A **`PROTO`** encapsulates a vector shape into a **`Vector`** node
- That node is used multiple times to create a vector field



[ vecfld1.wrl ]

# *An animated texture plane node*

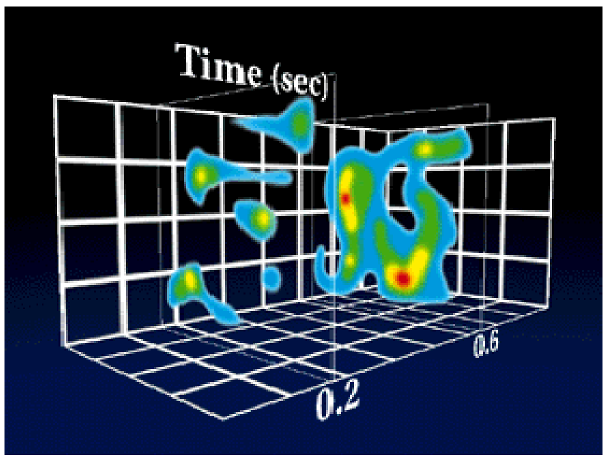
- A **`Script`** node selects a texture to map to a face
- A **`PROTO`** encapsulates the face shape, script, and routes to create a **`TexturePlane`** node type



[ texplane.wrl ]

# *A cutting plane node*

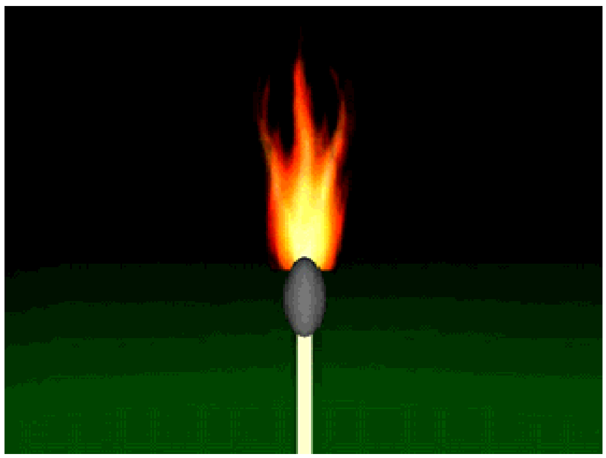
- A **`TexturePlane`** node creates textured face
- A **`PlaneSensor`** node slides the textured face
- A **`PROTO`** encapsulates the textured face, sensor, and translator script to create a **`SlidingPlane`** node




[ cutplane.wrl ]

# *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 ]

# *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 ]

# Miscellaneous extensions

# *Working groups*

- Several groups are working on VRML extensions
  - Color fidelity WG
  - Conformance WG
  - Database WG
  - External authoring interface WG
  - Human animation WG

# *Working groups*

- And more...
  - Keyboard input WG
  - Living worlds WG
  - Metaforms WG
  - Object-oriented WG
  - Universal media libraries WG
  - Widgets WG

# *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!

# *Using the external authoring interface*

- 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
  - Support is often incomplete or buggy

# Conclusion

# *Coverage*

- 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 ISO VRML 97 specification
  http://www.vrml.org/Specifications/

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

# *Where to find out more*

- Shameless plug for our VRML book...

    **`The VRML 2.0 Sourcebook`**
    by Andrea L. Ames, David R. Nadeau, and John L. Moreland
    published by John Wiley & Sons

# *Introduction to VRML 97*

*Thanks for coming!*

Dave Nadeau
San Diego Supercomputer Center
University of California at San Diego