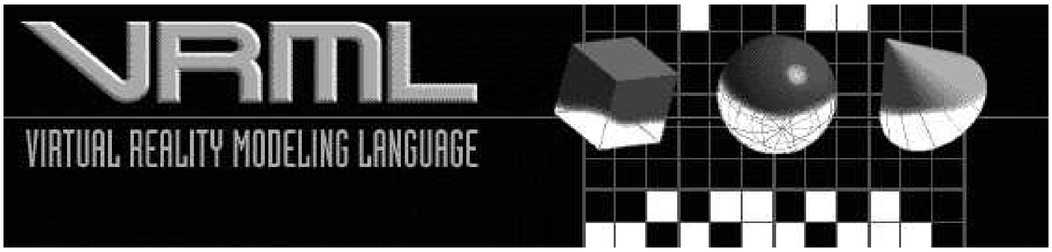

The Virtual Reality Modeling Language

Version 1.0C Specification

Now with Clarifications!

29-Jan-96

Gavin Bell, Silicon Graphics, Inc.
Anthony Parisi, Intervista Software
Mark Pesce, VRML List Moderator

This document is located at http://vag.vrml.org/vrml10c.html

Revision History

- First Draft - November 2, 1994
- Second Draft - May 8, 1995
- First 1.0 spec- May 26, 1995
- 1.0 clarifications - Jan 29, 1996

Table of Contents

Introduction

The **Virtual Reality Modeling Language** (VRML) is a language for describing multi-participant interactive simulations -- virtual worlds networked via the global Internet and hyper-linked with the World Wide Web. All aspects of virtual world display, interaction and internetworking can be specified using VRML. It is the intention of its designers that VRML become the standard language for interactive simulation within the World Wide Web.

The first version of VRML allows for the creation of virtual worlds with limited interactive behavior. These worlds can contain objects which have hyper-links to other worlds, HTML documents or other valid MIME types. When the user selects an object with a hyper-link, the appropriate MIME viewer is launched. When the user selects a link to a VRML document from within a correctly configured WWW browser, a VRML viewer is launched. Thus VRML viewers are the perfect companion applications to standard WWW browsers for navigating and visualizing the Web. Future versions of VRML will allow for richer behaviors, including animations, motion physics and real-time multi-user interaction.

This document specifies the features and syntax of Version 1.0 of VRML.

This VRML specification assumes that the reader has a fair amount of background in 3D graphics systems. Because this specification is intended only as a specification, not as a general introduction to 3D graphics concepts, some sections may be difficult to understand. There are at least three books on VRML that are now available, and at least three more in development. You are encouraged to survey the available books and select as many as you find useful.

If you'd like to learn enough 3D concepts to be able to write your own VRML browser, this book provides a good introduction to how to write 3D graphics programs. This is referred to as [FVDFH] below.

```
Computer Graphics: Principles and Practice
Foley, van Dam, Feiner, and Hughes
1990, Addison-Wesley Publishing Company
ISBN 0-201-12110-7
```

Of course, there will always be a lot of up-to-date information available on the web. Two good starting points for VRML information are:

- http://vrml.wired.org
- http://www.sdsc.edu/vrml

VRML Mission Statement

The history of the development of the Internet has had three distinct phases; first, the development of the TCP/IP infrastructure which allowed documents and data to be stored in a proximally independent way; that is, Internet provided a layer of abstraction between data sets and the hosts which manipulated them. While this abstraction was useful, it was also confusing; without any clear sense of "what went where", access to Internet was restricted to the class of sysops/net surfers who could maintain internal cognitive maps of the data space.

Next, Tim Berners-Lee's work at CERN, where he developed the hyper-media system known as **World Wide Web**, added another layer of abstraction to the existing structure. This abstraction provided an "addressing" scheme, a unique identifier (the Universal Resource Locator), which could tell anyone "where to go and how to get there" for any piece of data within the Web. While useful, it lacked dimensionality; there's no *there* there within the web, and the only type of navigation permissible (other than surfing) is by direct reference. In other words, I can only tell you how to get to the VRML Forum home page by saying, "http://www.wired.com/", which is not human-centered data. In fact, I need to make an effort to remember it at all. So, while the World Wide Web provides a retrieval mechanism to complement the existing storage mechanism, it leaves a lot to be desired, particularly for human beings.

Finally, we move to "perceptualized" Internetworks, where the data has been sensualized, that is, rendered sensually. If something is represented sensually, it is possible to make sense of it. VRML is an attempt (how successful, only time and effort will tell) to place humans at the center of the Internet, ordering its universe to our whims. In order to do that, the most important single element is a standard that defines the particularities of perception. Virtual Reality Modeling Language is that standard, designed to be a *universal description language for multi-participant simulations*.

These three phases, storage, retrieval, and perceptualization are analogous to the human process of consciousness, as expressed in terms of semantics and cognitive science. Events occur and are recorded (memory); inferences are drawn from memory (associations), and from sets of related events, maps of the universe are created (cognitive perception). What is important to remember is that the map is **not** the territory, and we should avoid becoming trapped in any single representation or world-view. Although we need to *design to avoid disorientation*, we should always push the envelope in the kinds of experience we can bring into manifestation!

This document is the living proof of the success of a process that was committed to being open and flexible, responsive to the needs of a growing Web community. Rather than re-invent the wheel, we have adapted an existing specification (Open Inventor) as the basis from which our own work can grow, saving years of design work and perhaps many mistakes. Now our real work can begin; that of rendering our noospheric space.

History

VRML was conceived in the spring of 1994 at the first annual World Wide Web Conference in Geneva, Switzerland. Tim Berners-Lee and Dave Raggett organized a Birds-of-a-Feather (BOF) session to discuss Virtual Reality interfaces to the World Wide Web. Several BOF attendees described projects already underway to build three dimensional graphical visualization tools which inter-operate with the Web. Attendees agreed on the need for these tools to have a common language for specifying 3D world description and WWW hyper-links -- an analog of HTML for virtual reality. The term Virtual Reality Markup Language (VRML) was coined, and the group resolved to begin specification work after the conference. The word 'Markup' was later changed to 'Modeling' to reflect the graphical nature of VRML.

Shortly after the Geneva BOF session, the www-vrml mailing list was created to discuss the development of a specification for the first version of VRML. The response to the list invitation was overwhelming: within a week, there were over a thousand members. After an initial settling-in period, list moderator Mark Pesce of Labyrinth Group announced his intention to have a draft version of the specification ready by the WWW Fall 1994 conference, a mere five months away. There was general agreement on the list that, while this schedule was aggressive, it was achievable provided that the requirements for the first version were not too ambitious and that VRML could be adapted from an existing solution. The list quickly agreed upon a set of requirements for the first version, and began a search for technologies which could be adapted to fit the needs of VRML.

The search for existing technologies turned up a several worthwhile candidates. After much deliberation the list came to a consensus: the Open Inventor ASCII File Format from Silicon Graphics, Inc. The Inventor File Format supports complete descriptions of 3D worlds with polygonally rendered objects, lighting, materials, ambient properties and realism effects. A subset of the Inventor File Format, with extensions to support networking, forms the basis of VRML. Gavin Bell of Silicon Graphics has adapted the Inventor File Format for VRML, with design input from the mailing list. SGI has publicly stated that the file format is available for use in the open market, and have contributed a file format parser into the public domain to bootstrap VRML viewer development.

This is a clarified version of the 1.0 specification. No features have been added or changed from the original 1.0 version of the spec. This is a 'bug-fix' release of the spec, correcting misspellings, vague wording and misleading examples, and adding wording to better define the semantics of VRML.

Version 1.0 Requirements

VRML 1.0 is designed to meet the following requirements:

- Platform independence
- Extensibility
- Ability to work well over low-bandwidth connections

As with HTML, the above are absolute requirements for a network language standard; they should need little explanation here.

Early on the designers decided that VRML would not be an extension to HTML. HTML is designed for text, not graphics. Also, VRML requires even more finely tuned network optimizations than HTML; it is expected that a typical VRML world will be composed of many more "inline" objects and served up by many more servers than a typical HTML document. Moreover, HTML is an accepted standard, with existing implementations that depend on it. To impede the HTML design process with VRML issues and constrain the VRML design process with HTML compatibility concerns would be to do both languages a disservice. As a network language, VRML will succeed or fail independent of HTML.

It was also decided that, except for the hyper-linking feature, the first version of VRML would not support interactive behaviors. This was a practical decision intended to streamline design and implementation. Design of a language for describing interactive behaviors is a big job, especially when the language needs to express behaviors of objects communicating on a network. Such languages do exist; if we had chosen one of them, we would have risked getting into a "language war." People don't get excited about the syntax of a language for describing polygonal objects; people get *very* excited about the syntax of real languages for writing programs. Religious wars can extend the design process by months or years. In addition, networked inter-object operation requires brokering services such as those provided by CORBA or OLE, services which don't exist yet within WWW; we would have had to invent them. Finally, by keeping behaviors out of Version 1, we have made it a much smaller task to implement a viewer. We acknowledge that support for arbitrary interactive behaviors is critical to the long-term success of VRML; they will be included in Version 2.

Language Specification

The language specification is divided into the following sections:

- Language Basics
- Coordinate System
- Fields
- Nodes
- Instancing
- Extensibility
- An Example

Language Basics

At the highest level of abstraction, VRML is just a way for objects to read and write themselves. Theoretically, the objects can contain anything -- 3D geometry, MIDI data, JPEG images, anything. VRML defines a set of objects useful for doing 3D graphics. These objects are called Nodes.

Nodes are arranged in hierarchical structures called **scene graphs**. Scene graphs are more than just a collection of nodes; the scene graph defines an ordering for the nodes. The scene graph has a notion of *state* -- nodes earlier in the world can affect nodes that appear later in the world. For example, a Rotation or Material node will affect the nodes after it in the world. A mechanism is defined to limit the effects of properties (separator nodes), allowing parts of the scene graph to be functionally isolated from other parts.

Applications that interpret VRML files need not maintain the scene graph structure internally; the scene graph is merely a convenient way of describing objects.

A node has the following characteristics:

- **What kind of object it is.** A node might be a cube, a sphere, a texture map, a transformation, etc.

- **The parameters that distinguish this node from other nodes of the same type.** For example, each Sphere node might have a different radius, and different texture maps nodes will certainly contain different images to use as the texture maps. These parameters are called Fields. A node can have 0 or more fields.

- **A name to identify this node.** Being able to name nodes and refer to them elsewhere is very powerful; it allows a world's author to give hints to applications using the world about what is in the world, and creates possibilities for very powerful scripting extensions. Nodes do not have to be named, but if they are named, they can have only one name. However, names do not have to be unique-- several different nodes may be given the same name.

- **Child nodes.** Object hierarchy is implemented by allowing some types of nodes to contain other nodes. Parent nodes traverse their children in order during rendering. Nodes that may have children are referred to as **group nodes**. Group nodes can have zero or more children.

The syntax chosen to represent these pieces of information is straightforward:

```
DEF objectname objecttype { fields  children }
```

Only the object type and curly braces are required; nodes may or may not have a name, fields, and children.

Node names must not begin with a digit, and must not contain spaces or control characters, single or double quote characters, backslashes, curly braces, the plus character or the period character.

For example, this file contains a simple world defining a view of a red cone and a blue sphere, lit by a directional light:

```
#VRML V1.0 ascii
Separator {
    DirectionalLight {
        direction 0 0 -1  # Light shining from viewer into world
    }
    PerspectiveCamera {
        position    -8.6 2.1 5.6
        orientation -0.1352 -0.9831 -0.1233  1.1417
        focalDistance       10.84
    }
    Separator {   # The red sphere
        Material {
            diffuseColor 1 0 0   # Red
        }
        Translation { translation 3 0 1 }
        Sphere { radius 2.3 }
    }
    Separator {  # The blue cube
        Material {
            diffuseColor 0 0 1  # Blue
        }
        Transform {
            translation -2.4 .2 1
            rotation 0 1 1  .9
        }
        Cube {}
    }
}
```

General Syntax

For easy identification of VRML files, every VRML file must begin with the characters:

```
#VRML V1.0 ascii
```

Any characters after these on the same line are ignored. The line is terminated by either the ASCII newline or carriage-return characters.

The ’#’ character begins a comment; all characters until the next newline or carriage return are ignored. The only exception to this is within double-quoted SFString and MFString fields, where the ’#’ character will be part of the string.

Note: Comments and whitespace may not be preserved; in particular, a VRML document server may strip comments and extraneous whitespace from a VRML file before transmitting it. Info nodes should be used for persistent information like copyrights or author information. Info nodes could also be used for object descriptions.

Blanks, tabs, newlines and carriage returns are whitespace characters wherever they appear outside of string fields. One or more whitespace characters separates the syntactical entities in VRML files, where necessary.

After the required header, a VRML file contains exactly one VRML node. That node may of course be a group node, containing any number of other nodes.

VRML is case-sensitive; 'Sphere' is different from 'sphere'.

Node names must not begin with a digit, and must not contain spaces or control characters, single or double quote characters, backslashes, curly braces, the sharp (#) character, the plus (+) character or the period character.

Field names start with lower case letters, Node types start with upper case. The remainder of the characters may be any printable ascii (21H-7EH) except curly braces {}, square brackets [], single ' or double " quotes, sharp #, backslash \\ plus +, period . or ampersand &.

Coordinate System

VRML uses a Cartesian, right-handed, 3-dimensional coordinate system. By default, objects are projected onto a 2-dimensional device by projecting them in the direction of the positive Z axis, with the positive X axis to the right and the positive Y axis up. A camera or modeling transformation may be used to alter this default projection.

The standard unit for lengths and distances specified is meters. The standard unit for angles is radians.

VRML worlds may contain an arbitrary number of local (or "object-space") coordinate systems, defined by modeling transformations using Translate, Rotate, Scale, Transform, and MatrixTransform nodes. Given a vertex **V** and a series of transformations such as:

```
Translation { translation T }
Rotation { rotation R }
Scale { scaleFactor S }
Coordinate3 { point V } PointSet { numPoints 1 }
```

the vertex is transformed into world-space to get **v'** by applying the transformations in the following order:

V' = T·R·S·V (if you think of vertices as column vectors) OR
V' = V·S·R·T (if you think of vertices as row vectors)

Conceptually, VRML also has a "world" coordinate system as well as a viewing or "Camera" coordinate system. The various local coordinate transformations map objects into the world coordinate system. This is where the scene is assembled. The scene is then viewed through a camera, introducing another conceptual coordinate system. Nothing in VRML is specified using these coordinates. They are rarely found in optimized implementations where all of the steps are concatenated. However, having a clear model of the object, world and camera spaces will help authors.

Fields

There are two general classes of fields; fields that contain a single value (where a value may be a single number, a vector, or even an image), and fields that contain multiple values. Single-valued fields all have names that begin with "SF", multiple-valued fields have names that begin with "MF". Each field type defines the format for the values it writes.

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

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

SFBitMask

A single-value field that contains a mask of bit flags. Nodes that use this field class define mnemonic names for the bit flags. SFBitMasks are written to file as one or more mnemonic enumerated type names, in this format:

```
( flag1 | flag2 | ... )
```

If only one flag is used in a mask, the parentheses are optional. These names differ among uses of this field in various node classes.

No more than 32 separate flags may be defined for an SFBitMask.

SFBool

A field containing a single boolean (true or false) value. SFBools may be written as 0 (representing FALSE), 1, TRUE, or FALSE.

SFColor/MFColor

Fields containing one (SFColor) or zero or more (MFColor) RGB colors. Each color is written to file as an RGB triple of floating point numbers in ANSI C floating point format, in the range 0.0 to 1.0. For example:

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

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

SFEnum

A single-value field that contains an enumerated type value. Nodes that use this field class define mnemonic names for the values. SFEnums are written to file as a mnemonic enumerated type name.

The name differs among uses of this field in various node classes.

SFFloat/MFFloat

Fields that contain one (SFFloat) or zero or more (MFFloat) single-precision floating point number. SFFloats are written to file in ANSI C floating point format. For example:

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

is an MFFloat field containing three values.

SFImage

A field that contain an uncompressed 2-dimensional color or grey-scale image.

SFImages are written to file as three integers representing the width, height and number of components in the image, followed by width*height hexadecimal values representing the pixels in the image, separated by whitespace. A one-component image will have one-byte hexadecimal values representing the intensity of the image. For example, 0xFF is full intensity, 0x00 is no intensity. A two-component image puts the intensity in the first (high) byte and the transparency in the second (low) byte. Pixels in a three-component image have the red component in the first (high) byte, followed by the green and blue components (so 0xFF0000 is red). Four-component images put the transparency byte after red/green/blue (so 0x0000FF80 is semi-transparent blue). A value of 1.0 is completely transparent, 0.0 is completely opaque. Note: each pixel is actually read as a single unsigned number, so a 3-component pixel with value "0x0000FF" can also be written as "0xFF" or "255" (decimal). Pixels are specified from left to right, bottom to top. The first hexadecimal value is the lower left pixel of the image, and the last value is the upper right pixel.

For example,

1 2 1 0xFF 0x00

is a 1 pixel wide by 2 pixel high grey-scale image, with the bottom pixel white and the top pixel black. And:

2 4 3 0xFF0000 0xFF00 0 0 0 0 0xFFFFFF 0xFFFF00

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

SFLong/MFLong

Fields containing one (SFLong) or zero or more (MFLong) 32-bit integers. SFLongs are written to file as an integer in decimal, hexadecimal (beginning with '0x') or octal (beginning with '0') format. For example:

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

is an MFLong field containing three values.

SFMatrix

A field containing a transformation matrix. SFMatrices are written to file in row-major order as 16 floating point numbers separated by whitespace. For example, a matrix expressing a translation of 7.3 units along the X axis is written as:

```
1 0 0 0  0 1 0 0  0 0 1 0  7.3 0 0 1
```

SFRotation

A field containing an arbitrary rotation. SFRotations are written to file as four floating point values separated by whitespace. The 4 values represent an axis of rotation followed by the amount of right-handed rotation about that axis, in radians. For example, a 180 degree rotation about the Y axis is:

```
0 1 0  3.14159265
```

SFString/MFString

Fields containing one (SFString) or zero or more (MFString) ASCII string (sequence of characters). Strings are written to file as a sequence of ASCII characters in double quotes (optional if the string doesn't contain any whitespace). Any characters (including newlines and '#') may appear within the quotes. To include a double quote character within the string, precede it with a backslash. For example:

```
Testing
"One, Two, Three"
"He said, \"Immel did it!\""
```

are all valid strings.

SFVec2f/MFVec2f

Field containing a two-dimensional vector. SFVec2fs are written to file as a pair of floating point values separated by whitespace.

SFVec3f/MFVec3f

Field containing a three-dimensional vector. SFVec3fs are written to file as three floating point values separated by whitespace.

Nodes

VRML defines several different classes of nodes. Most of the nodes can be classified into one of three categories; shape, property or group. Shape nodes define the geometry in the world. Conceptually, they are the only nodes that draw anything. Property nodes affect the way shapes are drawn. And grouping nodes gather other nodes together, allowing collections of nodes to be treated as a single object. Some group nodes also control whether or not their children are drawn.

Nodes may contain zero or more fields. Each node type defines the type, name, and default value for each of its fields. The default value for the field is used if a value for the field is not specified in the VRML file. The order in which the fields of a node are read is not important; for example, "Cube { width 2 height 4 depth 6 }" and "Cube { height 4 depth 6 width 2 }" are equivalent.

Here are the 36 nodes grouped by type. The first group is the shape nodes. These specify geometry:

- AsciiText, Cone, Cube, Cylinder, IndexedFaceSet, IndexedLineSet, PointSet, Sphere

The second group are the properties. These can be further grouped into properties of the geometry and its appearance, and matrix or transform properties:

- Coordinate3, FontStyle, Material, MaterialBinding, Normal, NormalBinding, Texture2, Texture2Transform, TextureCoordinate2, ShapeHints
- MatrixTransform, Rotation, Scale, Transform, Translation

These are the group nodes:

- Separator, Switch, WWWAnchor, LOD

Finally, the following nodes do not fit neatly into any category.

- OrthographicCamera, PerspectiveCamera
- DirectionalLight, PointLight, SpotLight
- WWWInline, Info

AsciiText

This node represents strings of text characters from the ASCII coded character set. The first string is rendered with its baseline at (0,0,0). All subsequent strings advance y by -(*size* * *spacing*). See **FontStyle** for a description of the *size* field. The *justification* field determines the placement of the strings in the x dimension. LEFT (the default) places the left edge of each string at x=0. CENTER places the center of each string at x=0. RIGHT places the right edge of each string at x=0. Text is rendered from left to right, top to bottom in the font set by FontStyle.

The width field specifies the maximum rendered widths (in object space) for the strings. If the text is too long, it must be scaled to fit within this width. The default is to use the natural width of each string. If there are not enough values in the width field, the last value will be used for the remaining strings. Setting any value to 0 indicates the natural width should be used for that string.

The text is transformed by the current cumulative transformation and is drawn with the current material and texture.

Textures are applied to 3D text as follows. The texture origin is at the origin of the first string, as determined by the justification. The texture is scaled equally in both S and T dimensions, with the font height representing 1 unit. S increases to the right. The T origin can occur anywhere along each character, depending on how that character’s outline is defined.

```
JUSTIFICATION
     LEFT     Align left edge of text to origin
     CENTER   Align center of text to origin
     RIGHT    Align right edge of text to origin
FILE FORMAT/DEFAULTS
     AsciiText {
          string         ""    # MFString
          spacing        1     # SFFloat
```

```
        justification  LEFT   # SFEnum
        width          0      # MFFloat
    }
```

Cone

This node represents a simple cone whose central axis is aligned with the y-axis. By default, the cone is centered at (0,0,0) and has a size of -1 to +1 in all three directions. The cone has a radius of 1 at the bottom and a height of 2, with its apex at 1 and its bottom at -1. The cone has two parts: the sides and the bottom.

The cone is transformed by the current cumulative transformation and is drawn with the current texture and material.

If the current material binding is PER_PART or PER_PART_INDEXED, the first current material is used for the sides of the cone, and the second is used for the bottom. Otherwise, the first material is used for the entire cone.

When a texture is applied to a cone, it is applied differently to the sides and bottom. On the sides, the texture wraps counterclockwise (from above) starting at the back of the cone. The texture has a vertical seam at the back, intersecting the yz-plane. For the bottom, a circle is cut out of the texture square and applied to the cone's base circle. The texture appears right side up when the top of the cone is rotated towards the -Z axis.

```
PARTS
     SIDES       The conical part
     BOTTOM      The bottom circular face
     ALL         All parts
FILE FORMAT/DEFAULTS
     Cone {
          parts          ALL      # SFBitMask
          bottomRadius   1        # SFFloat
          height         2        # SFFloat
     }
```

Coordinate3

This node defines a set of 3D coordinates to be used by a subsequent IndexedFaceSet, IndexedLineSet, or PointSet node. This node does not produce a visible result during rendering; it simply replaces the current coordinates in the rendering state for subsequent nodes to use.

```
FILE FORMAT/DEFAULTS
     Coordinate3 {
          point  0 0 0  # MFVec3f
     }
```

Cube

This node represents a cuboid aligned with the coordinate axes. By default, the cube is centered at (0,0,0) and measures 2 units in each dimension, from -1 to +1. The cube is transformed by the current cumulative transformation and is drawn with the current material and texture. A cube's width is its extent along its object-space X axis, its height is its extent along the object-space Y axis, and its depth is its extent along its object-space Z axis.

If the current material binding is PER_PART, PER_PART_INDEXED, PER_FACE, or

PER_FACE_INDEXED, materials will be bound to the faces of the cube in this order: front (+Z), back (-Z), left (-X), right (+X), top (+Y), and bottom (-Y).

Textures are applied individually to each face of the cube; the entire texture goes on each face. On the front, back, right, and left sides of the cube, the texture is applied right side up. On the top, the texture appears right side up when the top of the cube is tilted toward the camera. On the bottom, the texture appears right side up when the top of the cube is tilted towards the -Z axis.

```
FILE FORMAT/DEFAULTS
     Cube {
          width   2      # SFFloat
          height  2      # SFFloat
          depth   2      # SFFloat
     }
```

Cylinder

This node represents a simple capped cylinder centered around the y-axis. By default, the cylinder is centered at (0,0,0) and has a default size of -1 to +1 in all three dimensions. The cylinder has three parts: the sides, the top (y = +1) and the bottom (y = -1). You can use the radius and height fields to create a cylinder with a different size.

The cylinder is transformed by the current cumulative transformation and is drawn with the current material and texture.

If the current material binding is PER_PART or PER_PART_INDEXED, the first current material is used for the sides of the cylinder, the second is used for the top, and the third is used for the bottom. Otherwise, the first material is used for the entire cylinder.

When a texture is applied to a cylinder, it is applied differently to the sides, top, and bottom. On the sides, the texture wraps counterclockwise (from above) starting at the back of the cylinder. The texture has a vertical seam at the back, intersecting the yz-plane. For the top and bottom, a circle is cut out of the texture square and applied to the top or bottom circle. The top texture appears right side up when the top of the cylinder is tilted toward the +Z axis, and the bottom texture appears right side up when the top of the cylinder is tilted toward the -Z axis.

```
PARTS
     SIDES   The cylindrical part
     TOP     The top circular face
     BOTTOM  The bottom circular face
     ALL     All parts
FILE FORMAT/DEFAULTS
     Cylinder {
          parts   ALL    # SFBitMask
          radius  1      # SFFloat
          height  2      # SFFloat
     }
```

DirectionalLight

This node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector.

A light node defines an illumination source that may affect subsequent shapes in the scene graph, depending on the current lighting style. Light sources are affected by the current transformation. A light node under a separator does not affect any objects outside that separator.

Light intensity must be in the range 0.0 to 1.0, inclusive.

```
FILE FORMAT/DEFAULTS
     DirectionalLight {
          on         TRUE       # SFBool
          intensity  1          # SFFloat
          color      1 1 1      # SFColor
          direction  0 0 -1     # SFVec3f
     }
```

FontStyle

This node defines the current font style used for all subsequent AsciiText. Font attributes only are defined. It is up to the browser to assign specific fonts to the various attribute combinations. The size field specifies the height (in object space units) of glyphs rendered and determines the vertical spacing of adjacent lines of text.

FAMILY

```
     SERIF       Serif style (such as Times-Romana)
     SANS        Sans Serif Style (such as Helvetica)
     TYPEWRITER  Fixed pitch style (such as Courier)
STYLE
     NONE        No modifications to family
     BOLD        Embolden family
     ITALIC      Italicize or Slant family
FILE FORMAT/DEFAULTS
     FontStyle {
          size     10      # SFFloat
          family   SERIF   # SFEnum
          style    NONE    # SFBitMask
     }
```

IndexedFaceSet

This node represents a 3D shape formed by constructing faces (polygons) from vertices located at the current coordinates. IndexedFaceSet uses the indices in its coordIndex to define polygonal faces. An index of -1 separates faces (so a -1 at the end of the list is optional).

The vertices of the faces are transformed by the current transformation matrix.

Treatment of the current material and normal binding is as follows: The PER_PART and PER_FACE bindings specify a material or normal for each face. PER_VERTEX specifies a material or normal for each vertex. The corresponding _INDEXED bindings are the same, but use the materialIndex or normalIndex indices. The DEFAULT material binding is equal to OVERALL. The DEFAULT normal binding is equal to PER_VERTEX_INDEXED; if insufficient normals exist in the state, vertex normals will be generated automatically. When materialIndex or normalIndex are specified on a per vertex basis, an index of -1 is used in the same way as coordIndex.

Explicit texture coordinates (as defined by TextureCoordinate2) may be bound to vertices of a polygon in an indexed shape by using the indices in the textureCoordIndex field. Therefore, the length of the coordIndex field and the length of the textureCoordIndex field should be equal, if textures are defined. This allows a vertex to have different texture indices on different faces (television on six sides of a cube).

As with all vertex-based shapes, if there is a current texture but no texture coordinates are specified, a default texture coordinate mapping is calculated using the bounding box of the shape. The longest dimension of the bounding box defines the S coordinates, and the next longest defines the T coordinates. The value of the S coordinate ranges from 0 to 1, from one end of the bounding box to the other. The T coordinate ranges between 0 and the ratio of the second greatest dimension of the bounding box to the greatest dimension. If any dimensions are equal, then the ordering of X>Y>Z is chosen. An index of -1 is used to separate textureCoordIndex faces, in the same way as coordIndex.

Be sure that the indices contained in the coordIndex, materialIndex, normalIndex, and textureCoordIndex fields are valid with respect to the current state, or errors will occur.

```
FILE FORMAT/DEFAULTS
     IndexedFaceSet {
          coordIndex         0  # MFLong
          materialIndex      -1 # MFLong
          normalIndex        -1 # MFLong
          textureCoordIndex  -1 # MFLong
     }
```

IndexedLineSet

This node represents a 3D shape formed by constructing polylines from vertices located at the current coordinates. IndexedLineSet uses the indices in its coordIndex field to specify the polylines. An index of -1 separates one polyline from the next (thus, a final -1 is optional). the current polyline has ended and the next one begins.

The coordinates of the line set are transformed by the current cumulative transformation.

Treatment of the current material and normal binding is as follows: The PER_PART binding specifies a material or normal for each segment of the line. The PER_FACE binding specifies a material or normal for each polyline. PER_VERTEX specifies a material or normal for each vertex. The corresponding _INDEXED bindings are the same, but use the materialIndex or normalIndex indices. The DEFAULT material binding is equal to OVERALL. The DEFAULT normal binding is equal to PER_VERTEX_INDEXED; if insufficient normals exist in the state, the lines will be drawn unlit. The same rules for texture coordinate generation as IndexedFaceSet are used.

```
FILE FORMAT/DEFAULTS
     IndexedLineSet {
          coordIndex         0  # MFLong
          materialIndex      -1 # MFLong
          normalIndex        -1 # MFLong
          textureCoordIndex  -1 # MFLong
     }
```

Info

This class defines an information node in the scene graph. This node has no effect during traversal. It is used to store information in the scene graph, typically for application-specific purposes, copyright messages, or other strings.

New uses of named info nodes for conveying syntactically meaningfull information are deprecated.

By deprecated, we mean the feature is discouraged and alternatives should be used if they exist. It is recommended that VRML's extensibility features be used for extensions, not the Info node.

```
Info {
     string  "<Undefined info>"      # SFString
}
```

LOD

This group node is used to allow applications to switch between various representations of objects automatically. The children of this node typically represent the same object or objects at varying levels of detail, from highest detail to lowest. LOD must be used as a separator, since using LOD to side-effect state (such as color or translation) causes implementation difficulties for browser writers. To ensure that LOD is used properly, we recommend that each child of the LOD be contained in its own Separator node.

The distance from the viewpoint (transformed into the local coordinate space of the LOD node) to the specified center point of the LOD is calculated, in object coordinates. If the distance is less than the first value in the ranges array, then the first child of the LOD is drawn. If between the first and second values in the ranges array, the second child is drawn, etc. If there are N values in the ranges array, the LOD should have N+1 children. Specifying too few children will result in the last child being used repeatedly for the lowest levels of detail; if too many children are specified, the extra children will be ignored. Each value in the ranges array should be greater than the previous value, otherwise results are undefined.

Authors should set LOD ranges so that the transitions from one level of detail to the next are barely noticeable. Applications may treat the range field as a hint, and might adjust which level of detail is displayed to maintain interactive frame rates, to display an already-fetched level of detail while a higher level of detail (contained in a WWWInline node) is fetched, or might disregard the author-specified ranges for any other implementation-dependent reason. Authors should not use LOD nodes to emulate simple behaviors, because the results will be undefined. For example, using an LOD node to make a door appear to open when the user approaches probably will not work in all browsers.

It is expected that in a future version of VRML the LOD node will be defined to behave as a Separator node, not allowing its children to affect anything after it in the scene graph. To ensure future compatibility, it is recommended that all children of all LOD nodes be Separator nodes.

```
FILE FORMAT/DEFAULTS
     LOD {
          range [ ]    # MFFloat
          center 0 0 0  # SFVec3f
     }
```

Material

This node defines the current surface material properties for all subsequent shapes. Material sets several components of the current material during traversal. Different shapes interpret materials with multiple values differently. To bind materials to shapes, use a MaterialBinding node.

The fields in the Material node determine the way light reflects off of an object to create color:

- The ambientColor reflects ambient light evenly from all parts of an object regardless of viewing and lighting angles.

- The diffuseColor reflects all VRML light sources depending on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light reflects.
- The specularColor and shininess determine the specular highlights, e.g., the shiny spots on an apple. When the angle from the light to the surface is close to the angle from the surface to the viewer, the specularColor is added to the diffuse and ambient color calculations. Lower shininess values produce soft glows, while higher values result in sharper, smaller highlights.
- Emissive color models "glowing" objects. This can be useful for displaying radiosity-based models (where the light energy of the room is computed explicitly), or for displaying scientific data.
- Transparency is how "clear" the object is, with 1.0 being completely transparent, and 0.0 completely opaque.

The lighting parameters defined by the Material node are the same parameters defined by the OpenGL lighting model. For a rigorous mathematical description of how these parameters should be used to determine how surfaces are lit, see the description of lighting operations in the OpenGL Specification. Several of the OpenGL parameters (such as light attenuation factors) are also left unspecified in VRML. Also note that OpenGL specifies the specular exponent as a non-normalized 0-128 value, which is specified as a normalized 0-1 value in VRML (simply multiplying the VRML "shininess" value by 128 to translate to the OpenGL specular exponent parameter).

We assume that there is an implicit white ambient light of intensity 1.0 in any VRML scene.

Issues for Low-End Rendering Systems. For rendering systems that do not support the full OpenGL lighting model, the following simpler lighting model is recommended:

A transparency value of 0 is completely opaque, a value of 1 is completely transparent. Applications need not support partial transparency, but should support at least fully transparent and fully opaque surfaces, treating transparency values >= 0.5 as fully transparent.

Specifying only emissiveColors and no diffuse, specular, emissive, or ambient colors is the way to specify pre-computed lighting. It is expected that browsers will be able to recognize this as a special case and optimize their computations. For example:

```
Material {
    ambientColor [] diffuseColor [] specularColor []
    emissiveColor [ 0.1 0.1 0.2, 0.5 0.8 0.8 ]
}
```

Notice that an empty list, specified by [], is different than not specifying the field. An empty list explicitly means no value; if nothing is specified, the default value will be used.

Many low-end PC rendering systems are not able to support the full range of the VRML material specification. For example, many systems do not render individual red, green and blue reflected values as specified in the specularColor field. The following table describes which Material fields are typically supported in popular low-end systems and suggests actions for browser implementors to take when a field is not supported.

Field	Supported?	Suggested Action
ambientColor	No	Ignore
diffuseColor	Yes	Use as base color

```
specularColor   No              Ignore
emissiveColor   No              Ignore, unless all others are empty
shininess       Yes             Use
transparency    No              Ignore
```

Rendering systems which do not support specular color may nevertheless support a specular intensity. The following recommendation gives reasonable results for both plastic highlights where the specular color is white and also for metallic highlights where the specular color is the same as or a ratio of the diffuse color. To get the specular coefficient, take the highest specular component and, if the same component of the diffuse color is nonzero, divide by the diffuse. If the diffuse component is zero, use the specular component. In other symbols:

```
  max_channel = red;
  if specular[green] > specular[max_channel] then max_channel = green;
  if specular[blue] > specular[max_channel] then max_channel = blue;
  if diffuse[max_channel] > 0 then
    Ks = MAX(spec[mc]/diff[mc],1) else Ks = spec[mc]
```

If a system supports ambient intensity but not color, the intensity should be derived by taking the dot product of the specified RGB ambient value with the vector [.299 .587 .114]. This adjusts the value to compensate for the variable sensitivity of the eye to colors. For more on this, see [FVDH], 13.33, page 589. If a rendering system does not support per-object ambient values, it should set the ambient value for the entire scene at the average ambient value of all objects contained in the top level file.

It is also expected that simpler rendering systems will be unable to support both lit and unlit objects in the same world.

Many VRML implementations will support only either multiple diffuse colors with a single value for all other fields, or multiple emissive colors with one transparency value and NO (empty, '[]') values for all other fields. More complicated uses of the Material node should be avoided.

```
FILE FORMAT/DEFAULTS
     Material {
          ambientColor   0.2 0.2 0.2    # MFColor
          diffuseColor   0.8 0.8 0.8    # MFColor
          specularColor  0 0 0          # MFColor
          emissiveColor  0 0 0          # MFColor
          shininess      0.2            # MFFloat
          transparency   0              # MFFloat
     }
```

MaterialBinding

Material nodes may contain more than one material. This node specifies how the current materials are bound to shapes that follow in the scene graph. Each shape node may interpret bindings differently. For example, a Sphere node is always drawn using the first material in the material node, no matter what the current MaterialBinding, while a Cube node may use six different materials to draw each of its six faces, depending on the MaterialBinding.

The bindings for faces and vertices are meaningful only for shapes that are made from faces and vertices. Similarly, the indexed bindings are only used by the shapes that allow indexing.

When multiple material values are needed by a shape, the previous Material node should have at least as many materials as are needed, otherwise results are undefined.

Note that some rendering systems do not support per-vertex material changes. Browsers that do not support per-vertex colors should average the colors specified when a PER_VERTEX binding is used.

```
BINDINGS
     DEFAULT            Use default binding
     OVERALL            Whole object has same material
     PER_PART           One material for each part of object
     PER_PART_INDEXED   One material for each part, indexed
     PER_FACE           One material for each face of object
     PER_FACE_INDEXED   One material for each face, indexed
     PER_VERTEX         One material for each vertex of object
     PER_VERTEX_INDEXED One material for each vertex, indexed
FILE FORMAT/DEFAULTS
     MaterialBinding {
          value  OVERALL        # SFEnum
     }
```

MatrixTransform

This node defines a geometric 3D transformation with a 4 by 4 matrix. Only matrices that are the result of rotations, translations, and non-zero (but possibly non-uniform) scales must be supported. Non-invertible matrices should be avoided.

Matrices are specified in row-major order, so, for example, a MatrixTransform representing a translation of 6.2 units along the local Z axis would be specified as:

```
MatrixTransform { matrix
    1 0 0 0
    0 1 0 0
    0 0 1 0
    0 0 6.2 1
}

FILE FORMAT/DEFAULTS
     MatrixTransform {
          matrix  1 0 0 0       # SFMatrix
                  0 1 0 0
                  0 0 1 0
                  0 0 0 1
     }
```

Normal

This node defines a set of 3D surface normal vectors to be used by vertex-based shape nodes (IndexedFaceSet, IndexedLineSet, PointSet) that follow it in the scene graph. This node does not produce a visible result during rendering; it simply replaces the current normals in the rendering state for subsequent nodes to use. This node contains one multiple-valued field that contains the normal vectors.

To save network bandwidth, it is expected that implementations will be able to automatically generate appropriate normals if none are given. However, the results will vary from implementation to implementation.

```
FILE FORMAT/DEFAULTS
```

```
Normal {
     vector  [ ] # MFVec3f
}
```

NormalBinding

This node specifies how the current normals are bound to shapes that follow in the scene graph. Each shape node may interpret bindings differently.

The bindings for faces and vertices are meaningful only for shapes that are made from faces and vertices. Similarly, the indexed bindings are only used by the shapes that allow indexing. For bindings that require multiple normals, be sure to have at least as many normals defined as are necessary; otherwise, errors will occur.

Browsers that do not support per-vertex normals should average the normals specified when a PER_VERTEX binding is used.

```
BINDINGS
     DEFAULT             Use default binding
     OVERALL             Whole object has same normal
     PER_PART            One normal for each part of object
     PER_PART_INDEXED    One normal for each part, indexed
     PER_FACE            One normal for each face of object
     PER_FACE_INDEXED    One normal for each face, indexed
     PER_VERTEX          One normal for each vertex of object
     PER_VERTEX_INDEXED One normal for each vertex, indexed
FILE FORMAT/DEFAULTS
     NormalBinding {
          value  DEFAULT        # SFEnum
     }
```

OrthographicCamera

An orthographic camera defines a parallel projection from a viewpoint. This camera does not diminish objects with distance, as a PerspectiveCamera does. The viewing volume for an orthographic camera is a rectangular parallelepiped (a box).

By default, the camera is located at (0,0,1) and looks along the negative z-axis; the position and orientation fields can be used to change these values (orientation is a rotation from this angle). The height field defines the total height of the viewing volume.

A camera can be placed in a VRML world to specify the initial location of the viewer when that world is entered. VRML browsers will typically modify the camera to allow a user to move through the virtual world.

The results of traversing multiple cameras are undefined; to ensure consistent results, place multiple cameras underneath one or more Switch nodes, and set the Switch’s whichChild fields so that only one is traversed. By convention, these non-traversed cameras may be used to define alternate entry points into the world; these entry points may be named by simply giving the cameras a name (using DEF); see the specification of WWWAnchor for a conventional way of specifying an entry point in a URL.

Cameras are affected by the current transformation, so you can position a camera by placing a transformation node before it in the scene graph . The default position and orientation of a camera is at (0,0,1) looking along the negative z-axis, with the positive y-axis up.

The position and orientation fields of a camera are sufficient to place a camera anywhere in space with any orientation. The orientation field can be used to rotate the default view direction (looking down -z-, with +y up) so that it is looking in any direction, with any direction 'up'.

The focalDistance field is not to be confused with focal length used to describe a lens in optics. Instead, this is the distance from the Camera to a point in space along the vector defined by the view direction given by the Camera's angles and position. This point in space is where the "viewer" is focused attention. This value is a hint only and may be used by a browser to define the speed of travel for flying or walking.

For example, a focalDistance of 5 means the object of primary concern is 5 meters from the camera and the browser should adjust flying speed to reach that point in a reasonable amount of time. If the distance was 50 meters then perhaps the browser can use this as a hint to travel 10 times faster.

The heightAngle of a PerspectiveCamera can be used to simulate a particular lens and camera. To calculate a heightAngle use the following formula: `heightAngle = 2.0 * arctan((height/2) / focalLength )`. Where height is the height of the scene at the desired focalLength (distance from position to center of attention).

```
FILE FORMAT/DEFAULTS
     OrthographicCamera {
          position         0 0 1        # SFVec3f
          orientation      0 0 1  0     # SFRotation
          focalDistance    5            # SFFloat
          height           2            # SFFloat
     }
```

PerspectiveCamera

A perspective camera defines a perspective projection from a viewpoint. The viewing volume for a perspective camera is a truncated right pyramid.

By default, the camera is located at (0,0,1) and looks along the negative z-axis; the position and orientation fields can be used to change these values. The heightAngle field defines the total vertical angle of the viewing volume.

See more on cameras in the OrthographicCamera description.

```
FILE FORMAT/DEFAULTS
     PerspectiveCamera {
          position         0 0 1        # SFVec3f
          orientation      0 0 1  0     # SFRotation
          focalDistance    5            # SFFloat
          heightAngle      0.785398     # SFFloat
     }
```

PointLight

This node defines a point light source at a fixed 3D location. A point source illuminates equally in all directions; that is, it is omni- directional.

A light node defines an illumination source that may affect subsequent shapes in the scene graph, depending on the current lighting style. Light sources are affected by the current transformation. A light node under a separator should not affect any objects outside that separator (although some rendering systems do not currently support this).

Light intensity must be in the range 0.0 to 1.0, inclusive.

```
FILE FORMAT/DEFAULTS
     PointLight {
          on          TRUE        # SFBool
          intensity   1           # SFFloat
          color       1 1 1       # SFColor
          location    0 0 1       # SFVec3f
     }
```

PointSet

This node represents a set of points located at the current coordinates. PointSet uses the current coordinates in order, starting at the index specified by the startIndex field. The number of points in the set is specified by the numPoints field. A value of -1 for this field indicates that all remaining values in the current coordinates are to be used as points.

The coordinates of the point set are transformed by the current cumulative transformation. The points are drawn with the current material and texture.

Treatment of the current material and normal binding is as follows: PER_PART, PER_FACE, and PER_VERTEX bindings bind one material or normal to each point. The DEFAULT material binding is equal to OVERALL. The DEFAULT normal binding is equal to PER_VERTEX. The startIndex is also used for materials or normals when the binding indicates that they should be used per vertex.

```
FILE FORMAT/DEFAULTS
     PointSet {
          startIndex  0 # SFLong
          numPoints   -1        # SFLong
     }
```

Rotation

This node defines a 3D rotation about an arbitrary axis through the origin. The rotation is accumulated into the current transformation, which is applied to subsequent shapes.

```
FILE FORMAT/DEFAULTS
     Rotation {
          rotation  0 0 1  0    # SFRotation
     }
```

See rotation field description for more information.

Scale

This node defines a 3D scaling about the origin. If the components of the scaling vector are not all the same, this produces a non-uniform scale.

```
FILE FORMAT/DEFAULTS
     Scale {
          scaleFactor  1 1 1    # SFVec3f
     }
```

Separator

This group node performs a push (save) of the traversal state before traversing its children and a pop (restore) after traversing them. This isolates the separator’s children from the rest of the scene graph. A separator can include lights, cameras, coordinates, normals, bindings, and all other properties.

Separators can also perform render culling. Render culling skips over traversal of the separator’s children if they are not going to be rendered, based on the comparison of the separator’s bounding box with the current view volume. Culling is controlled by the renderCulling field. These are set to AUTO by default, allowing the implementation to decide whether or not to cull.

```
CULLING ENUMS
     ON    Always try to cull to the view volume
     OFF   Never try to cull to the view volume
     AUTO  Implementation-defined culling behavior
FILE FORMAT/DEFAULTS
     Separator {
          renderCulling       AUTO      # SFEnum
     }
```

ShapeHints

The ShapeHints node indicates that IndexedFaceSets are solid, contain ordered vertices, or contain convex faces.

These hints allow VRML implementations to optimize certain rendering features. Optimizations that may be performed include enabling back-face culling and disabling two-sided lighting. For example, if an object is solid and has ordered vertices, an implementation may turn on backface culling and turn off two-sided lighting. To ensure that an IndexedFaceSet can be viewed from either direction, set shapeType to be UNKNOWN_SHAPE_TYPE.

If you know that your shapes are closed and will alwsys be viewed from the outside, set vertexOrdering to be either CLOCKWISE or COUNTERCLOCKWISE (depending on how you built your object), and set shapeType to be SOLID. Placing this near the top of your VRML file will allow the scene to be rendered much faster:

```
     ShapeHints {
          vertexOrdering  CLOCKWISE   # (or COUNTERCLOCKWISE)
          shapeType       SOLID
     }
```

The ShapeHints node also affects how default normals are generated. When an IndexedFaceSet has to generate default normals, it uses the creaseAngle field to determine which edges should be smoothly shaded and which ones should have a sharp crease. The crease angle is the angle between surface normals on adjacent polygons. For example, a crease angle of .5 radians (the default value) means that an edge between two adjacent polygonal faces will be smooth shaded if the normals to the two faces form an angle that is less than .5 radians (about 30 degrees). Otherwise, it will be faceted.

```
VERTEX ORDERING ENUMS
     UNKNOWN_ORDERING    Ordering of vertices is unknown
     CLOCKWISE           Face vertices are ordered clockwise
                          (from the outside)
```

```
     COUNTERCLOCKWISE    Face vertices are ordered counterclockwise
                          (from the outside)
SHAPE TYPE ENUMS
     UNKNOWN_SHAPE_TYPE  Nothing is known about the shape
     SOLID               The shape encloses a volume
FACE TYPE ENUMS
     UNKNOWN_FACE_TYPE   Nothing is known about faces
     CONVEX              All faces are convex
FILE FORMAT/DEFAULTS
     ShapeHints {
          vertexOrdering  UNKNOWN_ORDERING      # SFEnum
          shapeType       UNKNOWN_SHAPE_TYPE    # SFEnum
          faceType        CONVEX                # SFEnum
          creaseAngle     0.5                   # SFFloat
     }
```

Sphere

This node represents a sphere. By default, the sphere is centered at the origin and has a radius of 1. The sphere is transformed by the current cumulative transformation and is drawn with the current material and texture.

A sphere does not have faces or parts. Therefore, the sphere ignores material and normal bindings, using the first material for the entire sphere and using its own normals. When a texture is applied to a sphere, the texture covers the entire surface, wrapping counterclockwise from the back of the sphere. The texture has a seam at the back on the yz-plane.

```
FILE FORMAT/DEFAULTS
     Sphere {
          radius  1     # SFFloat
     }
```

The radius should be positive. Special uses of negative radius spheres should be replaced by extension nodes.

SpotLight

This node defines a spotlight light source. A spotlight is placed at a fixed location in 3-space and illuminates in a cone along a particular direction. The intensity of the illumination drops off exponentially as a ray of light diverges from this direction toward the edges of the cone. The rate of drop-off and the angle of the cone are controlled by the dropOffRate and cutOffAngle fields.

A light node defines an illumination source that may affect subsequent shapes in the scene graph, depending on the current lighting style. Light sources are affected by the current transformation. A light node under a separator should not affect any objects outside that separator (although some rendering systems do not currently support this).

Light intensity must be in the range 0.0 to 1.0, inclusive.

```
FILE FORMAT/DEFAULTS
     SpotLight {
          on           TRUE      # SFBool
          intensity    1         # SFFloat
          color        1 1 1     # SFVec3f
          location     0 0 1     # SFVec3f
          direction    0 0 -1    # SFVec3f
          dropOffRate  0         # SFFloat
```

```
          cutOffAngle  0.785398 # SFFloat
     }
```

Switch

This group node traverses one or none of its children. One can use this node to switch on and off the effects of some properties or to switch between different properties.

The whichChild field specifies the index of the child to traverse, where the first child has index 0.

A value of -1 (the default) means do not traverse any children.

```
FILE FORMAT/DEFAULTS
     Switch {
          whichChild  -1        # SFLong
     }
```

The behavior where a whichChild value of -3 traverses all children, making the switch behave exactly like a Group node, has been deprecated.

Texture2

This property node defines a texture map and parameters for that map. This map is used to apply texture to subsequent shapes as they are rendered.

The texture can be read from the URL specified by the filename field. To turn off texturing, set the filename field to an empty string (""). Implementations should support at least the JPEG image file format, with PNG strongly recommended. Due to legal issues, we do not require supporting the GIF format, though many existing scenes contain GIF files.

Renderers which support transparent texture maps should pay attention to any alpha channel information in the texture map. This allows for "cookie-cutter" effects (trees, people, etc).

Textures can also be specified inline by setting the image field to contain the texture data. Supplying both image and filename fields will result in undefined behavior.

Texture images may be one component (grey-scale), two component (grey-scale plus transparency), three component (full RGB color), or four-component (full RGB color plus transparency). An ideal VRML implementation will use the texture image to modify the diffuse color and transparency of an object's material (specified in a Material node), then performing any lighting calculations using the rest of the object's material properties with the modified diffuse color to produce the final image. The texture image modifies the diffuse color and transparency depending on how many components are in the image, as follows:

1. Diffuse color is multiplied by the grey-scale values in the texture image.
2. Diffuse color is multiplied by the grey-scale values in the texture image, material transparency is multiplied by transparency values in texture image.
3. RGB colors in the texture image replace the material's diffuse color.
4. RGB colors in the texture image replace the material's diffuse color, transparency values in the texture image replace the material's transparency.

Browsers may approximate this ideal behavior to increase performance. One common optimization is to calculate lighting only at each vertex and combining the texture image with the color computed from lighting (performing the texturing after lighting). Another common optimization is to perform no lighting calculations at all when texturing is enabled, displaying only the colors of the texture image.

```
WRAP ENUM
     REPEAT  Repeats texture outside 0-1 texture coordinate range
     CLAMP   Clamps texture coordinates to lie within 0-1 range
FILE FORMAT/DEFAULTS
     Texture2 {
          filename    ""        # SFString
          image       0 0 0     # SFImage
          wrapS       REPEAT    # SFEnum
          wrapT       REPEAT    # SFEnum
     }
```

Texture2Transform

This node defines a 2D transformation applied to texture coordinates. This affects the way textures are applied to the surfaces of subsequent shapes. The transformation consists of (in order) a non-uniform scale about an arbitrary center point, a rotation about that same point, and a translation. This allows a user to change the size and position of the textures on shapes.

```
FILE FORMAT/DEFAULTS
     Texture2Transform {
          translation  0 0      # SFVec2f
          rotation     0        # SFFloat
          scaleFactor  1 1      # SFVec2f
          center       0 0      # SFVec2f
     }
```

TextureCoordinate2

This node defines a set of 2D coordinates to be used to map textures to the vertices of subsequent PointSet, IndexedLineSet, or IndexedFaceSet objects. It replaces the current texture coordinates in the rendering state for the shapes to use.

Texture coordinates range from 0 to 1 across the texture. The horizontal coordinate, called S, is specified first, followed by the vertical coordinate, T.

```
FILE FORMAT/DEFAULTS
     TextureCoordinate2 {
          point  0 0    # MFVec2f
     }
```

Transform

This node defines a geometric 3D transformation consisting of (in order) a (possibly) non-uniform scale about an arbitrary point, a rotation about an arbitrary point and axis, and a translation. The transform node

```
Transform {
    translation T1
    rotation R1
    scaleFactor S
    scaleOrientation R2
    center T2
  }
```

is equivalent to the sequence:

```
Translation { translation T1 }
Translation { translation T2 }
Rotation { rotation R1 }
Rotation { rotation R2 }
Scale { scaleFactor S }
Rotation { rotation -R2 }
Translation { translation -T2 }
```

```
FILE FORMAT/DEFAULTS
     Transform {
          translation       0 0 0        # SFVec3f
          rotation          0 0 1  0     # SFRotation
          scaleFactor       1 1 1        # SFVec3f
          scaleOrientation  0 0 1  0     # SFRotation
          center            0 0 0        # SFVec3f
     }
```

Translation

This node defines a translation by a 3D vector.

```
FILE FORMAT/DEFAULTS
     Translation {
          translation  0 0 0    # SFVec3f
     }
```

WWWAnchor

The WWWAnchor group node loads a new world into a VRML browser when one of its children is chosen, assuming it is a .wrl file. Other types of URLs may be handled differently. Exactly how a user "chooses" a child of the WWWAnchor is up to the VRML browser; typically, clicking on one of its children with the mouse will result in the new world replacing the current world. A WWWAnchor with an empty ("") name does nothing when its children are chosen. If WWWAnchors are nested, the most deeply nested WWWAnchor is the one which is chosen.

The name is an arbitrary URL, as defined in RFC 1738, with relative URL semantics, as defined in RFC 1808. Browsers which require additional information to be associated with following a link are encouraged to create an extension node with additional fields.

WWWAnchor behaves like a Separator, pushing the traversal state before traversing its children and popping it afterwards.

The description field in the WWWAnchor allows for a friendly prompt to be displayed as an alternative to the URL in the name field. Ideally, browsers will allow the user to choose the description, the URL or both to be displayed for a candidate WWWAnchor.

The WWWAnchor’s map field is an enumerated value that can be either NONE (the default) or POINT. If it is POINT then the object-space coordinates of the point on the object the user chose will be added to the URL in the name field, with the syntax "?x,y,z".

A WWWAnchor may be used to take the viewer to a particular *viewpoint* in a virtual world by specifying a URL ending with "#cameraName", where "cameraName" is the name of a camera defined in the world. For example:

```
WWWAnchor {
    name "http://www.school.edu/vrml/someWorld.wrl#OverView"
    Cube { }
}
```

specifies an anchor that puts the viewer in the "someWorld" world looking from the camera named "OverView" when the Cube is chosen. If no world is specified, then the current world is implied; for example:

```
WWWAnchor {
    name "#Doorway"
    Sphere { }
}
```

will take the viewer to the viewpoint defined by the "Doorway" camera in the current world when the sphere is chosen.

```
MAP ENUM
     NONE  Do not add information to the URL
     POINT Add object-space coordinates to URL
FILE FORMAT/DEFAULTS
     WWWAnchor {
          name ""        # SFString
          description "" # SFString
          map NONE       # SFEnum
     }
```

WWWInline

The WWWInline node reads its children from anywhere in the World Wide Web. Exactly when its children are read is not defined; reading the children may be delayed until the WWWInline is actually displayed. A WWWInline with an empty name does nothing. The name is an arbitrary URL.

The effect of referring to a non-VRML URL in a WWWInline node is undefined.

If the WWWInline's bboxSize field specifies a non-empty bounding box (a bounding box is non-empty if at least one of its dimensions is greater than zero), then the WWWInline's object-space bounding box is specified by its bboxSize and bboxCenter fields. This allows an implementation to quickly determine whether or not the contents of the WWWInline might be visible. This is an optimization hint; if the true bounding box of the contents of the WWWInline is different from the specified bounding box results will be undefined. Browsers may draw the bounding box in wireframe while they are fetching the contents.

```
FILE FORMAT/DEFAULTS
     WWWInline {
          name ""               # SFString
          bboxSize 0 0 0        # SFVec3f
          bboxCenter 0 0 0      # SFVec3f
     }
```

Instancing

A node may be the child of more than one group. This is called "instancing" (using the same instance of a node multiple times, called "aliasing" or "multiple references" by other systems), and is accomplished by using the "USE" keyword.

The DEF keyword gives a node a name. DEF also creates a copy of the node. The USE keyword indicates that a named node should be used again. If several nodes were given the same name, then the last DEF encountered during parsing "wins" (the occurence of multiple nodes with the same name is strongly discouraged and may cause problems in the future). DEF/USE is limited to a single file; there is no mechanism for USE'ing nodes that are DEF'ed in other files. Refer to the " General Syntax" section of this specification for the legal syntax of node names.

A name goes into scope as soon as the DEF is encountered, and does not go out of scope until another DEF of the same name or end-of-file are encountered. Nodes cannot be shared between files (you cannot USE a node that was DEF'ed inside the file to which a WWWInline refers).

For example, rendering this world will result in three spheres being drawn. Both of the spheres are named 'Joe'; the second (smaller) sphere is drawn twice:

```
#VRML 1.0 ascii
Separator {
        DEF Joe Sphere { }
        Translation { translation 2 0 0 }
        Separator {
                DEF Joe Sphere { radius .2 }
        }
        Translation { translation 2 0 0 }
        USE Joe    # radius .2 sphere will be used here
}
```

Extensibility

Extensions to VRML are supported by supporting self-describing nodes. Nodes that are not part of standard VRML must write out a description of their fields first, so that all VRML implementations are able to parse and ignore the extensions.

This description is written just after the opening curly-brace for the node, and consists of the keyword 'fields' followed by a list of the types and names of fields used by that node, all enclosed in square brackets and separated by commas. For example, if Cube was not a standard VRML node, it would be written like this:

```
Cube {
  fields [ SFFloat width, SFFloat height, SFFloat depth ]
  width 10 height 4 depth 3
}
```

Specifying the fields for nodes that ARE part of standard VRML is not an error; VRML parsers must silently ignore the field[] specification. However, incorrectly specifying the fields of a built-in node is an error.

The fields specification must be written out with every non-standard node, whether or not that node type was previously encountered during parsing. For each instance of a non-standard node, only the fields written as part of that instance need to be described in the fields[] specification; that is, fields that aren't written because they contain their default value may be omitted from the fields[] specification. It is expected that future versions of VRML will relax this requirement, requiring only that the first non-standard node of a given type be given the fields[] specification.

Just like standard nodes, instances of non-standard nodes do not automatically share anything besides the default values of their fields, which are not specified in the VRML file but are considered part of the implementation of the non-standard nodes.

Is-a relationships

A new node type may also be a superset of an existing node that is part of the standard. In this case, if an implementation for the new node type cannot be found the new node type can be safely treated as as the existing node it is based on (with some loss of functionality, of course). To support this, new node types can define an MFString field called 'isA' containing the names of the types of which it is a superset. For example, a new type of Material called "ExtendedMaterial" that adds index of refraction as a material property can be written as:

```
ExtendedMaterial {
  fields [ MFString isA, MFFloat indexOfRefraction,
           MFColor diffuseColor, MFFloat transparency ]
  isA [ "Material" ]
  indexOfRefraction .34
  diffuseColor .8 .54 1
}
```

Multiple is-a relationships may be specified in order of preference; implementations are expected to use the first for which there is an implementation.

Note that is-a is NOT meant to be a full prototyping mechanism.

Naming conventions

Check the General Syntax section of this standard for the rules on valid characters in names.

To avoid namespace collisions with nodes defined by other people, the following conventions should be followed:

1. Anyone can pick names that include a suffix of an underscore followed by a domain name that you own with the periods changed into underscores. For example a company owning foo.com could create an extension node "Cube_foo_com"
2. If you are building a product for example an authoring tool, or a browser, or defining a lot of new nodes then you can apply for a short suffix. Email type_registry@vrml.org to register for the suffix. This will normally be accepted if it is the most significant part of a .com, .org or .net address, in the above example foo.com could register the extension "_foo" and create nodes of the form "Cube_foo".
3. Extensions supported by several companies should be registered and use the "_X" extension. These registered nodes are documented at http://vag.vrml.org/vrml_ext_nodes.html.

This process may change as more experience is gained.

An Example

This is a longer example of a VRML world. It contains a simple model of a track-light consisting of primitive shapes, plus three walls (built out of polygons) and a reference to a shape defined elsewhere, both of which are illuminated by a spotlight. The shape acts as a hyper-link to some HTML text.

```
#VRML V1.0 ascii
```

```
Separator {
    Separator {       # Simple track-light geometry:
        Translation { translation 0 4 0 }
        Separator {
            Material { emissiveColor 0.1 0.3 0.3 }
            Cube {
                width   0.1
                height  0.1
                depth   4
            }
        }
        Rotation { rotation 0 1 0  1.57079 }
        Separator {
            Material { emissiveColor 0.3 0.1 0.3 }
            Cylinder {
                radius  0.1
                height  .2
            }
        }
        Rotation { rotation -1 0 0  1.57079 }
        Separator {
            Material { emissiveColor 0.3 0.3 0.1 }
            Rotation { rotation 1 0 0  1.57079 }
            Translation { translation 0 -.2 0 }
            Cone {
                height  .4
                bottomRadius .2
            }
            Translation { translation 0 .4 0 }
            Cylinder {
                radius  0.02
                height  .4
            }
        }
    }
    SpotLight {      # Light from above
        location 0 4 0
        direction 0 -1 0
        intensity       0.9
        cutOffAngle     0.7
    }
    Separator {      # Wall geometry; just three flat polygons
        Coordinate3 {
            point [
                   -2 0 -2, -2 0 2, 2 0 2, 2 0 -2,
                   -2 4 -2, -2 4 2, 2 4 2, 2 4 -2]
        }
        IndexedFaceSet {
            coordIndex [ 0, 1, 2, 3, -1,
                         0, 4, 5, 1, -1,
                         0, 3, 7, 4, -1
                         ]
        }
    }
    WWWAnchor {   # A hyper-linked cow:
        name "http://www.foo.edu/CowProject/AboutCows.html"
        Separator {
            Translation { translation 0 1 0 }
            WWWInline {   # Reference another object
                name "http://www.foo.edu/3DObjects/cow.wrl"
            }
        }
    }
}
```

Browser Considerations

This section describes the file naming and MIME conventions to be used in building VRML browsers and configuring WWW browsers to work with them.

File Extensions

The file extension for VRML files is `.wrl` (for *world*).

MIME

The MIME type for VRML files is defined as follows:

```
x-world/x-vrml
```

The MIME major type for 3D world descriptions is `x-world`. The MIME minor type for VRML documents is `x-vrml`. Other 3D world descriptions, such as `oogl` for The Geometry Center's *Object-Oriented Geometry Language*, or `iv`, for SGI's *Open Inventor* ASCII format, can be supported by using different MIME minor types.

Deprecated Features

It is expected that these features will be removed in a future release of VRML, and their use is discouraged.

Group

This node defines the base class for all group nodes. Group is a node that contains an ordered list of child nodes. This node is simply a container for the child nodes and does not alter the traversal state in any way. During traversal, state accumulated for a child is passed on to each successive child and then to the parents of the group (Group does not push or pop traversal state as separator does).

This node is being deprecated because it is expected that future versions of VRML will not allow properties to "leak out" of group nodes; that is, all group-like nodes will behave like Separators, making Group obsolete.

```
FILE FORMAT/DEFAULTS
     Group {
     }
```

Material Cycling

Previous versions of VRML specified that, if the user did not provide enough material values, that the values that were specified would be re-used. This feature is being deprecated because it is difficult to implement and of little practical value.

TransformSeparator

This group node is similar to the separator node in that it saves state before traversing its children and restores it afterwards. However, it saves only the current transformation; all other state is left as is. This node can be useful for positioning a camera, since the transformations to the camera will not affect the rest of the world, even through the camera will view the world. Similarly, this node can be used to isolate transformations to light sources or other objects.

This node is being deprecated because it is expected that future versions of VRML will not allow properties to "leak out" of group nodes; that is, all group-like nodes will behave like Separators, making TransformSeparator obsolete.

```
FILE FORMAT/DEFAULTS
     TransformSeparator {
     }
```

Acknowledgments

> I want to thank three people who have been absolutely instrumental in the design process: Brian Behlendorf, whose drive (and disk space) made this process happen; and Tony Parisi and Gavin Bell, the final authors of this specification, who have put in a great deal of design work, ensuring that we have a satisfactory product. My hat goes off to all of them, and to all of you who have made this process a success.

Mark Pesce

> I would like to add a personal note of thanks to Jan Hardenbergh of Oki Advanced Products for his diligent efforts to keep the specification process on track, and his invaluable editing assistance. I would also like to acknowledge Chris Marrin of Silicon Graphics for his timely contributions to the final design.

Tony Parisi

> VRML 1.0 is a result of years of effort from the Inventor group at Silicon Graphics. All of the past and present members of the Inventor team deserve recognition and thanks for their excellent work over the last five years.

Gavin Bell

Jan Hardenbergh and Tom Meyer would like to thank Bernie Roehl, Mitra, Steve Ghee, Greg Scallan, Jim Dunn, Jon Marbry, Jim Doubek, Brian Blau, Eric Haines, James Waldrop and Mike Heck for help with the clarifications.