The Virtual Reality Modeling Language Specification

Version 2.0

August 4, 1996

The VRML 2.0 Specification

An Overview of VRML

What is VRML?

VRML is an acronym for "Virtual Reality Modeling Language". It is a file format for describing interactive 3D objects and worlds to be experienced on the world wide web (similar to how HTML is used to view text). The first release of The VRML 1.0 Specification was created by Silicon Graphics, Inc. and based on the *Open Inventor* file format. The second release of VRML adds significantly more interactive capabilities. It was designed by the Silicon Graphics' VRML 1.0 team with contributions from Sony Research and Mitra. VRML 2.0 was reviewed by the VRML moderated email discussion group (www-vrml@wired.com), and later adopted and endorsed by a plethora of companies and individuals. See the *San Diego Supercomputer Center's VRML Repository* or *Silicon Graphics' VRML site* for more information.

What is *Moving Worlds*?

Moving Worlds is the name of Silicon Graphics' submission to the Request-for-Proposals for VRML 2.0. It was chosen by the VRML community as the working document for VRML 2.0. It was created by Silicon Graphics, Inc. in collaboration with Sony and Mitra. Many people in the VRML community were actively involved with *Moving Worlds* and contributed numerous ideas, reviews, and improvements.

What is the VRML Specification?

The VRML Specification is the technical document that precisely describes the VRML file format. It is primarily intended for implementors writing VRML browsers and authoring systems. It is also intended for readers interested in learning the details about VRML. Note however that many people (especially non-programmers) find the VRML Specification inadequate as a starting point or primer. There are a variety of excellent introductory books on VRML in bookstores.

How was *Moving Worlds* chosen as the VRML 2.0 Specification?

The VRML Architecture Group (VAG) put out a Request-for-Proposals (RFP) in January 1996 for VRML 2.0. Six proposals were received and then debated for about 2 months. *Moving Worlds* developed a strong consensus and was eventually selected by the VRML community in a poll. The VAG made it official on March 27th.

How can I start using VRML 2.0?

You must install a VRML 2.0 browser. The following VRML 2.0 **Draft** browsers or toolkits are available:

- DimensionX's *Liquid Reality toolkit*
- Silicon Graphics' *Cosmo Player for Windows95* browser
- Sony's *CyberPassage* browser

See San Diego Supercomputer Center's VRML Repository for more details on available VRML browsers and tools.

Official VRML 2.0 Specification

Changes from Draft 3 to FINAL
Compressed **PostScript** (xxxk)
Compressed **tar HTML Directory** (xxxk)

Draft 3 VRML 2.0

Changes from Draft #2b --> #3	Compressed (gzip) **Postscript** (880K)
Compressed (gzip) **HTML** (140K)	Uncompressed **HTML** (536K)
Compressed (gzip) **tar HTML dir** (952K)	**PDF format** (thanks to Sandy Ressler)

Draft 2 VRML 2.0

Compressed (gzip) Postscript (404K)	Compressed (gzip) HTML (84K)

The Virtual Reality Modeling Language specification was originally developed by Silicon Graphics, Inc. in collaboration with Sony and Mitra. Many people in the VRML community have been involved in the review and evolution of the specification (see Credits). Moving Worlds VRML 2.0 is a tribute to the successful collaboration of all of the members of the VRML community. Gavin Bell, Rikk Carey, and Chris Marrin have headed the effort to produce the final specification.

Please send errors or suggestions to rikk@best.com, cmarrin@sgi.com, and/or gavin@acm.org.

Related Documents

- VRML 1.0 Specification
- VRML 2.0: Request-for-Proposal from the VAG
- VRML 2.0: Process from the VAG
- VRML 2.0: Polling Results

Related Sites

- VRML Architecture Group (VAG)
- `www-vrml@wired.com email list information`
- VRML FAQ
- San Diego Supercomputer Center VRML Repository
- Silicon Graphics VRML/Cosmo site
- SONY's Virtual Society site
- www-vrml email list archive

Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm.org with questions or comments.

This URL: http://vrml.sgi.com/moving-worlds/index.html

An Overview of the

Virtual Reality Modeling Language

Version 2.0

August 4, 1996

Introduction

This overview provides a brief high-level summary of the VRML 2.0 specification. The purposes of the overview are to give you the general idea of the major features, and to provide a summary of the differences between VRML 1.0 and VRML 2.0. The overview consists of two sections:

- Summary of VRML 2.0 Features
- Changes from VRML 1.0

This overview assumes that readers are at least vaguely familiar with VRML 1.0. If you're not, read the introduction to the official VRML 1.0 specification. Note that VRML 2.0 includes some changes to VRML 1.0 concepts and names, so although you should understand the basic idea of what VRML is about, you shouldn't hold on too strongly to details and definitions from 1.0 as you read the specification.

The VRML 2.0 specification is available at: http://vrml.sgi.com/moving-worlds/spec/.

Summary of VRML 2.0 Features

VRML 1.0 provided a means of creating and viewing static 3D worlds; VRML 2.0 provides much more.

The overarching goal of VRML 2.0 is to provide a richer, more exciting, more interactive user experience than is possible within the static boundaries of VRML 1.0. The secondary goal is to provide a solid foundation for future VRML expansion to grow from, and to keep things as simple and as fast as possible -- for everyone from browser developers to world designers to end users.

VRML 2.0 provides these extensions and enhancements to VRML 1.0:

- Enhanced static worlds
- Interaction
- Animation
- Scripting
- Prototyping

Each section of this summary contains links to relevant portions of the official specification.

Enhanced Static Worlds

You can add realism to the static geometry of your world using new features of VRML 2.0:

New nodes allow you to create ground-and-sky backdrops to scenes, add distant mountains and clouds, and dim distant objects with fog. Another new node lets you easily create irregular terrain instead of using flat planes for ground surfaces.

VRML 2.0 provides 3D spatial sound-generating nodes to further enhance realism -- you can put crickets, breaking glass, ringing telephones, or any other sound into a scene.

If you're writing a browser, you'll be happy to see that optimizing and parsing files are easier than in VRML 1.0, thanks to a new simplified scene graph structure.

Interaction

No more moving like a ghost through cold, dead worlds: now you can directly interact with objects and creatures you encounter. New sensor nodes set off events when you move in certain areas of a world and when you click certain objects. They even let you drag objects or controls from one place to another. Another kind of sensor keeps track of the passage of time, providing a basis for everything from alarm clocks to repetitive animations.

And no more walking through walls. Collision detection ensures that solid objects react like solid objects; you bounce off them (or simply stop moving) when you run into them. Terrain following allows you to travel up and down steps or ramps.

Animation

VRML2.0 includes a variety of animation objects called Interpolators. This allow you to create pre-defined animations of a many aspects of the world and then play it at some opportune time. With animation interpolators you can create moving objects such as flying birds, automatically opening doors, or walking robots, objects that change color as they move, such as the sun, objects that morph their geometry from one shape to another, and you can create guided tours that automatically move the user along a predefined path.

Scripting

VRML 2.0 wouldn't be able to move without the new Script nodes. Using Scripts, you can not only animate creatures and objects in a world, but give them a semblance of intelligence. Animated dogs can fetch newspapers or frisbees; clock hands can move; birds can fly; robots can juggle.

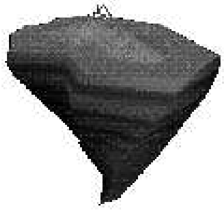
These effects are achieved by means of events; a script takes input from sensors and generates events based on that input which can change other nodes in the world. Events are passed around among nodes by way of special statements called routes.

Prototyping

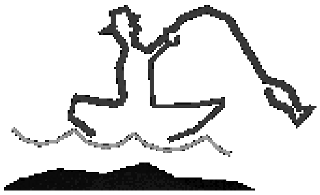
Have an idea for a new kind of geometry node that you want everyone to be able to use? Got a nifty script that you want to turn into part of the next version of VRML? In VRML 2.0, you can encapsulate a group of nodes together as a new node type, a prototype, and then make that node type available to anyone who wants to use it. You can then create instances of the new type, each with different field values -- for instance, you could create a Robot prototype with a robotColor field, and then create as many individual different-colored Robot nodes as you like.

Example

So how does all this fit together? Here's a look at possibilities for implementing a fully-interactive demo world called Gone Fishing.



In Gone Fishing, you start out hanging in space near a floating worldlet. If you wanted a more earthbound starting situation, you could (for instance) make the worldlet an island in the sea, using a Background node to show shaded water and sky meeting at the horizon as well as distant unmoving geometry like mountains. You could also add a haze in the distance using the fog parameters in a Fog node.



As you approach the little world, you can see two neon signs blinking on and off to attract you to a building. Each of those signs consists of two pieces of geometry under a Switch node. A TimeSensor generates time events which a

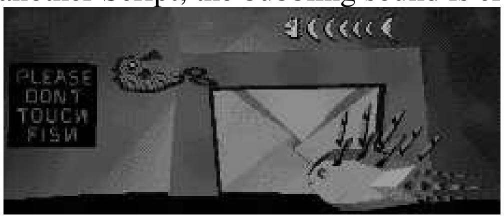
Script node picks up and processes; the Script then sends other events to the Switch node telling it which of its children should be active. All events are sent from node to node by way of ROUTE statements.

As you approach the building -- a domed aquarium on a raised platform -- you notice that the entry portals are closed. There appears to be no way in, until you click the front portal; it immediately slides open with a motion like a camera's iris. That portal is attached to a TouchSensor that detects your click; the sensor tells a Script node that you've clicked, and

the Script animates the opening portal, moving the geometry for each piece of the portal a certain amount at a time. The script writer only had to specify certain key frames of the animation; interpolator nodes generate intermediate values to provide smooth animation between the key frames. The door, by the way, is set up for collision detection using a Collision node, so that without clicking to open it you'd never be able to get in.

You enter the aquarium and a light turns on. A ProximitySensor node inside the room noticed you coming in and sent an event to, yes, another Script node, which told the light to turn on. The sensor, script, and light can also easily be set up to darken the room when you leave.

Inside the aquarium, you can see and hear bubbles drifting up from the floor. The bubbles are moved by another Script; the bubbling sound is created by a PointSound node. As you move further into the building and closer to the bubbles, the bubbling sound gets louder.



Besides the bubbles, which always move predictably upward, three fish swim through the space inside the building. The fish could all be based on a single Fish node type, defined in this file by a PROTO statement as a collection of geometry, appearance, and behavior; to create new kinds of fish, the world builder could just plug in new geometry or behavior.

Proximity sensors aren't just for turning lights on and off; they can be used by moving creatures as well. For example, the fish could be programmed (using a similar ProximitySensor/Script/ROUTE combination to the one described above) to avoid you by swimming away whenever you got too close. Even that behavior wouldn't save them from users who don't follow directions, though:

Despite (or maybe because of) the warning sign on the wall, most users "touch" one or more of the swimming fish by clicking them. Each fish behaves differently when touched; one of them swims for the door, one goes belly-up. These behaviors are yet again controlled by Script nodes.

To further expand Gone Fishing, a world designer might allow users to "pick up" the fish and move them from place to place. This could be accomplished with a PlaneSensor node, which translates a user's click-and-drag motion into translations within the scene. Other additions -- sharks that eat fish, tunnels for the fish to swim through, a kitchen to cook fish dinners in, and so on -- are limited only by the designer's imagination.

Gone Fishing is just one example of the sort of rich, interactive world you can build with VRML 2.0. For details of the new nodes and file structure, see the "Concepts" section of the VRML 2.0 Specification.

Changes from VRML 1.0

This section provides a very brief list of the changes to the set of predefined node types for VRML 2.0. It briefly describes all the newly added nodes, summarizes the changes to VRML 1.0 nodes, and lists the

VRML 1.0 nodes that have been deleted in VRML 2.0. (For fuller descriptions of each node type, click the type name to link to the relevant portion of the VRML 2.0 specification proposal.) Finally, this document briefly describes the new field types in VRML 2.0.

New Node Types

The new node types are listed by category:

- Grouping Nodes
- Browser Information
- Lights and Lighting
- Sound
- Shapes
- Geometry
- Appearance
- Geometric Sensors
- Special Nodes

Grouping Nodes

Collision
: Tells the browser whether or not given pieces of geometry can be navigated through.

Transform
: Groups nodes together under a single coordinate system, or "frame of reference"; incorporates the fields of the VRML 1.0 Separator and Transform nodes.

Browser Information

In place of the old Info node type, VRML 2.0 provides several new node types to give specific information about the scene to the browser:

Background
: Provides a shaded plane and/or distant geometry to be used as a backdrop, drawn behind the displayed scene.

NavigationInfo
: Provides hints to the browser about what kind of viewer to use (walk, examiner, fly, etc.), suggested average speed of travel, a radius around the camera for use by collision detection, and an indication of whether the browser should turn on a headlight.

Viewpoint
: Specifies an interesting location in a local coordinate system from which a user might wish to view the scene. Replaces the former PerspectiveCamera node.

WorldInfo
: Provides the scene's title and other information about the scene (such as author and copyright information), in a slightly more structured manner than a VRML 1.0 Info node.

Lights and Lighting

Fog
: Describes a variety of atmospheric effects such as fog, haze, and smoke.

Sound

Sound
: Defines a sound source that emits sound primarily in a 3D space.

Shapes

Shape
: A node whose fields specify a set of geometry nodes and a set of property nodes to apply to the geometry.

Geometry

ElevationGrid
: Provides a compact method of specifying an irregular "ground" surface.

Extrusion
: A compact representation of extruded shapes and solids of rotation.

Text
: Replaces VRML 1.0's AsciiText node; has many more options, to allow easy use of non-English text.

Geometric Properties

Color
: Defines a set of RGB colors to be used in the *color* fields of various geometry nodes.

Appearance

Appearance
: Gathers together all the appearance properties for a given Shape node.

Sensors

ProximitySensor
: Generates events when the camera moves within a bounding box of a specified size around a specified point.

TouchSensor
: Generates events when the user moves the pointing device across an associated piece of geometry, and when the user clicks on said geometry.

CylinderSensor
: Generates events that interpret a user's click-and-drag on a virtual cylinder.

PlaneSensor
: Generates events that interpret a user's click-and-drag as translation in two dimensions.

SphereSensor

Generates events that interpret a user’s click-and-drag on a virtual sphere.

VisibilitySensor
: Generates events as a regions enters and exits rendered view.

TimeSensor
: Generates events at a given time or at given intervals.

Scripting

Script
: Contains a program which can process incoming events and generating outgoing ones.

Interpolator Nodes

ColorInterpolator
: Interpolates intermediate values from a given list of color values.

CoordinateInterpolator
: Interpolates intermediate values from a given list of 3D vectors.

NormalInterpolator
: Interpolates intermediate normalized vectors from a given list of 3D vectors.

OrientationInterpolator
: Interpolates intermediate absolute rotations from a given list of rotation amounts.

PositionInterpolator
: Interpolates intermediate values from a given list of 3D vectors, suitable for a series of translations.

ScalarInterpolator
: Interpolates intermediate values from a given list of floating-point numbers.

Changed Node Types

Almost all node types have been changed in one way or another -- if nothing else, most can now send and receive simple events. The most far-reaching changes, however, are in the new approaches to grouping nodes: in particular, Separators have been replaced by Transforms, which incorporate the fields of the now-defunct Transform node, and Groups no longer allow state to leak. The other extensive changes are in the structure of geometry-related nodes (which now occur only as fields in a Shape node). See the section of the spec titled "Structuring the Scene Graph" for details.

Deleted Node Types

The following VRML 1.0 node types have been removed from VRML 2.0:

- AsciiText: replaced with Text
- Info: replaced with WorldInfo
- OrthographicCamera: shifted to browser UI responsibility (that is, browsers may provide an orthographic view of a world as an option)
- PerspectiveCamera: replaced with Viewpoint.
- Separator: use Transform instead
- transformation nodes: incorporated into Transform

- MatrixTransform
- Transform
- Translation
- Rotation
- Scale

New Field Types

In addition to all of the other changes, VRML 2.0 introduces a couple of new field types:

- An SFInt32 field (formerly SFLong) contains a 32-bit integer. An MFInt32 field contains a list of 32-bit integers.
- An SFNode field contains a node (or rather, a pointer to a node). An MFNode field contains a list of pointers to nodes.
- An SFTime field contains a double-precision floating point value indicating a number of seconds since 00:00:00 Jan 1, 1970 GMT.

Contact rikk@best.com. cmarrin@sgi.com, or gavin@acm.org with questions or comments.

This URL: http://vrml.sgi.com/moving-worlds/Overview/overview.main.html.

The Virtual Reality Modeling Language Specification

Version 2.0, ISO/IEC WD 14772

August 4, 1996

This document is the official and complete specification of the **Virtual Reality Modeling Language**, (VRML), Version 2.0.

The ***Foreword*** provides background on the standards process for VRML, and ***Introduction*** describes the conventions used in the specification. The following annexes define the specifications for VRML:

1. ***Scope*** defines the problems that VRML addresses.
2. ***References*** lists the normative standards referenced in the specification.
3. ***Definitions*** contains the glossary of terminology used in the specification.
4. ***Concepts*** describes various fundamentals of VRML.
5. ***Nodes*** defines the syntax and semantics of VRML.
6. ***Fields*** specifies the datatype primitives used by nodes.
7. ***Conformance*** describes the minimum support requirements for a VRML implementation.

There are several appendices included in the specification:

- *A.* ***Grammar*** presents the BNF for the VRML file format.
- *B.* ***Examples*** includes a variety of VRML example files.
- *C.* ***Java*** describes how VRML scripting integrates with Java.
- *D.* ***JavaScript*** describes how VRML scripting integrates with JavaScript.
- *E.* ***Bibliography*** lists the informative, non-standard topics referenced in the specification.
- *F.* The ***Index*** lists the concepts, nodes, and fields in alphabetical order.

The ***Document Change Log*** summarizes significant changes to this document and ***Credits*** lists the major contributors to this document:

Document change log	**Credits**

Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm.org with questions or comments.

The Virtual Reality Modeling Language

Foreword

Version 2.0, ISO/IEC WD 14772

August 4, 1996

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 14772 was prepared by Joint Technical Committee ISO/IEC JTC 1, Information Technology Sub-Committee 24, Computer Graphics and Image Processing in collaboration with the VRML Architecture Group (VAG, http://vag.vrml.org) and the VRML moderated email list (www-vrml@wired.com).

ISO/IEC 14772 is a single part standard, under the general title of Information Technology - Computer Graphics and Image Processing - Virtual Reality Modeling Language (VRML).

The Virtual Reality Modeling Language

Introduction

Version 2.0, ISO/IEC WD #14772

August 4, 1996

Purpose

The Virtual Reality Modeling Language (VRML) is a file format for describing 3D interactive worlds and objects. It may be used in conjunction with the World Wide Web. It may be used to create three-dimensional representations of complex scenes such as illustrations, product definition and virtual reality presentations.

Design Criteria

VRML has been designed to fulfill the following requirements:

Authorability
: Make it possible to develop application generators and editors, as well as to import data from other industrial formats.

Completeness
: Provide all information necessary for implementation and address a complete feature set for wide industry acceptance.

Composability
: The ability to use elements of VRML in combination and thus allow re-usability.

Extensibility
: The ability to add new elements.

Implementability
: Capable of implementation on a wide range of systems.

Multi-user potential
: Should not preclude the implementation of multi-user environments.

Orthogonality
: The elements of VRML should be independent of each other, or any dependencies should be structured and well defined.

Performance

The elements should be designed with the emphasis on interactive performance on a variety of computing platforms.

Scalability

The elements of VRML should be designed for infinitely large compositions.

Standard practice

Only those elements that reflect existing practice, that are necessary to support existing practice, or that are necessary to support proposed standards should be standardized.

Well-structured

An element should have a well-defined interface and a simply stated unconditional purpose. Multipurpose elements and side effects should be avoided.

Characteristics of VRML

VRML is capable of representing static and animated objects and it can have hyperlinks to other media such as sound, movies, and image. Interpreters (browsers) for VRML are widely available for many different platforms as well as authoring tools for the creation VRML files.

VRML supports an extensibility model that allows new objects to be defined and a registration process to allow application communities to develop interoperable extensions to the base standard. There is a mapping between VRML elements and commonly used 3D application programmer interface (API) features.

Conventions used in the specification

Field names are in *italics*. File format and api are in **`bold, fixed-spacing`**.
New terms are in *italics*.

The Virtual Reality Modeling Language

1. Scope and Field of Application

Version 2.0, ISO/IEC 14772

August 4, 1996

1. Scope and Field of Application

The scope of the standard incorporates the following:

- a mechanism for storing and transporting two-dimensional and three-dimensional data
- elements for representing two-dimensional and three-dimensional primitive information
- elements for defining characteristics of such primitives
- elements for viewing and modeling two-dimensional and three-dimensional information
- a container mechanism for incorporating data from other metafile formats
- mechanisms for defining new elements which extend the capabilities of the metafile to support additional types and forms of information

Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm.org with questions or comments.

This URL: http://vrml.sgi.com/moving-worlds/spec/part1/scope.html

The Virtual Reality Modeling Language

2. References

Version 2.0, ISO/IEC WD 14772

August 4, 1996

This annex contains the list of published standards referenced by the specification. See "*Appendix E. Bibliography*" for a list of informative documents and technology.

[JPEG]	"Joint Photographic Experts Group", International Organization for Standardization, "Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and guidelines" ISO/IEC IS 10918-1, 1991, [http://www.iso.ch/isob/switch-engine-cate.pl?KEYWORDS=10918&searchtype=refnumber, ftp://rtfm.mit.edu/pub/usenet/news.answers/jpeg-faq/part1].
[MIDI]	"Musical Instrument Digital Interface", International MIDI Association, 23634 Emelita Street, Woodland Hills, California 91367 USA, 1983, [ftp://rtfm.mit.edu/pub/usenet/news.answers/music/midi/bibliography].
[MPEG]	"Motion Picture Experts Group", International Organization for Standardization, ISO/IEC IS 11172-1:1993, [http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=11172].
[PNG]	"PNG (Portable Network Graphics), Specification Version 0.96", W3C Working Draft 11-Mar-1996, [http://www.w3.org/pub/WWW/TR/WD-png , http://www.boutell.com/boutell/png/].
[RURL]	"Relative Uniform Resource Locator", IETF RFC 1808, [http://ds.internic.net/rfc/rfc1808.txt].
[URL]	"Uniform Resource Locator", IETF RFC 1738, [http://ds.internic.net/rfc/rfc1738.txt]
[UTF8]	"Information Technology Universal Multiple-Octet Coded Character Set (UCS)", Part 1: Architecture and Basic Multi-Lingual Plane, ISO/IEC 10646-1:1993, [http://www.iso.ch/cate/d18741.html , http://www.dkuug.dk/JTC1/SC2/WG2/docs/n1335].
[WAV]	Waveform Audio File Format, "Multimedia Programming Interface and Data Specification v1.0", Issued by IBM & Microsoft, 1991, [ftp://ftp.cwi.nl/pub/audio/RIFF-format].

Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm.org with questions or comments.

The Virtual Reality Modeling Language

3. Definitions

Version 2.0, ISO/IEC WD 14772

August 4, 1996

appearance node

A node of type Appearance, FontStyle, ImageTexture, Material, MovieTexture, PixelTexture, or TextureTransform. Appearance nodes control the rendered appearance of the geometry nodes with which they are associated.

bindable leaf node

A node of type Background, Fog, NavigationInfo, or Viewpoint. These nodes may have many instances in a scene graph, but only one instance may be active at any instant of time.

children nodes

Nodes which are parented by grouping nodes and thus are affected by the transformations of all ancestors. See "*Concepts - Grouping and Children Nodes*" for list of allowable children nodes.

colour model

Characterization of a colour space in terms of explicit parameters.VRML allows colors to be defined only with the RGB color model.

display device

A graphics device on which VRML scenes can be represented.

drag sensor

Drag sensors (CylinderSensor, PlaneSensor, SphereSensor) cause events to be generated in response to pointer motions which are sensor-dependent. For example, the SphereSensor generates spherical rotation events. See "*Concepts - Drag Sensors*" for details.

event

Messages sent from one node to another as defined by a Route. Events signal changes to field values, external stimuli, interactions between nodes, etc.

exposed field

A field which can receive events to change its value(s) and generates events when its value(s) change.

execution model

The characterization of the way in which scripts execute within the context of VRML.

external prototype

Prototypes defined in external files and referenced by a URL.

field

The parameters that distinguish one node from another of the same type. Fields can contain various kind of data and one or many values.

geometry node

Nodes of type Box, Cone, Cylinder, ElevationGrid, Extrusion, IndexedFaceSet, IndexedLineSet, PointSet, Sphere, and Text which contain mathematical descriptions of three-dimensional points, lines, surfaces, text strings and solid objects.

geometric property node

A node of type Color, Coordinate, Normal, or TextureCoordinate. These nodes define the properties of specific geometry nodes.

geometric sensor node

A node of type ProximitySensor, VisibilitySensor, TouchSensor, CylinderSensor, PlaneSensor, or SphereSensor. These nodes generate events based on user actions, such as a mouse click or navigating close to a particular object.

grouping node

A node of type Anchor, Billboard, Collision, Group, or Transform. These nodes group child nodes and other grouping nodes together and cause the group to exhibit special behavior which is dependent on the node type.

IETF

Internet Engineering Task Force. The organization which develops Internet standards.

instance

An instantiation of a previously defined node created by the USE syntax.

interpolator node

A node of type ColorInterpolator, CoordinateInterpolator, NormalInterpolator, OrientationInterpolator, PositionInterpolator, or ScalarInterpolator.These nodes define a piece-wise linear interpolation of a particular type of value at specified times.

JPEG

Joint Photographic Experts Group.

MIDI

Musical Instrument Digital Interface - a standard for digital music representation.

MIME

Multipurpose Internet Mail Extension used to specify filetyping rules for browsers. See "*Concepts - File Extension and MIME Types*" for details.

node

The fundamental component of a scene graph in VRML. Nodes are abstractions of various real-world objects and concepts. Examples include spheres, lights, and material descriptions. Nodes contain fields, and events. Messages are sent between nodes via routes.

node type

A required parameter for each node that describes, in general, its particular semantics. For example, Box, Group, Sound, and SpotLight. See "*Concepts - Nodes, Fields, and Events*" and "*Nodes Reference*" for details.

prototype

The definition of a new node type in terms of the nodes defined in this standard.

RGB

The VRML colour model. Each colour is represented as a combination of the three primary colours red, green, and blue.

route

The connection between a node generating an event and a node receiving an event.

scene graph

An ordered collection of grouping nodes and leaf nodes. Grouping nodes, such as Transform, LOD, and Switch nodes, can have child nodes. These children can be other grouping nodes or leaf nodes, such as shapes, browser information nodes, lights, viewpoints, and sounds.

sensor node

A node of type Anchor, CylinderSensor, PlaneSensor, ProximitySensor, SphereSensor, TimeSensor, TouchSensor, or VisibilitySensor. These nodes detect changes and generate events. Geometric sensor nodes generate events based on user actions, such as a mouse click or navigating close to a particular object. TimeSensor nodes generate events at regular intervals in time.

special group node

A node of type LOD (level of detail), InLine, or Switch. These nodes are grouping nodes which exhibit special behavior, such as selecting one of many children to be rendered based on a dynamically changing parameter value or dynamically loading its children from an external file.

texture coordinates

The set of 2D coordinates used by vertex-based geometry nodes (e.g. IndexedFaceSet and ElevationGrid) and specified in the TextureCoordinate node to map textures to the vertices of some geometry nodes. Texture coordinates range from 0 to 1 across the texture image.

texture transform

A node which defines a 2D transformation that is applied to texture coordinates.

URL

Uniform Resource Locator as defined in IETF RFC 1738.

URN

Uniform Resource Name

VRML document server

An application that locates and transmits VRML files and supporting files to VRML client applications (browsers).

VRML file

A file containing information encoded according to this standard.

The Virtual Reality Modeling Language Specification

4. Concepts

Version 2.0, ISO/IEC WD 14772

August 4, 1996

This section describes key concepts relating to the definition and use of the VRML specification. This includes how nodes are combined into scene graphs, how nodes receive and generate events, how to create node types using prototypes, how to add node types to VRML and export them for use by others, how to incorporate programmatic scripts into a VRML file, and various general topics on nodes.

4.1 File Syntax and Structure

4.1.1 Syntax Basics

For easy identification of VRML files, every VRML 2.0 file must begin with the characters:

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

The identifier utf8 allows for international characters to be displayed in VRML using the UTF-8 encoding of the ISO 10646 standard. Unicode is an alternate encoding of ISO 10646. UTF-8 is explained under the Text node.

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 extra whitespace from a VRML file before transmitting it. WorldInfo nodes should be used for persistent information such as copyrights or author information. To extend the set of existing nodes in VRML 2.0, use prototypes or external prototypes rather than named information nodes.

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

After the required header, a VRML file can contain any combination of the following:

- Any number of prototypes (see "*Prototypes*")
- Any number of *children* nodes (see "*Grouping and Children Nodes*")
- Any number of ROUTE statements (see "*Routes*")

See the "*Grammar Reference*" annex for precise grammar rules.

Field, event, prototype, and node names must not begin with a digit (0x30-0x39) but may otherwise contain any characters except for non-printable ASCII characters (0x0-0x20), double or single quotes (0x22: ", 0x27: '), sharp (0x23: #), plus (0x2b: +), comma (0x2c: ,), minus (0x2d: -), period (0x2e: .), square brackets (0x5b, 0x5d: []), backslash (0x5c: \) or curly braces (0x7b, 0x7d: {}). Characters in names are as specified in ISO 10646, and are encoded using UTF-8. VRML is case-sensitive; "Sphere" is different from "sphere" and "BEGIN" is different from "begin."

The following reserved keywords shall not be used for node, PROTO, EXTERNPROTO, or DEF names:

DEF	**EXTERNPROTO**	**FALSE**	**IS**	**NULL**	**PROTO**	**ROUTE**
TO	**TRUE**	**USE**	**eventIn**	**eventOut**	**exposedField**	**field**

4.1.2 File Syntax vs. Public Interface

In this document, the first item in a node specification is the public interface for the node. The syntax for the public interface is the same as that for that node's prototype. This interface is the definitive specification of the fields, events, names, types, and default values for a given node. Note that this syntax is not the actual file format syntax. However, the parts of the interface that are identical to the file syntax are in **bold**. For example, the following defines the `Collision` node's public interface and file format:

```
Collision {
  eventIn      MFNode   addChildren
  eventIn      MFNode   removeChildren
  exposedField MFNode   children        []
  exposedField SFBool   collide         TRUE
  field        SFVec3f  bboxCenter      0 0 0
  field        SFVec3f  bboxSize        -1 -1 -1
  field        SFNode   proxy           NULL
  eventOut     SFTime   collideTime
}
```

Fields that have associated implicit *set_* and *_changed* events are labeled **exposedField**. For example, the *on* field has an implicit *set_on* input event and an *on_changed* output event. Exposed fields may be connected using **ROUTE** statements, and may be read and/or written by Script nodes. Also, any exposedField or EventOut name can be prefixed with *get_* to indicate a read of the current value of the eventOut. This is used only in Script nodes or when accessing the VRML world from an external API.

Note that this information is arranged in a slightly different manner in the actual file syntax. The keywords "`field`" or "`exposedField`" and the types of the fields (e.g. `SFColor`) are not specified when expressing a node in the file format. An example of the file format for the `Collision` node is:

```
Collision {
  children        []
  collide         TRUE
  bboxCenter      0 0 0
  bboxSize        -1 -1 -1
  proxy           NULL
}
```

The rules for naming fields, exposedFields, eventOuts and eventIns for the built-in nodes are as follows:

- All names containing multiple words start with a lower case letter and the first letter of all subsequent words are capitalized (e.g. *bboxCenter*), with the exception of *get_* and *_changed* described below.
- All eventIns have the prefix "*set_*" - with the exception of the *addChildren* and *removeChildren* eventIns.
- All eventOuts have the suffix "*_changed*" appended - with the exception of eventOuts of type SFBool. Boolean eventOuts begin with the word "*is*" (e.g. *isFoo*) for better readability.
- All eventIns and eventOuts of type SFTime do not use the "*set_*" prefix or "*_changed*" suffix.

User defined field names found in Script and PROTO nodes are recommended to follow these naming conventions, but are not required.

4.1.3 URLs and URNs

A *URL* (Uniform Resource Locator) [URL] specifies a file located on a particular server and accessed through a specified protocol (e.g. http). A *URN* (Uniform Resource Name) [URN] provides a more abstract way to refer to data than is provided by a URL.

All URL/URN fields are of type MFString. The strings in the field indicate multiple locations to look for data, in decreasing order of preference. If the browser cannot locate the first URL/URN or doesn't support the protocol type, then it may try the second location, and so on. Note that the URL and URN field entries are delimited by " ", and due to the "*Data Protocol* "and the *"Scripting Language Protocols*" are a superset of the standard URL syntax (IETF RFC 1738). Browsers may skip to the next URL/URN by searching for the closing, un-escaped ". See "*Field and Event Reference - SFString and MFString*" for details on the string field.

URLs are described in "*Uniform Resource Locator*", IETF RFC 1738, http://ds.internic.net/rfc/rfc1738.txt.

Relative URLs are handled as described in "*Relative Uniform Resource Locator*", IETF RFC 1808, http://ds.internic.net/rfc/rfc1808.txt.

VRML 2.0 browsers are not required to support URNs. If they do not support URNs, they should ignore any URNs that appear in MFString fields along with URLs.

See "*URN's*" for more details on URNs.

4.1.4 Data Protocol

The IETF is in the process of standardizing a "Data:" URL to be used for inline inclusion of base64 encoded data, such as JPEG images. This capability should be supported as specified in: "*Data: URL scheme*", http://www.internic.net/internet-drafts/draft-masinter-url-data-01.txt., [DATA]. Note that this is an Internet Draft, and the specification may (but is unlikely to) change.

4.1.5 Scripting Language Protocols

The Script node's URL field may also support a custom protocol for the various scripting languages. For example, a script URL prefixed with *javascript:* shall contain JavaScript source, with newline characters allowed in the string. A script prefixed with *javabc:* shall contain Java bytecodes using a base64 encoding. The details of each language protocol are defined in the appendix for each language. Browsers are not required to support any specific scripting language, but if they do then they shall adhere to the protocol for that particular scripting language. The following example, illustrates the use of mixing custom protocols and standard protocols in a single url (order of precedence determines priority):

```
#VRML V2.0 utf8
Script {
    url [ "javascript: ...",            # custom protocol JavaScript
          "http://bar.com/foo.js",       # std protocol JavaScript
          "http://bar.com/foo.class" ]   # std protocol Java byte
}
```

4.1.6 File Extension and Mime Type

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

The official MIME type for VRML files is defined as:

```
model/vrml
```

where the MIME major type for 3D data descriptions is `model`, and the minor type for VRML documents is `vrml`.

For historical reasons (VRML 1.0) the following MIME type must also be supported:

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

where the MIME major type is `x-world`, and the minor type for VRML documents is `x-vrml`.

IETF work-in-progress on this subject can be found in "*The Model Primary Content Type for Multipurpose Internet Mail Extensions*", (ftp://ds.internic.net/internet-drafts/draft-nelson-model-mail-ext-01.txt).

4.1.7 URN's

URN's are location independent pointers to a file, or to different representations of the same content. In most ways they can be used like URL's except that when fetched a smart browser should fetch them from the closest source. While URN resolution over the net has not been standardized yet, they may be used now as persistent unique identifiers for files, prototypes, textures etc. For more information on the standardization effort see: http://services.bunyip.com:8000/research/ietf/urn-ietf/ . VRML 2.0 browsers are not required to support URN's however they are required to ignore them if they do not support them.

URN's may be assigned by anyone with a domain name, for example if the company Foo owns foo.com then it may allocate URN's that begin with "urn:inet:foo.com:" such as, for example "urn:inet:foo.com:texture/wood01". No special semantics are required of the string following the prefix, except that they should be lower case, and characters should be "URL" encoded as specified in RFC1738.

To reference a texture, proto or other file by URN it should be included in the url field of another node, for example:

```
ImageTexture {
    url [ "http://www.foo.com/textures/woodblock_floor.gif",
          "urn:inet:foo.com:textures/wood001" ]
}
```

specifies a URL file as the first choice and URN as the second choice. Note that until URN resolution is widely deployed, it is advisable to include a URL alternative whenever a URN is used. See http://earth.path.net/mitra/papers/vrml-urn.html for more details and recommendations.

4.2 Nodes, Fields, and Events

4.2.1 Introduction

At the highest level of abstraction, VRML is simply a file format for describing objects. Theoretically, the objects can contain anything -- 3D geometry, MIDI data, JPEG images, and so on. VRML defines a set of objects useful for doing 3D graphics, multi-media, and interactive object/world building. These objects are called *nodes*, and contain elemental data which is stored in *fields* and *events.*

4.2.2 General Node Characteristics

A node has the following characteristics:

- **A type name -** This is a name like Box, Color, Group, Sphere, Sound, SpotLight, and so on.
- **The parameters that distinguish a node from other nodes of the same type -** For example, each Sphere node might have a different radius, and different spotlights have different intensities, colors, and locations. These parameters are called fields. A node can have 0 or more fields. Each node specification 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 does not matter. For example, "Cone { bottomRadius 1 height 6 }" and "Cone { height 6 bottomRadius 1}" are equivalent. There are two kinds of fields: ***field*** and ***exposedField***. *Fields* define the initial values for the node's state, but cannot be changed and are considered private. *ExposedFields* also define the initial value for the node's state, but are public and may be modified by other nodes.
- **A set of associated events that nodes can receive and send -** Nodes can receive a number of incoming *set_* events, denoted as ***eventIn***, (such as *set_position*, *set_color*, and *set_on*), which typically change the node. Nodes can also send out a number of *_changed* events, denoted as ***eventOut***, which indicate that something in the node has changed (for example, *position_changed*, *color_changed*, *on_changed*). The *exposedField* keyword may be used as a short-hand for specifying that a given field has a *set_* eventIn that is directly wired to a field value and a *_changed* eventOut. For example, the declaration:

  ```
  exposedField foo
  ```

  is equivalent to the declaration:

  ```
  eventIn set_foo
  field foo
  eventOut foo_changed
  ```

  where *set_foo,* if written to, automatically sets the value of the field *foo* and generates a *foo_changed* eventOut.

The file syntax for representing nodes is as follows:

```
nodetype { fields }
```

Only the node type and braces are required; nodes may or may not have field values specified. Unspecified field values are set to the default values in the specification.

4.3 The Structure of the Scene Graph

This section describes the general scene graph hierarchy, how to reuse nodes within a file, coordinate systems and transformations in VRML files, and the general model for viewing and interaction within a VRML world.

4.3.1 Grouping and Children Nodes

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

Anchor	**LOD**	**Sound**
Background	**NavigationInfo**	**SpotLight**
Billboard	**NormalInterpolator**	**SphereSensor**
Collision	**OrientationInterpolator**	**Switch**
ColorInterpolator	**PlaneSensor**	**TimeSensor**
CoordinateInterpolator	**PointLight**	**TouchSensor**
CylinderSensor	**PositionInterpolator**	**Transform**
DirectionalLight	**ProximitySensor**	**Viewpoint**
Fog	**ScalarInterpolator**	**VisibilitySensor**
Group	**Script**	**WorldInfo**
Inline	**Shape**	**PROTO'd child nodes**

All grouping nodes also have *addChildren* and *removeChildren* eventIn definitions. The *addChildren* event adds the node(s) passed in to the grouping node's *children* field. Any nodes passed to the *addChildren* event that are already in the group's children list are ignored. The *removeChildren* event removes the node(s) passed in from the grouping node's *children* field. Any nodes passed in the *removeChildren* event that are not in the grouping nodes's *children* list are ignored.

The following nodes are grouping nodes:

Anchor	**Billboard**	**Collision**	**Group**	**Transform**

4.3.2 Instancing

A node may be referenced in a VRML file multiple times. This is called *instancing* (using the same instance of a node multiple times; called "sharing", "aliasing" or "multiple references" by other systems) and is accomplished by using the DEF and USE keywords.

The DEF keyword defines a node’s name and creates a node of that type. The USE keyword indicates that a reference to a previously named node should be inserted into the scene graph. This has the affect of sharing a single node in more than one location in the scene. If the node is modified, then all references to that node are modified. DEF/USE name scope is limited to a single file. If multiple nodes are given the same name, then the last DEF encountered during parsing is used for USE definitions.

Tools that create VRML files may need to modify user-defined node names to ensure that a multiply instanced node with the same name as some other node will be read correctly. The recommended way of doing this is to append an underscore followed by an integer to the user-defined name. Such tools should automatically remove these automatically generated suffixes when VRML files are read back into the tool (leaving only the user-defined names).

Similarly, if an un-named node is multiply instanced, tools will have to automatically generate a name to correctly write the VRML file. The recommended form for such names is just an underscore followed by an integer.

4.3.3 Standard Units

VRML provides no capability to define units of measure. All linear distances are assumed to be in meters and all angles are in radians. Time units are specified in seconds. Colors are specified in the RGB (Red-Green-Blue) color space and are restricted to the 0.0 to 1.0 range.

4.3.4 Coordinate Systems and Transformations

VRML uses a Cartesian, right-handed, 3-dimensional coordinate system. By default, objects are projected onto a 2-dimensional display 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 modeling transformation (Transform and Billboard) or viewing transformation (Viewpoint) can be used to alter this default projection.

Scenes may contain an arbitrary number of *local* (or *object-space*) *coordinate systems*, defined by the transformation fields of the Transform and Billboard nodes.

Conceptually, VRML also has a *world coordinate system*. The various local coordinate transformations map objects into the world coordinate system, which is where the scene is assembled. Transformations accumulate downward through the scene graph hierarchy, with each Transform and Billboard inheriting transformations of their parents. (Note however, that this series of transformations *takes effect* from the leaf nodes *up* through the hierarchy. The local transformations closest to the Shape object take effect first, followed in turn by each successive transformation upward in the hierarchy.)

4.3.5 Viewing Model

This specification assumes that there is a real person viewing and interacting with the VRML world. The VRML author may place any number of *viewpoints* in the world -- interesting places from which the user might wish to view the world. Each viewpoint is described by a Viewpoint node. Viewpoints exist in a specific coordinate system, and both the viewpoint and the coordinate system may be animated. Only one Viewpoint may be active at a time. See the description of "*Bindable Children Nodes*" for details. When a viewpoint is activated, the browser parents its view (or camera) into the scene graph under the currently active viewpoint. Any changes to the coordinate system of the viewpoint have effect on the browser view. Therefore, if a user teleports to a viewpoint that is moving (one of its parent coordinate systems is being animated), then the user should move along with that viewpoint. It is intended, but not required, that browsers support a user-interface by which users may "teleport" themselves from one viewpoint to another.

4.3.6 Bounding Boxes

Several of the nodes in this specification include a bounding box field. This is typically used by grouping nodes to provide a hint to the browser on the group's approximate size for culling optimizations. The default size for bounding boxes (-1, -1, -1) implies that the user did not specify the bounding box and the browser must compute it or assume the most conservative case. A *bboxSize* value of (0, 0, 0) is valid and represents a point in space (i.e. infinitely small box). Note that the bounding box of may change as a result of changing children. The *bboxSize* field values must be >= 0.0. Otherwise, results are undefined. The *bboxCenter* fields specify a translation offset from the local coordinate system and may be in the range: *-infinity* to *+infinity*.

The *bboxCenter* and *bboxSize* fields may be used to specify a maximum possible bounding box for the objects inside a grouping node (e.g. Transform). These are used as hints to optimize certain operations such as determining whether or not the group needs to be drawn. If the specified bounding box is smaller than the true bounding box of the group, results are undefined. The bounding box should be large enough to completely contain the effects of all sounds, lights and fog nodes that are children of this group. If the size of this group may change over time due to animating children, then the bounding box must also be large enough to contain all possible animations (movements). The bounding box should typically be the union of the group's children bounding boxes; it should not include any transformations performed by the group itself (i.e. the bounding box is defined in the local coordinate system of the group).

4.4 Events

Most nodes have at least one eventIn definition and thus can receive *events*. Incoming events are data messages sent by other nodes to change some state within the receiving node. Some nodes also have eventOut definitions. These are used to send data messages to destination nodes that some state has changed within the source node.

If an eventOut is read before it has sent any events (e.g. *get_foo_changed*), the *initial value* as specified in "*Field and Event Reference*" for each field/event type is returned.

4.4.1 Routes

The connection between the node generating the event and the node receiving the event is called a *route*. A node that produces events of given type can be routed to a node that receives events of the same type using the following syntax:

```
ROUTE NodeName.eventOutName_changed TO NodeName.set_eventInName
```

The prefix *set_* and the suffix *_changed* are recommended conventions, not strict rules. Thus, when creating prototypes or scripts, the names of the eventIns and the eventOuts may be any legal identifier name. Note however, that exposedField's implicitly define *set_xxx* as an eventIn, *xxx_changed* as an eventOut, and *xxx* as a field for a given exposedField named *xxx*. It is strongly recommended that developers follow these guidelines when creating new types. There are three exceptions in the VRML Specification to this recommendation: Boolean events, Time events, and children events. All SF/MFBool eventIns and eventOuts are named *isFoo* (e.g. *isActive*). All SF/MFTime eventIns and eventOuts are named *fooTime* (e.g. *enterTime*). The eventIns on groups for adding and removing children are named: *addChildren* and *removeChildren*. These exceptions were made to improve readability.

Routes are not nodes; ROUTE is merely a syntactic construct for establishing event paths between nodes. ROUTE statements may appear at either the top-level of a .wrl file or prototype implementation, or may appear inside a node wherever fields may appear.

The types of the eventIn and the eventOut must match exactly. For example, it is illegal to route from an SFFloat to an SFInt32 or from an SFFloat to an MFFloat.

Routes may be established only from eventOuts to eventIns. Since exposedField's implicitly define a field, an eventIn, and an eventOut, it is legal to use the exposedField's defined name when routing to and from it, (rather than specifying the *set_* prefix and *_changed* suffix). For example, the following TouchSensor's *enabled* exposedField is routed to the DirectionalLight's *on* exposed field. Note that each of the four routing examples below are legal syntax:

```
    DEF CLICKER TouchSensor { enabled TRUE }
    DEF LIGHT DirectionalLight { on  FALSE }
    ROUTE CLICKER.enabled TO LIGHT.on
or
    ROUTE CLICKER.enabled_changed TO LIGHT.on
or
    ROUTE CLICKER.enabled TO LIGHT.set_on
or
    ROUTE CLICKER.enabled_changed TO LIGHT.set_on
```

Redundant routing is ignored. If a file repeats a routing path, the second (and all subsequent identical routes) are ignored. Likewise for dynamically created routes via a scripting language supported by the browser.

4.4.2 Sensors

Sensor nodes generate events. Geometric sensor nodes (ProximitySensor, VisibilitySensor, TouchSensor, CylinderSensor, PlaneSensor, SphereSensor and the Collision group) generate events based on user actions, such as a mouse click or navigating close to a particular object. TimeSensor nodes generate events as time passes. See "*Sensor Nodes*" for more details on the specifics of sensor nodes.

Each type of sensor defines when an event is generated. The state of the scene graph after several sensors have generated events must be as if each event is processed separately, in order. If sensors generate events at the same time, the state of the scene graph will be undefined if the results depends on the ordering of the events (world creators must be careful to avoid such situations).

It is possible to create dependencies between various types of sensors. For example, a TouchSensor may result in a change to a VisibilitySensor’s transformation, which may cause it’s visibility status to change. World authors must be careful to avoid creating indeterministic or paradoxical situations (such as a TouchSensor that is active if a VisibilitySensor is visible, and a VisibilitySensor that is NOT visible if a TouchSensor is active).

4.4.3 Execution Model

Once a Sensor or Script has generated an **initial event**, the event is propagated along any ROUTES to other nodes. These other nodes may respond by generating additional events, and so on. This process is called an **event cascade**. All events generated during a given event cascade are given the same timestamp as the initial event (they are all considered to happen instantaneously).

Some sensors generate multiple events simultaneously; in these cases, each event generated initiates a different event cascade.

4.4.4 Loops

Event cascades may contain **loops**, where an event ’E’ is routed to a node that generated an event that eventually resulted in ’E’ being generated. Loops are broken as follows: implementations must not generate two events from the same eventOut that have identical timestamps. Note that this rule also breaks loops created by setting up cyclic dependencies between different Sensor nodes.

4.4.5 Fan-in and Fan-out

Fan-in occurs when two or more routes write to the same eventIn. If two events with different values but the same timestamp are received at an eventIn, then the results are undefined. World creators must be careful to avoid such situations.

Fan-out occurs when one eventOut routes to two or more eventIns. This case is perfectly legal and results in multiple events sent with the same values and the same timestamp.

4.5 Time

4.5.1 Introduction

The browser controls the passage of time in a world by causing TimeSensors to generate events as time passes. Specialized browsers or authoring applications may cause time to pass more quickly or slowly than in the real world, but typically the times generated by TimeSensors will roughly correspond to "real" time. A world’s creator must make no assumptions about how often a TimeSensor will generate events but can safely assume that each time event generated will be greater than any previous time event.

Time (0.0) starts at 00:00:00 GMT January 1, 1970.

Events that are "in the past" cannot be generated; processing an event with timestamp ’t’ may only result in generating events with timestamps greater than or equal to ‘t’.

4.5.2 Discrete and Continuous Changes

VRML does not distinguish between discrete events (like those generated by a TouchSensor) and events that are the result of sampling a conceptually continuous set of changes (like the fraction events generated by a TimeSensor). An ideal VRML implementation would generate an infinite number of samples for continuous changes, each of which would be processed infinitely quickly.

Before processing a discrete event, all continuous changes that are occurring at the discrete event’s timestamp should behave as if they generate events at that same timestamp.

Beyond the requirements that continuous changes be up-to-date during the processing of discrete changes, implementations are free to otherwise sample continuous changes as often or as infrequently as they choose. Typically, a TimeSensor affecting a visible (or otherwise perceptible) portion of the world will generate events once per "frame," where a "frame" is a single rendering of the world or one time-step in a simulation.

4.6 Prototypes

4.6.1 Introduction to Prototypes

Prototyping is a mechanism that allows the set of node types to be extended from within a VRML file. It allows the encapsulation and parameterization of geometry, attributes, behaviors, or some combination thereof.

A prototype definition consists of the following:

- the PROTO keyword,
- the name of the new node type,

- the *prototype declaration* which contains:
  - a list of public eventIns and eventOuts that can send and receive events
  - a list of public exposedFields and fields, with default values,
- the *prototype definition* which contains a list of one or more nodes, and zero or more routes and prototypes. The nodes in this list may also contain the **IS** syntax associates field and event names contained within the prototype definition with the events and fields names in the prototype declaration.

Square brackets enclose the list of events and fields, and braces enclose the definition itself:

```
PROTO prototypename [ eventIn      eventtypename name
                      eventOut     eventtypename name
                      exposedField fieldtypename name defaultValue
                      field        fieldtypename name defaultValue
                      ... ] {
  Zero or more routes and prototypes
  First node (defines the node type of this prototype)
  Zero or more nodes (of any type), routes, and prototypes
}
```

The names of the fields, exposedFields, eventIns, and eventOuts must be unique for a single prototype (or built-in node). Therefore, the following prototype is illegal:

```
PROTO badNames [ field        SFBool   foo
                 eventOut     SFColor  foo
                 eventIn      SFVec3f  foo
                 exposedField SFString foo ] {...}
```

because the name foo is overloaded. Prototype and built-in node field and event name spaces do not overlap. Therefore, it is legal to use the same names in different prototypes, as follows:

```
PROTO foo  [ field    SFBool   foo
             eventOut SFColor  foo2
             eventIn  SFVec3f  foo3 ] {...}

PROTO bar  [ field    SFBool   foo
             eventOut SFColor  foo2
             eventIn  SFVec3f  foo3 ] {...}
```

A prototype statement does **not** define an actual instance of node in the scene. Rather, it creates a new node type (named *prototypename*) that can be created later in the same file as if it were a built-in node. It is thus necessary to define a node of the type of the prototype to actually create an object. For example, the following file is an empty scene with a *fooSphere* prototype that serves no purpose:

```
#VRML V2.0 utf8
PROTO fooSphere [ field SFFloat fooRadius 3.0 ] {
    Sphere {
        radius 3             # default radius value for fooSphere
        radius IS fooRadius  # associates radius with fooRadius
    }
}
```

In the following example, a *fooSphere* is created and thus produces a visible result:

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

```
PROTO fooSphere [ field SFFloat fooRadius 3.0 ] {
    Sphere {
        radius 3             # default radius value for fooSphere
        radius IS fooRadius  # associates radius with fooRadius
    }
}
fooSphere { fooRadius 42.0 }
```

The first node found in the prototype definition is used to define the node type of this prototype. This first node type determines how instantiations of the prototype can be used in a VRML file. An instantiation is created by filling in the parameters of the prototype declaration and inserting the first node (and its scene graph) wherever the prototype instantiation occurs. For example, if the first node in the prototype definition is a Material node, then instantiations of the prototype can be used wherever a Material can be used. Any other nodes and accompanying scene graphs are not rendered, but may be referenced via routes or scripts (and thus cannot be ignored). The following example defines a *RampMaterial* prototype which animates a Material's *diffuseColor* continuously and that must be used wherever a Material can be used in the file (i.e. within an Appearance node):

```
#VRML V2.0 utf8
PROTO RampMaterial [ field MFColor colors 0 0 0 field SFTime cycle 1 ] {
    DEF M Material {}
    DEF C ColorInterpolator { keyValue IS colors  key ... }
    DEF T TimeSensor { enabled TRUE loop TRUE cycleInterval IS cycle }
    ROUTE T.fraction_changed TO C.set_fraction
    ROUTE C.value_changed TO M.diffuseColor
}

Transform {
    children Shape {
        geometry Sphere {}
        appearance Appearance {
            material RampMaterial {
                colors [ 1 0 0, 0 0 1, 1 0 0 ] # red to green to red
                cycle 3.0                      # 3 second cycle
            }
        }
    }
}
```

The next example defines a *SphereCone* (fused Sphere and Cone) and illustrates how the first node in the prototype definition may contain a complex scene graph:

```
#VRML V2.0 utf8
PROTO SphereCone [ field SFFloat radius    2.0
                   field SFFloat height    5.0
                   field SFNode  sphereApp NULL
                   field SFNode  coneApp   NULL   ] {
    Transform {
        children [
            Shape {
                appearance IS sphereApp
                geometry Sphere { radius IS radius }
            }
            Shape {
                appearance IS coneApp
                geometry Cone { height IS height }
            }
        ]
```

```
    }
}

Transform {
    translation 15 0 0
    children SphereCone {
        radius 5.0
        height 20.0
        sphereApp Appearance { material Material { ... } }
        coneApp Appearance { texture ImageTexture { ... } }
    }
}
Transform {
    translation -10 0 0
    children SphereCone {           # default proto's radius and height
        sphereApp Appearance { texture ImageTexture { ... } }
        coneApp Appearance {  material Material { ... } }
    }
}
```

PROTO and EXTERNPROTO statements may appear anywhere ROUTE statements may appear--either at the top-level of a file or a prototype definition, or wherever fields may appear.

4.6.2 IS Statement

The *eventIn* and *eventOut* prototype declarations receive and send events to and from the prototype's definition. Each eventIn in the prototype declaration is associated with an eventIn or exposedField defined in the prototype's node definition via the *IS* syntax. The eventIn declarations define the events that the prototype can receive. Each eventOut in the prototype declaration is associated with an eventOut or exposedField defined in the prototype's node definition via the IS syntax. The eventOut declarations define the events that the prototype can send. For example, the following statement exposes a Transform node's *set_translation* event by giving it a new name (*set_position*) in the prototype interface:

```
PROTO FooTransform [ eventIn SFVec3f set_position ] {
    Transform { set_translation IS set_position }
}
```

Fields, (exposedField and field), specify the initial state of nodes. Defining fields in a prototype's declaration allows the initial state of associated fields in the prototype definition to be specified when an instance of the prototype is created. The fields of the prototype are associated with fields in the node definition using the **IS** keyword. Field default values must be specified in the prototype declaration. For example:

```
PROTO BarTransform [ exposedField SFVec3f position 42 42 42 ] {
    Transform {
        translation IS position
        translation  100 100 100
    }
}
```

defines a prototype, BarTransform, that specifies the initial values (42, 42, 42) of the *position* exposed field . The *position* field is associated with the *translation* field of the Tranform node in the prototype definition using the IS syntax. Note that the field values in the prototype definition for *translation* (100, 100, 100) are legal, but overridden by the prototype declaration defaults.

Note that in some cases, it is necessary to specify the field defaults inside the prototype definition. For example, the following prototype associates the prototype definition's Material node *diffuseColor* (exposedField) to the prototype declaration's eventIn *myColor* and also defines the default *diffuseColor* values:

```
PROTO foo [ eventIn myColor ] {
    Material {
        diffuseColor  1 0 0
        diffuseColor  IS myColor   # or set_diffuseColor IS myColor
    }
}
```

IS statements may appear inside the prototype definition wherever fields may appear. IS statements must refer to fields or events defined in the prototype declaration. Inversely, it is an error for an IS statement to refer to a non-existent declaration. It is an error if the type of the field or event being associated does not match the type declared in the prototype's interface declaration. For example, it is illegal to associate an SFColor with an SFVec3f, and it is also illegal to associate an SFColor with an MFColor, and vice versa. The following table defines the rules for mapping between the prototype declarations and the primary scene graph's nodes (yes denotes a legal mapping, no denotes an error):

			Prototype	**declaration**	
		exposedField	**field**	**eventIn**	**eventOut**
N	**exposedField**	yes	yes	yes	yes
o	**field**	no	yes	no	no
d	**eventIn**	no	no	yes	no
e	**eventOut**	no	no	no	yes

Specifying the field and event types both in the prototype declaration and in the node definition is intended to prevent user errors and to provide consistency with "*External Prototypes*".

4.6.3 Prototype Scoping Rules

A prototype is instantiated as if *prototypename* were a built-in node. The prototype name must be unique within the scope of the file, and cannot rename a built-in node or prototype.

Prototype instances may be named using DEF and may be multiply instanced using USE as any built-in node. A prototype instance can be used in the scene graph wherever the first node of the primary scene graph can be used. For example, a prototype defined as:

```
    PROTO MyObject [ ... ] {
      Box { ... }
      ROUTE ...
      Script { ... }
```

```
    ...
}
```

may be instantiated wherever a Box may be used (e.g. Shape node's *geometry* field), since the first node of the prototype definition is a Box.

A prototype's scene graph defines a DEF/USE name scope separate from the rest of the scene; nodes DEF'd inside the prototype may not be USE'd outside of the prototype's scope, and nodes DEF'ed outside the prototype scope may not be USE'ed inside the prototype scope.

Prototype definitions appearing inside a prototype implementation (i.e. nested) are local to the enclosing prototype. For example, given the following:

```
PROTO one [...] {
    PROTO two [...] { ... }
    ...
    two { } # Instantiation inside "one":  OK
}
two { } # ERROR: "two" may only be instantiated inside "one".
```

The second instantiation of "two" is illegal. IS statements inside a nested prototype's implementation may refer to the prototype declarations of the innermost prototype. Therefore, IS statements in "two" cannot refer to declarations in "one".

A prototype may be instantiated in a file anywhere after the completion of the prototype definition. A prototype may not be instantiated inside its own implementation (i.e. recursive prototypes are illegal). The following example produces an error:

```
PROTO Foo [] {
    Foo {}
}
```

4.6.4 Defining Prototypes in External Files

The syntax for defining prototypes in external files is as follows:

```
EXTERNPROTO extern prototypename [ eventIn eventtypename name
                                   eventOut eventtypename name
                                   field fieldtypename name
                                   exposedField fieldtypename name ]
  "URL/URN" or [ "URL/URN", "URL/URN", ... ]
```

The external prototype is then given the name *externprototypename* in this file's scope. It is an error if the eventIn/eventOut declaration in the EXTERNPROTO is not a subset of the eventIn/eventOut declarations specified in the PROTO referred to by the URL. If multiple URLs or URNs are specified, the browser searches in the order of preference (see "*URLs and URNs*").

Unlike a prototype, an external prototype does not contain an inline implementation of the node type. Instead, the prototype implementation is fetched from a URL or URN. The other difference between a prototype and an external prototype is that external prototypes do not contain default values for fields. The external prototype references a file that contains the prototype implementation, and this file contains the field default values.

The URL/URNs refer to legal VRML files in which the first prototype found in the file is used to define the external prototype's definition. Note that the *prototypename* does not need to match the *externprotoname*. The following example illustrates how an external prototype's declaration may be a subset of the prototype's declaration (*diff* vs. *diffuse* and *shiny*) and how the external prototype's typename may differ from the prototype's typename (e.g. *FooBar* != *SimpleMaterial*):

```
foo.wrl:
-------
#VRML V2.0 utf8
EXTERNPROTO FooBar [ eventIn SFColor diff ] "http://foo.com/coolNode.wrl
...


http://foo.com/coolNode.wrl:
---------------------------
#VRML V2.0 utf8
PROTO SimpleMaterial [ exposedField SFColor diffuse 1 0 0
                       eventIn      SFFloat shiny   0.5   ]
{
    Material { ... }
}
```

To allow the creation of libraries of small, reusable PROTO definitions, browsers shall recognize EXTERNPROTO URLs that end with "#*name*" to mean the prototype definition of "name" in the given file. For example, a library of standard materials might be stored in a file called "materials.wrl" that looks like:

```
#VRML V2.0 utf8
PROTO Gold   [] { Material { ... } }
PROTO Silver [] { Material { ... } }
...etc.
```

A material from this library could be used as follows:

```
#VRML V2.0 utf8
EXTERNPROTO Gold [] "http://.../materials.wrl#Gold"
...
    Shape {
        appearance Appearance { material Gold {} }
        geometry   ...
    }
```

The advantage is that only one http fetch needs to be done if several things are used from the library; the disadvantage is that the entire library will be transmitted across the network even if only one prototype is used in the file.

4.7 Scripting

4.7.1 Introduction

Decision logic and state management is often needed to decide what effect an event should have on the scene -- "if the vault is currently closed AND the correct combination is entered, then open the vault." These kinds of decisions are expressed as Script nodes (see "*Nodes Reference - Script*") that receive events from other nodes, process them, and send events to other nodes. A Script node can also keep track of information between execution, (i.e. managing internal state over time). This section describes the general mechanisms and semantics that all scripting languages must support. See the specific scripting language appendix for the syntax and details of any language (see "*Appendix C. Java Reference*" and "*Appendix D. JavaScript Reference*").

Event processing is done by a program or script contained in (or referenced by) the Script node's *url* field. This program or script can be written in any programming language that the browser supports. Browsers are not required to implement any specific scripting languages in VRML 2.0.

A Script node is activated when it receives an event. At that point the browser executes the program in the Script node's *url* field (passing the program to an external interpreter if necessary). The program can perform a wide variety of actions: sending out events (and thereby changing the scene), performing calculations, communicating with servers elsewhere on the Internet, and so on. See "*Execution Model*" for a detailed description of the ordering of event processing.

4.7.2 Script Execution

Scripts nodes allow the world author to insert logic into the middle of an event cascade. Scripts also allow the world author to generate an event cascade when a Script node is created or, in some scripting languages, at arbitrary times.

Script nodes receive events in timestamp order. Any events generated as a result of processing an event are given timestamps corresponding to the event that generated them. Conceptually, it takes no time for a Script node to receive and process an event, even though in practice it does take some amount of time to execute a Script.

4.7.3 *Initialize* and *Shutdown*

The scripting language binding may define an *initialize* method (or constructor). This method is called before any events are generated. Events generated by the initialize method must have timestamps less than any other events that are generated by the Script node.

Likewise, the scripting language binding may define a *shutdown* method (or destructor). This method is called when the corresponding Script node is deleted or the world containing the Script node is unloaded or replaced by another world. This can be used as a clean up operation, such as informing external mechanisms to remove temporary files.

4.7.4 *EventsProcessed*

The scripting language binding may also define an *eventsProcessed* routine that is called after one or more events are received. It allows Scripts that do not rely on the order of events received to generate fewer events than an equivalent Script that generates events whenever events are received. If it is used in some other way, eventsProcessed can be non-deterministic, since different implementations may call eventsProcessed at different times.

For a single event cascade, a given Script node's eventsProcessed routine must be called at most once. Events generated from an eventsProcessed routine are given the timestamp of the last event processed.

4.7.5 Scripts with Direct Outputs

Scripts that have access to other nodes (via SFNode or MFNode fields or eventIns) and that have their "directOutputs" field set to TRUE may directly post eventIns to those nodes. They may also read the last value sent from any of the node's eventOuts.

When setting a value in another node, implementations are free to either immediately set the value or to defer setting the value until the Script is finished. When getting a value from another node, the value returned must be up-to-date; that is, it must be the value immediately before the time of the current timestamp (the current timestamp is the timestamp of the event that caused the Script node to execute).

The order of execution of Script nodes that do not have ROUTES between them is undefined. If multiple directOutputs Scripts all read and/or write the same node, the results may be undefined. Just as with ROUTE fan-in, these cases are inherently non-deterministic and it is up to the world creator to ensure that these cases do not happen.

4.7.6 Asynchronous Scripts

Some languages supported by VRML browser may allows Script nodes to spontaneously generate events, allowing users to create Script nodes that function like new Sensor nodes. In these cases, the Script is generating the initial event that cause the event cascade, and the scripting language and/or the browser will determine an appropriate timestamp for that initial event. Such events are then sorted into the event stream and processed like any other event, following all of the same rules for looping, etc.

4.7.7 Script Languages

The Script node's *url* field may specify a URL which refers to a file (e.g. http:) or directly inlines (e.g. javabc:) scripting language code. The mime-type of the returned data defines the language type. Additionally instructions can be included inline using either the data: protocol (which allows a mime-type specification) or a "*Scripting Language Protocol*" defined for the specific language (in which the language type is inferred).

4.7.8 EventIn Handling

Events received by the Script node are passed to the appropriate scripting language function in the script. The function's name depends on the language type used--in some cases it is identical to name of the eventIn, while in others it is a general callback function for all eventIns (see the language appendices for details). The function is passed two arguments, the event value and the event timestamp.

For example, the following Script node has one eventIn field named *start* and three different URL values specified in the *url* field: JavaScript, Java, and inline JavaScript:

```
Script {
    eventIn SFBool start
    url [ "http://foo.com/fooBar.class",
```

```
            "http://foo.com/fooBar.js",
            "javascript:function start(value, timestamp) { ... }"
        ]
    }
```

In the above example when a *start* eventIn is received by the Script node, one of the scripts found in the url field is executed. The Java code is the first choice, the JavaScript code is the second choice, and the inline JavaScript code the third choice - see "*URLs and URNs*" for a description of order of preference for multiple valued URL fields. In the above example, .

4.7.9 Accessing Fields and Events

The fields, eventIns and eventOuts of a Script node are accessible from scripting language functions. The Script's eventIns can be routed to and its eventOuts can be routed from. Another Script node with a pointer to this node can access its eventIns and eventOuts just like any other node.

Accessing Fields and EventOuts of the Script

Fields defined in the Script node are available to the script through a language specific mechanism (e.g. a member variable is automatically defined for each field and event of the Script node). The field values can be read or written and are persistent across function calls. EventOuts defined in the script node can also be read - the value is the last value sent.

Accessing Fields and EventOuts of Other Nodes

The script can access any exposedField, eventIn or eventOut of any node to which it has a pointer. The syntax of this mechanism is language dependent. The following example illustrates how a Script node accesses and modifies an exposed field of another node (i.e. sends a *set_translation* eventIn to the Transform node) using a fictitious scripting language:

```
    DEF SomeNode Transform { }
    Script {
        field   SFNode  tnode USE SomeNode
        eventIn SFVec3f pos
        directOutput TRUE
        url "...
            function pos(value, timestamp) {
                tnode.set_translation = value;
            }"
    }
```

Sending EventOuts

Each scripting language provides a mechanism for allowing scripts to send a value through an eventOut defined by the Script node. For example, one scripting language may define an explicit function for sending each eventOut, while another language may use assignment statements to automatically defined eventOut variables to implicitly send the eventOut. The results of sending multiple values through an eventOut during a single script execution are undefined - it may result in multiple eventOuts with the same timestamp or a single event out with the value of the last assigned value.

4.7.10 Browser Script Interface

The browser interface provides a mechanism for scripts contained by Script nodes to get and set browser state, such as the URL of the current world. This section describes the **semantics** that functions/methods that the browser interface supports. A C-like syntax is used to define the type of parameters and returned values, but is hypothetical. See the specific appendix for a language for the actual syntax required. In this hypothetical syntax, types are given as VRML field types. Mapping of these types into those of the underlying language (as well as any type conversion needed) is described in the appropriate language reference.

SFString getName( );

SFString getVersion( );

The **getName()** and **getVersion()** methods get the "name" and "version" of the browser currently in use. These values are defined by the browser writer, and identify the browser in some (unspecified) way. They are not guaranteed to be unique or to adhere to any particular format, and are for information only. If the information is unavailable these methods return empty strings.

SFFloat getCurrentSpeed( );

The **getCurrentSpeed()** method returns the speed at which the viewpoint is currently moving, in meters per second. If speed of motion is not meaningful in the current navigation type, or if the speed cannot be determined for some other reason, 0.0 is returned.

SFFloat getCurrentFrameRate( );

The **getCurrentFrameRate()** method returns the current frame rate in frames per second. The way in which this is measured and whether or not it is supported at all is browser dependent. If frame rate is not supported, or can't be determined, 0.0 is returned.

SFString getWorldURL( );

The **getWorldURL()** method returns the URL for the root of the currently loaded world.

void replaceWorld(MFNode nodes);

The **replaceWorld()** method replaces the current world with the world represented by the passed nodes. This will usually not return, since the world containing the running script is being replaced.

void loadURL(MFString url, MFString parameter);

The **loadURL** method loads *url* with the passed parameters. *Parameter* is as described in the Anchor node. This method returns immediately but if the URL is loaded into this browser window (e.g. - there is no TARGET parameter to redirect it to another frame) the current world will be terminated and replaced with the data from the new URL at some time in the future.

void setDescription(SFString description);

The **setDescription** method sets the passed string as the current description. This message is displayed in a browser dependent manner. To clear the current description, send an empty string.

MFNode createVrmlFromString(SFString vrmlSyntax);

The **createVrmlFromString()** method takes a string consisting of a VRML scene description, parses the nodes contained therein and returns the root nodes of the corresponding VRML scene.

void createVrmlFromURL(MFString url, SFNode node, SFString event);

The **createVrmlFromURL()** instructs the browser to load a VRML scene description from the given URL or URLs. After the scene is loaded, *event* is sent to the passed *node* returning the root nodes of the corresponding VRML scene. The event parameter contains a string naming an MFNode eventIn on the passed node.

void addRoute(SFNode fromNode, SFString fromEventOut, SFNode toNode,

SFString toEventIn);

void deleteRoute(SFNode fromNode, SFString fromEventOut,

SFNode toNode, SFString toEventIn);

These methods respectively add and delete a route between the given event names for the given nodes.

4.8 Browser Extensions

4.8.1 Creating Extensions

Browsers that wish to add functionality beyond the capabilities in the specification should do so by creating prototypes or external prototypes. If the new node cannot be expressed using the prototyping mechanism (i.e. it cannot be expressed as VRML scene graph), then it should be defined as an external prototype with a unique URN specification. Authors who use the extended functionality may provide multiple, alternative URLs or URNs to represent the content to ensure that it is viewable on all browsers.

For example, suppose a browser wants to create a native Torus geometry node implementation:

```
EXTERNPROTO Torus [ field SFFloat bigR, field SFFloat smallR ]
    ["urn:inet:library:Torus", "http://.../proto_torus.wrl" ]
```

This browser will recognize the URN and use its own private implementation of the Torus node. Other browsers may not recognize the URN, and skip to the next entry in the URL list and search for the specified prototype file. If no URLs or URNs are found, the Torus is assumed to be a an empty node.

Note that the prototype name, "Torus", in the above example has no meaning whatsoever. The URN/URL uniquely and precisely defines the name/location of the node implementation. The prototype name is strictly a convention chosen by the author and shall not be interpreted in any semantic manner. The following example uses both "Ring" and "Donut" to name the torus node, but that the URN/URL, "urn:library:Torus, http://.../proto_torus.wrl", specify the actual definition of the Torus node:

```
#VRML V2.0 utf8

EXTERNPROTO Ring [field SFFloat bigR, field SFFloat smallR ]
    ["urn:library:Torus", "http://.../proto_torus.wrl" ]

EXTERNPROTO Donut [field SFFloat bigR, field SFFloat smallR ]
    ["urn:library:Torus", "http://.../proto_torus.wrl" ]

Transform { ... children Shape { geometry Ring } }
Transform { ... children Shape { geometry Donut } }
```

4.8.2 Reading Extensions

VRML-compliant browsers must recognize and implement the PROTO, EXTERNPROTO, and URN specifications. Note that the prototype names (e.g. Torus) has no semantic meaning whatsoever. Rather, the URL and the URN uniquely determine the location and semantics of the node. Browsers shall not use the PROTO or EXTERNPROTO name to imply anything about the implementation of the node.

4.9 Node Concepts

4.9.1 Bindable Children Nodes

The Background, Fog, NavigationInfo, and Viewpoint nodes have the unique behavior that only one of each type can be active (i.e. affecting the user's experience) at any point in time. See "*Grouping and Children Nodes*" for a description of legal children nodes. The browser shall maintain a stack for each type of binding node. Each of these nodes includes a *set_bind* eventIn and an *isBound* eventOut. The *set_bind* eventIn is used to moves a given node to and from its respective top of stack. A TRUE value sent to *set_bind* eventIn, moves the node to the top of the stack, and a FALSE value removes it from the stack. The *isBound* event is output when a given node is moved to the top of the stack, removed from the stack, or is pushed down in the stack by another node being placed on top. That is, the *isBound* event is sent when a given node ceases to be the active node. The node at the top of stack, (the most recently bound node), is the active node for its type and is used by the browser to set world state. If the stack is empty (i.e. either the file has no binding nodes for a given type or the stack has been popped until empty), then the default field values for that node type are used to set world state. The results are undefined if a multiply instanced (DEF/USE) bindable node is bound.

Bind Stack Behavior

1. During read:
    - the first encountered *<binding node>* is bound by pushing it to the top of the *<binding node>* binding stack:

        - nodes contained within Inlines are not candidates for the first encountered binding node,
        - the first node within a prototype is valid as the a first encountered binding node,
    - the first encountered node sends an *isBound* TRUE event.
2. When a *set_bind* TRUE eventIn is received by a *<binding node>:*
    - if it is not on the top of the stack:
        - the existing top of stack node sends an *isBound* eventOut FALSE,
        - the new node is moved to the top of the stack (i.e. there is only one entry in the stack for any node at any time) and becomes the currently bound *<binding node>,*
        - the new top of stack node sends an *isBound* TRUE eventOut;
    - else if the node is already at the top of the stack, then this event has no affect.
3. When a *set_bind* FALSE eventIn is received by a *<binding node>:*
    - it is removed from the stack,
    - if it is on the top of the stack:
        - it sends an *isBound* eventOut FALSE,
        - the next node in the stack becomes the currently bound *<binding node>* (i.e. pop) and issues an *isBound* TRUE eventOut.
4. If a *set_bind* FALSE eventIn is received by a node not in the stack, the event is ignored and *isBound* events are not sent.
5. When a node replaces another node at the top of the stack, the *isBound* TRUE and FALSE eventOuts from the two nodes are sent simultaneously (i.e. identical timestamps).
6. If a bound node is deleted then it behaves as if it received a *set_bind* FALSE event (see #3).

4.9.2 Geometry

Geometry nodes must be contained by a Shape node in order to be visible to the user. The Shape node contains exactly one geometry node in its *geometry* field. This node must be one of the following node types:

Box	**Cone**	**Cylinder**	**ElevationGrid**	**Extrusion**
IndexedFaceSet	**IndexedLineSet**	**PointSet**	**Sphere**	**Text**

Several geometry nodes also contain Coordinate, Color, Normal, and TextureCoordinate as geometric property nodes. These property nodes are separated out as individual nodes so that instancing and sharing is possible between different geometry nodes. All geometry nodes are specified in a local coordinate system and are affected by parent transformations.

Application of material, texture, and colors:
: See "*Lighting Model*" for details on how material, texture, and color specifications interact.

Shape Hints Fields:
: The ElevationGrid, Extrusion, and IndexedFaceSet nodes each have three SFBool fields that provide hints about the shape--whether it contains ordered vertices, whether the shape is solid, and whether it contains convex faces. These fields are *ccw*, *solid*, and *convex*.

: The *ccw* field indicates whether the vertices are ordered in a counter-clockwise direction when the shape is viewed from the outside (TRUE). If the order is clockwise, this field value is FALSE and the vertices are ordered in a clockwise direction when the shape is viewed from the outside. The *solid* field indicates whether the shape encloses a volume (TRUE), and can be used as a hint to perform backface culling. If nothing is known about the shape, this field value is FALSE (and implies that backface culling cannot be performed and that the polygons are two-sided). If solid is TRUE, the ccw field has no affect. The *convex* field indicates whether all faces in the shape are convex (TRUE). If nothing is known about the faces, this field value is FALSE.

These hints allow VRML implementations to optimize certain rendering features. Optimizations that may be performed include enabling backface 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. If the object is not solid but has ordered vertices, it may turn off backface culling and turn on two-sided lighting.

Crease Angle Field:

The *creaseAngle* field, used by the ElevationGrid, Extrusion, and IndexedFaceSet nodes, affects how default normals are generated. For example, 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 positive angle between surface normals on adjacent polygons. For example, a crease angle of .5 radians 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. Crease angles must be greater than or equal to 0.0.

4.9.3 Interpolators

Interpolators nodes are designed for linear keyframed animation. That is, an interpolator node defines a piecewise linear function, *f(t)*, on the interval (*-infinity, infinity).* The piecewise linear function is defined by *n* values of *t,* called `key`, and the *n* corresponding values of *f(t)*, called `keyValue`. The keys must be monotonic non-decreasing and are not restricted to any interval. An interpolator node evaluates *f(t)* given any value of *t* (via the *set_fraction* eventIn).

Let the *n* keys *k0, k1, k2, ..., k(n-1)* partition the domain (*-infinity, infinity*) into the *n+1* subintervals given by (*-infinity*, *k0), [k0, k1), [k1, k2), ... , [k(n-1), infinity).* Also, let the n values *v0, v1, v2, ..., v(n-1)* be the values of an unknown function, *F(t),* at the associated key values. That is, *vj = F(kj).* The piecewise linear interpolating function, *f(t)*, is defined to be

```
f(t) = v0,      if t < k0,
     = v(n-1), if t > k(n-1),
     = vi,      if t = ki for some value of i, where -1<i<n,
     = linterp(t, vj, v(j+1)), if kj < t < k(j+1),
```

where *linterp(t,x,y)* is the linear interpolant, and *-1< j < n-1*. The third conditional value of *f(t)* allows the defining of multiple values for a single key, i.e. limits from both the left and right at a discontinuity in *f(t)*.The first specified value will be used as the limit of *f(t)* from the left, and the last specified value will be used as the limit of *f(t)* from the right. The value of *f(t)* at a multiply defined key is indeterminate, but should be one of the associated limit values.

There are six different types of interpolator nodes, each based on the type of value that is interpolated:

ColorInterpolator	**CoordinateInterpolator**	**NormalInterpolator**
OrientationInterpolator	**PositionInterpolator**	**ScalarInterpolator**

All interpolator nodes share a common set of fields and semantics:

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

The type of the *keyValue* field is dependent on the type of the interpolator (e.g. the ColorInterpolator's *keyValue* field is of type MFColor). Each value in the *keyValue* field corresponds in order to a parameterized time in the *key* field. Therefore, there exists exactly the same number of values in the *keyValue* field as key values in the *key* field.

The set_fraction eventIn receives a float event and causes the interpolator function to evaluate. The results of the linear interpolation are sent to *value_changed* eventOut.

Four of the six interpolators output a single-valued field to *value_changed*. The exceptions, CoordinateInterpolator and NormalInterpolator, send multiple-value results to *value_changed*. In this case, the *keyValue* field is an *n*x*m* array of values, where *n* is the number of keys and *m* is the number of values per key. It is an error if *m* is not a positive integer value.

The following example illustrates a simple ScalarInterpolator which contains a list of float values (11.0, 99.0, and 33.0), the keyframe times (0.0, 5.0, and 10.0), and outputs a single float value for any given time:

```
ScalarInterpolator {
   key      [ 0.0,  5.0,  10.0]
   value    [11.0, 99.0, 33.0]
}
```

For an input of 2.5 (via `set_fraction`), this `ScalarInterpolator` would send an output value of:

```
eventOut SFFloat value_changed 55.0
                     # = 11.0 + ((99.0-11.0)/(5.0-0.0)) * 2.5
```

Whereas the CoordinateInterpolator below defines an array of coordinates for each keyframe value and sends an array of coordinates as output:

```
CoordinateInterpolator {
   key   [ 0.0,  0.5,  1.0]
   value [ 0  0  0,    10 10 30,   # 2 keyValue(s) at key 0.0
           10 20 10,   40 50 50,   # 2 keyValue(s) at key 0.5
           33 55 66,   44 55 65 ] # 2 keyValue(s) at key 1.0

}
```

In this case, there are two coordinates for every keyframe. The first two coordinates (0, 0, 0) and (10, 10,

30) represent the value at keyframe 0.0, the second two coordinates (10, 20, 10) and (40, 50, 50) represent that value at keyframe 0.5, and so on. If a `set_fraction` value of 0.25 (meaning 25% of the animation) was sent to this CoordinateInterpolator, the resulting output value would be:

```
eventOut MFVec3f value_changed [ 5 10 5,  25 30 40 ]
```

If an interpolator node's *value* eventOut is read (e.g. *get_value*) before it receives any inputs, then *keyValue[0]* is returned.

The location of an interpolator node in the scene graph has no affect on its operation. For example, if a parent of an interpolator node is a Switch node with *whichChoice* set to -1 (i.e. ignore its children), the interpolator continues to operate as specified (receives and sends events).

4.9.4 Light Sources

In general, shape nodes are illuminated by the sum of all of the lights in the world that affect them. This includes the contribution of both the direct and ambient illumination from light sources. Ambient illumination results from the scattering and reflection of light originally emitted directly by light sources. The amount of ambient light is associated with the individual lights in the scene. This is a gross approximation to how ambient reflection actually occurs in nature.

There are three types of light source nodes:

DirectionalLight	**PointLight**	**SpotLight**

All light source node contain an *intensity*, a *color*, and an *ambientIntensity* field. The *intensity* field specifies the brightness of the direct emission from the light, and the *ambientIntensity* specifies the intensity of the ambient emission from the light. Light intensity may range from 0.0, no light emission, to 1.0, full intensity. The color field specifies the spectral color properties of the light emission, as an RGB value in the 0.0 to 1.0 range.

PointLight and SpotLight illuminate all objects in the world that fall within their volume of lighting influence regardless of location within the file. PointLight defines this volume of influence as a sphere centered at the light (defined by a radius). SpotLight defines the volume of influence a solid angle defined by a radius and a cutoff angle. DirectionalLights illuminate only the objects descended from the light's parent grouping node (including any descendant children of the parent group node).

4.9.5 Lighting Model

Lighting 'off'

A Shape node is unlit if any of the following are true:

- The shape's appearance field is NULL (default)
- The material field in the Appearance node is NULL (default)

If the shape is unlit, then the color (I) and alpha (A, 1-transparency) of the shape at each point on the

If the shape is unlit, then the color (I_{rgb}) and alpha (A, 1-transparency) of the shape at each point on the shape’s geometry is given by the following table:

Unlit Geometry	Color per-vertex or per-face	Color NULL
No texture	$I_{rgb} = I_{Crgb}$ $A = 1$	$I_{rgb} = (1, 1, 1)$ $A = 1$
Intensity (one-component) texture	$I_{rgb} = I_T \times I_{Crgb}$ $A = 1$	$I_{rgb} = (I_T, I_T, I_T)$ $A = 1$
Intensity+Alpha (two-component) texture	$I_{rgb} = I_T \times I_{Crgb}$ $A = A_T$	$I_{rgb} = (I_T, I_T, I_T)$ $A = A_T$
RGB (three-component) texture	$I_{rgb} = I_{Trgb}$ $A = 1$	$I_{rgb} = I_{Trgb}$ $A = 1$
RGBA (four-component) texture	$I_{rgb} = I_{Trgb}$ $A = A_T$	$I_{rgb} = I_{Trgb}$ $A = A_T$

where:

A_T = normalized (0-1) alpha value from 2 or 4 component texture image
I_{Crgb} = interpolated per-vertex color, or per-face color, from Color node
I_T = normalized (0-1) intensity from 1-2 component texture image
I_{Trgb}= color from 3-4 component texture image

Lighting ‘on’

If the shape is lit (a Material and an Appearance node are specified for the Shape), then the Color and Texture nodes determine the diffuse color for the lighting equation, as specified in the following table:

Lit Geometry	Color per-vertex or per-face	Color NULL
No texture	$O_{drgb} = I_{Crgb}$ $A = 1 - T_M$	$O_{drgb} = I_{Mrgb}$ $A = 1 - T_M$
Intensity texture (one-component)	$O_{drgb} = I_T \times I_{Crgb}$ $A = 1 - T_M$	$O_{drgb} = I_T \times I_{Mrgb}$ $A = 1 - T_M$
Intensity+Alpha texture (two-component)	$O_{drgb} = I_T \times I_{Crgb}$ $A = A_T$	$O_{drgb} = I_T \times I_{Mrgb}$ $A = A_T$
RGB texture (three-component)	$O_{drgb} = I_{Trgb}$ $A = 1 - T_M$	$O_{drgb} = I_{Trgb}$ $A = 1 - T_M$
RGBA texture (four-component)	$O_{drgb} = I_{Trg}$ $A = A_T$	$O_{drgb} = I_{Trgb}$ $A = A_T$

where:

I_{Mrgb} = material diffuseColor
O_{drgb} = diffuse factor, used in lighting equations below
T_M = material transparency

... and all other terms are as above.

Lighting equations

An ideal VRML 2.0 implementation will evaluate the following lighting equation at each point on a surface. RGB intensities at each point on a geometry (I_{rgb}) are given by:

$I_{rgb} = I_{frgb} \times (1 - s_0) + s_0 \times (O_{ergb} + \mathrm{SUM}(on_i \times attenuation_i \times spot_i \times I_{iprgb} \times (ambient_i + diffuse_i + specular_i)))$

where:

$\cdot$ = modified vector dot product: 0 if dot product < 0, dot product otherwise.
I_{frgb} = currently bound fog's color
I_{iprgb} = light i color
I_{ia} = light i ambientIntensity
$\mathbf{L}$ = (Point/SpotLight) normalized vector from point on geometry to light source i position
$\mathbf{L}$ = (DirectionalLight) -direction of light source i
$\mathbf{N}$ = normalized normal vector at this point on geometry
O_a = material ambientIntensity
O_{drgb} = diffuse color, from material node, Color node, and/or Texture node
O_{ergb} = material emissiveColor
O_{srgb} = material specularColor
$\mathbf{V}$ = normalized vector from point on geometry to viewer's position
$attenuation_i = \max (1/(c_1 + c_2 \times d_L + c_3 \times d_L^2), 1)$
$ambient_i = I_{ia} \times I_{iprgb} \times O_{drgb} \times O_a$
c_1, c_2, c_3 = light i attenuation
d_V = distance from point on geometry to viewer's position, in world space
d_L = distance from light to point on geometry, in light's coordinate system
$diffuse_i = k_d \times O_{drgb} \times (\mathbf{N} \cdot \mathbf{L})$
$k_d = k_s$ = light i intensity
on_i = 1 if light source i affects this point on the geometry
on_i = 0 if light source does not affect this geometry (if farther away than radius for Point or SpotLights, outside of enclosing Group/Transform for a DirectionalLight, or on field is FALSE).
shininess = material shininess

$\text{specular}_i = k_s \times O_{Srgb} \times (\mathbf{N} \cdot ((\mathbf{L}+\mathbf{v}) / |\mathbf{L}+\mathbf{v}|)^{\text{shininess}\times 128}$

$\text{spot}_i = 1$	spotCutoff_i >= pi/2 or light i is PointLight or DirectionalLight
$\text{spot}_i = 0$	spotCutoff_i < pi/2 and $\mathbf{L} \cdot \mathbf{spotDir}_i < \cos(\text{spotCutoff}_i)$
$\text{spot}_i = (\mathbf{L} \cdot \mathbf{spotDir}_i)^{\text{spotExponent}*128}$	spotCutoff < pi/2 and $\mathbf{L} \cdot \mathbf{spotDir}_i > \cos(\text{spotCutoff}_i)$

spotCutoff_i = SpotLight i cutoff angle
$\mathbf{spotDir}_i$ = normalized SpotLight i direction
spotExponent = SpotLight i exponent
SUM: sum over all light sources i

$s_0 = 1$	no fog
$s_0 = (\text{fogVisibility}-d_V) / \text{fogVisibility}$	fogType "LINEAR", d_V < fogVisibility
$s_0 = 0$	fogType "LINEAR", d_V > fogVisibility
$s_0 = \exp(-d_V / (\text{fogVisibility}-d_V))$	fogType "EXPONENTIAL", d_V < fogVisibility
$s_0 = 0$	fogType "EXPONENTIAL", d_V > fogVisibility

References

The VRML lighting equations are based on the simple illumination equations given in "Computer Graphics: Principles and Practice", Foley, van Dam, Feiner and Hughes, section 16.1, "Illumination and Shading", [FOLE], and in the OpenGL 1.1 specification (http://www.sgi.com/Technology/openGL/spec.html) section 2.13 (Lighting) and 3.9 (Fog), [OPEN].

4.9.6 Sensor Nodes

There are several different kinds of sensor nodes: ProximitySensor, TimeSensor, VisibilitySensor, and a variety of *pointing device sensors* (Anchor, CylinderSensor, PlaneSensor, SphereSensor, TouchSensor). Sensors are children nodes in the hierarchy and therefore may be parented by grouping nodes, see "*Grouping and Children Nodes*".

The ProximitySensor detects when the user navigates into a specified invisible region in the world. The TimeSensor is a clock that has no geometry or location associated with it - it is used to start and stop time-based nodes, such as interpolators. The VisibilitySensor detects when a specific part of the world becomes visible to the user. Pointing device sensors detect user pointing events, such as the user activating on a piece of geometry (i.e. TouchSensor). Proximity, time, and visibility sensors are additive. Each one is processed independently of whether others exist or overlap.

Pointing Device Sensors

The following nodes are considered to be pointing device sensors:

Anchor	**CylinderSensor**	**PlaneSensor**	**SphereSensor**	**TouchSensor**

Pointing device sensors are activated when the user points to geometry that is influenced by a specific pointing device sensor. These sensors have influence over all geometry that is descendant from the sensor's parent group. [In the case of the Anchor node, the Anchor itself is considered to be the parent group.] Typically, the pointing device sensor is a sibling to the geometry that it influences. In other cases, the sensor is a sibling to groups which contain geometry (that is influenced by the pointing device sensor).

For a given user activation, the *lowest*, enabled pointing device sensor in the hierarchy is activated - all other pointing device sensors *above* it are ignored. The hierarchy is defined by the geometry node which is activated and the entire hierarchy upward. If there are multiple pointing device sensors tied for lowest, then each of these is activated simultaneously and independently, possibly resulting in multiple sensors activated and outputting simultaneously. This feature allows useful combinations of pointing device sensors (e.g. TouchSensor and PlaneSensor). If a pointing device sensor is instanced (DEF/USE), then the any geometry associated with any of its parents must be tested for intersection and activated hit.

The Anchor node is considered to be a pointing device sensor when trying to determine which sensor (or Anchor) to activate. For example, in the following file a click on *Shape3* is handled by *SensorD*, a click on *Shape2* is handled by *SensorC* and the *AnchorA*, and a click on *Shape1* is handled by *SensorA* and *SensorB*:

```
Group {
    children [
        DEF Shape1  Shape       { ... }
        DEF SensorA TouchSensor { ... }
        DEF SensorB PlaneSensor { ... }
        DEF AnchorA Anchor {
            url "..."
            children [
                DEF Shape2  Shape { ... }
                DEF SensorC TouchSensor { ... }
                Group {
                    children [
                        DEF Shape3  Shape { ... }
                        DEF SensorD TouchSensor { ... }
                    ]
                }

            ]

        }
    ]
}
```

Drag Sensors

Drag sensors are a subset of pointing device sensors. There are three drag sensors (CylinderSensor, PlaneSensor, SphereSensor) in which pointer motions cause events to be generated according to the "virtual shape" of the sensor. For instance the output of the SphereSensor is an SFRotation, *rotation_changed*, which can be connected to a Transform node's *set_rotation* field to rotate an object. The effect is the user grabs an object and spins it about the center point of the SphereSensor.

To simplify the application of these sensors, each node has an *offset* and an *autoOffset* exposed field. Whenever the sensor generates output, (as a response to pointer motion), the output value (e.g. SphereSensor's *rotation_changed*) is added to the *offset*. If *autoOffset* is TRUE (default), this offset is set to the last output value when the pointing device button is released (*isActive* FALSE). This allows subsequent grabbing operations to generate output relative to the last release point. A simple dragger can be constructed by sending the output of the sensor to a Transform whose child is the object being grabbed. For example:

```
Group {
    children [
        DEF S SphereSensor { autoOffset TRUE }
        DEF T Transform {
            children Shape { geometry Box {} }
        }
    ]
    ROUTE S.rotation_changed TO T.set_rotation
}
```

The box will spin when it is grabbed and moved via the pointer.

When the pointing device button is released, *offset* is set to the last output value and an *offset_changed* event is sent out. This behavior can be disabled by setting the *autoOffset* field to FALSE.

4.9.7 Time Dependent Nodes

AudioClip, MovieTexture, and TimeSensor are time dependent nodes that should activate and deactivate themselves at specified times. Each of these nodes contains the exposedFields: *startTime*, *stopTime*, and *loop,* and the eventOut: *isActive*. The exposedField values are used to determine when the container node becomes active or inactive. Also, under certain conditions, these nodes ignore events to some of their exposedFields. A node ignores an eventIn by not accepting the new value and not generating an eventOut_*changed* event. In this section we refer to an abstract ***TimeDep*** node which can be any one of AudioClip, MovieTexture, or TimeSensor.

TimeDep nodes can execute for 0 or more cycles. A cycle is defined by field data within the node. If, at the end of a cycle, the value of *loop* is FALSE, then execution is terminated (see below for events at termination). Conversely, if *loop* is TRUE at the end of a cycle, then a TimeDep node continues execution into the next cycle. A TimeDep node with loop TRUE at the end of every cycle continues cycling forever if *startTime >= stopTime*, or until *stopTime* if *stopTime > startTime*.

A TimeDep node will generate an *isActive* TRUE event when it becomes active and will generate an *isActive* FALSE event when it becomes inactive. These are the only times at which an *isActive* event is generated, i.e., they are not sent at each tick of a simulation.

A TimeDep node is inactive until its *startTime* is reached. When time `now` is equal to *startTime* an *isActive* TRUE event is generated and the TimeDep node becomes active. When a TimeDep node is read from a file, and the ROUTEs specified within the file have been established, the node should determine if it is active and, if so, generate an *isActive* TRUE event and begin generating any other necessary events. However, if a node would have become inactive at any time before the reading of the file, then no events are generated upon the completion of the read.

An active TimeDep node will become inactive at time `now` for `now` = *stopTime > startTime.* The value of *stopTime* is ignored if *stopTime <= startTime*. Also, an active TimeDep node will become inactive at the end of the current cycle if *loop* = FALSE. If an active TimeDep node receives a *set_loop* = FALSE event, then execution continues until the end of the current cycle or until *stopTime* (if *stopTime > startTime*), whichever occurs first. The termination at the end of cycle can be overridden by a subsequent *set_loop* = TRUE event.

set_startTime events to an active TimeDep node are ignored. *set_stopTime* events, where *set_stopTime <= startTime,* to an active TimeDep node are also ignored. A *set_stopTime* event to an active TimeDep node, where *startTime < set_stopTime <=* `now,` result in events being generated as if *stopTime* = `now.` That is, final events, including an *isActive* FALSE, are generated and the node becomes inactive. The *stopTime_changed* event will have the *set_stopTime* value. Other final events are node dependent (c.f., TimeSensor).

A TimeDep node may be re-started while it is active by sending it a *set_stopTime* = `now` event (which will cause the node to become inactive) and a *set_startTime* event (setting it to `now` or any time in the future). Browser authors should note that these events will have the same time stamp and should be processed as *set_stopTime,* then *set_startTime* to produce the correct behavior.

The default values for each of the TimeDep nodes have been specified such that a node with default values became inactive in the past (and, therefore, will generate no events upon reading). A TimeDep node can be made active upon reading by specifying *loop* TRUE. This use of a nonterminating TimeDep node should be used with caution since it incurs continuous overhead on the simulation.

The Virtual Reality Modeling Language

5. Node Reference

Version 2.0, ISO/IEC WD 14772

August 4, 1996

This section provides a detailed definition of the syntax and semantics of each node in the specification.

Grouping nodes

- Anchor
- Billboard
- Collision
- Group
- Transform

Special Groups

- Inline
- LOD
- Switch

Common Nodes

- AudioClip
- DirectionalLight
- PointLight
- Script
- Shape
- Sound
- SpotLight
- WorldInfo

Sensors

- CylinderSensor
- PlaneSensor
- ProximitySensor
- SphereSensor
- TimeSensor
- TouchSensor
- VisibilitySensor

Geometry

- Box
- Cone
- Cylinder
- ElevationGrid
- Extrusion
- IndexedFaceSet
- IndexedLineSet
- PointSet
- Sphere
- Text

Geometric Properties

- Color
- Coordinate
- Normal
- TextureCoordinate

Appearance

- Appearance
- FontStyle
- ImageTexture
- Material
- MovieTexture
- PixelTexture
- TextureTransform

Interpolators

- ColorInterpolator
- CoordinateInterpolator
- NormalInterpolator
- OrientationInterpolator
- PositionInterpolator
- ScalarInterpolator

Bindable Nodes

- Background
- Fog
- NavigationInfo
- Viewpoint

Anchor

```
Anchor {
  eventIn       MFNode   addChildren
  eventIn       MFNode   removeChildren
  exposedField  MFNode   children        []
  exposedField  SFString description     ""
  exposedField  MFString parameter       []
  exposedField  MFString url             []
  field         SFVec3f  bboxCenter      0 0 0
  field         SFVec3f  bboxSize        -1 -1 -1
}
```

The Anchor grouping node causes a URL to be fetched over the network when the viewer activates (e.g. clicks) some geometry contained within the Anchor’s children. If the URL pointed to is a legal VRML world, then that world replaces the world which the Anchor is a part of. If non-VRML data type is fetched, it is up to the browser to determine how to handle that data; typically, it will be passed to an appropriate general viewer.

Exactly how a user activates a child of the Anchor depends on the pointing device and is determined by the VRML browser. Typically, clicking with the pointing device will result in the new scene replacing the current scene. An Anchor with an empty ("") *url* does nothing when its children are chosen. See "*Concepts - Sensors and Pointing Device Sensors*" for a description of how multiple Anchors and pointing device sensors are resolved on activation.

See the "*Concepts - Grouping and Children Nodes*" section for a description of *children*, *addChildren*, and *removeChildren* fields and eventIns.

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

The *parameter* exposed field may be used to supply any additional information to be interpreted by the VRML or HTML browser. Each string should consist of "keyword=value" pairs. For example, some browsers allow the specification of a 'target' for a link, to display a link in another part of the HTML document; the *parameter* field is then:

```
Anchor {
  parameter [ "target=name_of_frame" ]
  ...
}
```

An Anchor may be used to bind the initial Viewpoint in a world by specifying a URL ending with "#ViewpointName", where "ViewpointName" is the name of a viewpoint defined in the file. For example:

```
Anchor {
  url "http://www.school.edu/vrml/someScene.wrl#OverView"
  children  Shape { geometry Box {} }
}
```

specifies an anchor that loads the file "someScene.wrl", and binds the initial user view to the Viewpoint named "OverView" (when the Box is activated). If the named Viewpoint is not found in the file, then ignore it and load the file with the default Viewpoint. If no world is specified, then this means that the Viewpoint specified should be bound (*set_bind* TRUE). For example:

```
Anchor {
  url "#Doorway"
  children Shape { geometry Sphere {} }
}
```

binds viewer to the viewpoint defined by the "Doorway" viewpoint in the current world when the sphere is activated. In this case, if the Viewpoint is not found, then do nothing on activation.

See "*Concepts - URLS and URNs*" for more details on the *url* field.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Anchor's children. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, then the results are undefined. A default *bboxSize* value, (-1 -1 -1), implies that the bounding box is not specified and if needed must be calculated by the browser. See "*Concepts - Bounding Boxes*" for a description of *bboxCenter* and *bboxSize* fields.

Appearance

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

The Appearance node specifies the visual properties of geometry by defining the material and texture nodes. The value for each of the fields in this node can be NULL. However, if the field is non-NULL, it must contain one node of the appropriate type.

The *material* field, if specified, must contain a Material node. If the *material* field is NULL or unspecified, lighting is off (all lights are ignored during rendering of the object that references this Appearance) and the unlit object color is (0, 0, 0) - see "*Concepts - Lighting Model*" for details of the VRML lighting model.

The *texture* field, if specified, must contain one of the various types of texture nodes (ImageTexture, MovieTexture, or PixelTexture). If the texture node is NULL or unspecified, the object that references this Appearance is not textured.

The *textureTransform* field, if specified, must contain a TextureTransform node. If the *texture* field is NULL or unspecified, or if the *textureTransform* is NULL or unspecified, the *textureTransform* field has no effect.

AudioClip

```
AudioClip {
  exposedField   SFString description      ""
  exposedField   SFBool   loop             FALSE
  exposedField   SFFloat  pitch            1.0
  exposedField   SFTime   startTime        0
  exposedField   SFTime   stopTime         0
  exposedField   MFString url              []
  eventOut       SFTime   duration_changed
  eventOut       SFBool   isActive
}
```

An AudioClip node specifies audio data that can be referenced by other nodes that require an audio source.

The *description* field is a textual description of the audio source. A browser is not required to display the *description* field but may choose to do so in addition to or in place of playing the sound.

The *url* field specifies the URL from which the sound is loaded. Browsers shall support at least the *wavefile* format in uncompressed PCM format [WAVE]. It is recommended that browsers also support the MIDI file type 1 sound format [MIDI]. MIDI files are presumed to use the General MIDI patch set. See the section on URLs and URNs in "*Concepts - URLs and URNs*" for details on the *url* field. Results are not defined when the URL references unsupported data types.

The *loop, startTime,* and *stopTime* exposedFields and the *isActive* eventOut, and their affects on the AudioClip node, are discussed in detail in the "*Concepts - Time Dependent Nodes*" section. The "*cycle"* of an AudioClip is the length of time in seconds for one playing of the audio at the specified *pitch*.

The *pitch* field specifies a multiplier for the rate at which sampled sound is played. Only positive values are valid for *pitch* (a value of zero or less will produce undefined results). Changing the *pitch* field affects both the pitch and playback speed of a sound. A *set_pitch* event to an active AudioClip is ignored (and no *pitch_changed* eventOut is generated). If *pitch* is set to 2.0, the sound should be played one octave higher than normal and played twice as fast. For a sampled sound, the *pitch* field alters the sampling rate at which the sound is played. The proper implementation of the *pitch* control for MIDI (or other note sequence sound clip) is to multiply the tempo of the playback by the *pitch* value and adjust the MIDI Coarse Tune and Fine Tune controls to achieve the proper pitch change. The *pitch* field must be > 0.0.

A *duration_changed* event is sent whenever there is a new value for the "normal" duration of the clip. Typically this will only occur when the current *url* in use changes and the sound data has been loaded, indicating that the clip is playing a different sound source. The duration is the length of time in seconds for one cycle of the audio for a *pitch* set to 1.0. Changing the *pitch* field will not trigger a *duration_changed* event. A duration value of -1 implies the sound data has not yet loaded or the value is unavailable for some reason.

The *isActive* eventOut can be used by other nodes to determine if the clip is currently active. If an AudioClip is active, then it should be playing the sound corresponding to the sound time (i.e., in the sound’s local time system with sample 0 at time 0):

```
fmod (now - startTime, duration / pitch).
```

Background

```
Background {
  eventIn       SFBool   set_bind
  exposedField MFFloat  groundAngle   []
  exposedfield MFColor  groundColor   []
  exposedField MFString backUrl       []
  exposedField MFString bottomUrl     []
  exposedField MFString frontUrl      []
  exposedField MFString leftUrl       []
  exposedField MFString rightUrl      []
  exposedField MFString topUrl        []
```

```
  exposedField MFFloat  skyAngle      []
  exposedField MFColor  skyColor      [ 0 0 0 ]
  eventOut     SFBool   isBound
}
```

The Background node is used to specify a color backdrop that simulates ground and sky, as well as a background texture, or *panorama*, that is placed behind all geometry in the scene and in front of the ground and sky. Background nodes are specified in the local coordinate system and are affected by the accumulated rotation of their parents (see below).

Background nodes are bindable nodes (see "*Concepts - Bindable Children Nodes*"). There exists a Background stack, in which the top-most Background on the stack is the currently active Background and thus applied to the view. To move a Background to the top of the stack, a TRUE value is sent to the *set_bind* eventIn. Once active, the Background is then bound to the browsers view. A FALSE value of *set_bind*, removes the Background from the stack and unbinds it from the browser viewer. See "*Concepts - Bindable Children Nodes*" for more details on the the bind stack.

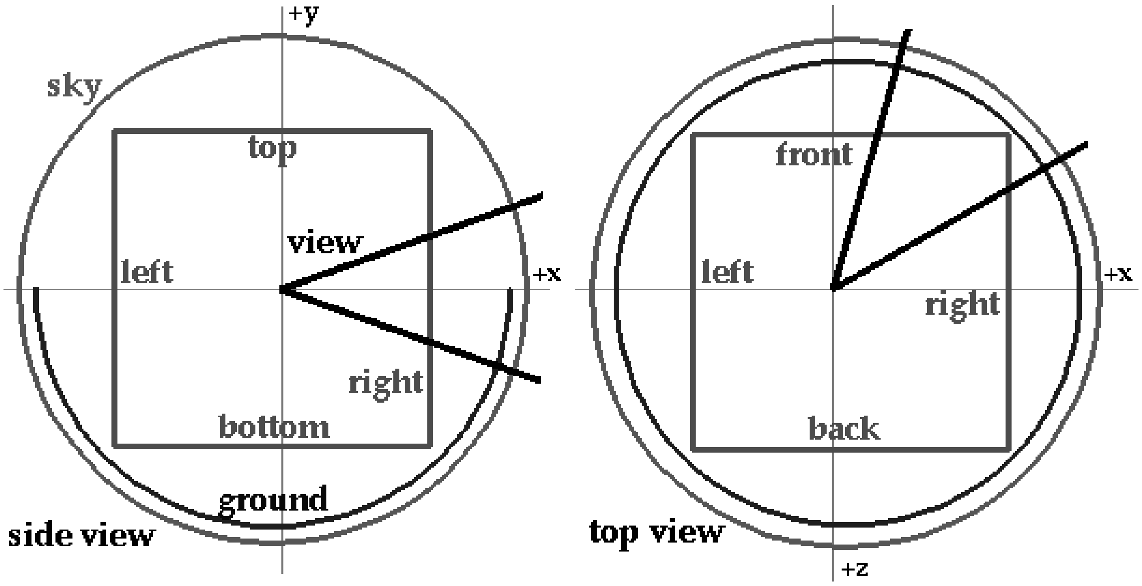
The ground and sky backdrop is conceptually a partial sphere (i.e. ground) enclosed inside of a full sphere (i.e. sky) in the local coordinate system, with the viewer placed at the center of the spheres. Both spheres have infinite radius (epsilon apart), and each is painted with concentric circles of interpolated color perpendicular to the local Y axis of the sphere. The Background node is subject to the accumulated rotations of its parent transformations - scaling and translation transformations are ignored. The sky sphere is always slightly farther away from the viewer than the ground sphere - the ground appears in front of the sky in cases where they overlap.

The *skyColor* field specifies the color of the sky at the various angles on the sky sphere. The first value of the *skyColor* field specifies the color of the sky at 0.0 degrees, the north pole (i.e. straight up from the viewer). The *skyAngle* field specifies the angles from the north pole in which concentric circles of color appear - the north pole of the sphere is implicitly defined to be 0.0 degrees, the natural horizon at *pi*/2 radians, and the south pole is *pi* radians. *skyAngle* is restricted to increasing values in the range 0.0 to *pi*. There must be one more *skyColor* value than there are *skyAngle* values - the first color value is the color at the north pole, which is not specified in the *skyAngle* field. If the last *skyAngle* is less than *pi*, then the color band between the last *skyAngle* and the south pole is clamped to the last *skyColor*. The sky color is linearly interpolated between the specified *skyColor* values.

The *groundColor* field specifies the color of the ground at the various angles on the ground sphere. The first value of the *groundColor* field specifies the color of the ground at 0.0 degrees, the south pole (i.e. straight down). The *groundAngle* field specifies the angles from the south pole that the concentric circles of color appear - the south pole of the sphere is implicitly defined at 0.0 degrees. *groundAngle* is restricted to increasing values in the range 0.0 to *pi*. There must be one more *groundColor* values than there are *groundAngle* values - the first color value is for the south pole which is not specified in the *groundAngle* field. If the last *groundAngle* is less than *pi* (it usually is), then the region between the last *groundAngle* and the north pole is invisible. The ground color is linearly interpolated between the specified *groundColor* values.

The *backUrl*, *bottomUrl*, *frontUrl*, *leftUrl*, *rightUrl*, and *topUrl* fields specify a set of images that define a background panorama, between the ground/sky backdrop and the world's geometry. The panorama consists of six images, each of which is mapped onto the faces of an infinitely large cube centered in the local coordinate system. The images are applied individually to each face of the cube; the entire image goes on each face. On the front, back, right, and left faces of the cube, when viewed from the inside with the Y-axis up, the texture is mapped onto each face with the same orientation as the if image was displayed normally in 2D. On the top face of the cube, when viewed from the inside looking up along the +Y axis with the +Z axis as the view up direction, the texture is mapped onto the face with the same orientation as the if image was displayed normally in 2D. On the bottom face of the box, when viewed from the inside down the -Y axis with the -Z axis as the view up direction, the texture is mapped onto the face with the same orientation as the if image was displayed normally in 2D.

Alpha values in the panorama images (i.e. two or four component images) specify that the panorama is semi-transparent or transparent in regions, allowing the *groundColor* and *skyColor* to be visible. One component images are displayed in greyscale; two component images are displayed in greyscale with alpha transparency; three component images are displayed in full RGB color; four component images are displayed in full RGB color with alpha transparency. Often, the *bottomUrl* and *topUrl* images will not be specified, to allow sky and ground to show. The other four images may depict surrounding mountains or other distant scenery. Browsers are required to support the JPEG [JPEG] and PNG [PNG] image file formats, and in addition, may support any other image formats. Support for the GIF [GIF] format (including transparent backgrounds) is recommended. See the section "*Concepts - URLS and URNs*" for details on the *url* fields.



Panorama images may be one component (greyscale), two component (greyscale plus alpha), three component (full RGB color), or four-component (full RGB color plus alpha).

Ground colors, sky colors, and panoramic images do not translate with respect to the viewer, though they do rotate with respect to the viewer. That is, the viewer can never get any closer to the background, but can turn to examine all sides of the panorama cube, and can look up and down to see the concentric rings of ground and sky (if visible).

Background is not affected by Fog. Therefore, if a Background is active (i.e bound) while a Fog is active, then the Background will be displayed with no fogging effects. It is the author’s responsibility to set the Background values to match the Fog (e.g. ground colors fade to fog color with distance and panorama images tinted with fog color).

The first Background node found during reading of the world is automatically bound (receives *set_bind* TRUE) and is used as the initial background when the world is loaded.

Billboard

```
Billboard {
  eventIn      MFNode   addChildren
  eventIn      MFNode   removeChildren
  exposedField SFVec3f  axisOfRotation  0 1 0
  exposedField MFNode   children        []
  field        SFVec3f  bboxCenter      0 0 0
  field        SFVec3f  bboxSize        -1 -1 -1
}
```

The Billboard node is a grouping node which modifies its coordinate system so that the billboard node's local Z-axis turns to point at the viewer. The Billboard node has children which may be other grouping or leaf nodes.

The *axisOfRotation* field specifies which axis to use to perform the rotation. This axis is defined in the local coordinates of the Billboard node. The default (0,1,0) is useful for objects such as images of trees and lamps positioned on a ground plane. But when an object is oriented at an angle, for example, on the incline of a mountain, then the *axisOfRotation* may also need to be oriented at a similar angle.

A special case of billboarding is *screen-alignment* -- the object rotates to always stay aligned with the viewer even when the viewer elevates, pitches and rolls. This special case is distinguished by setting the *axisOfRotation* to (0, 0, 0).

To rotate the Billboard to face the viewer, determine the line between the Billboard's origin and the viewer's position; call this the *billboard-to-viewer* line. The *axisOfRotation* and the billboard-to-viewer line define a plane. The local z-axis of the Billboard is then rotated into that plane, pivoting around the *axisOfRotation*.

If the *axisOfRotation* and the billboard-to-viewer line are coincident (the same line), then the plane cannot be established, and the rotation results of the Billboard are undefined. For example, if the *axisOfRotation* is set to (0,1,0) (Y-axis) and the viewer flies over the Billboard and peers directly down the Y-axis the results are undefined**.**

Multiple instances of Billboards (DEF/USE) operate as expected - each instance rotates in its unique coordinate system to face the viewer.

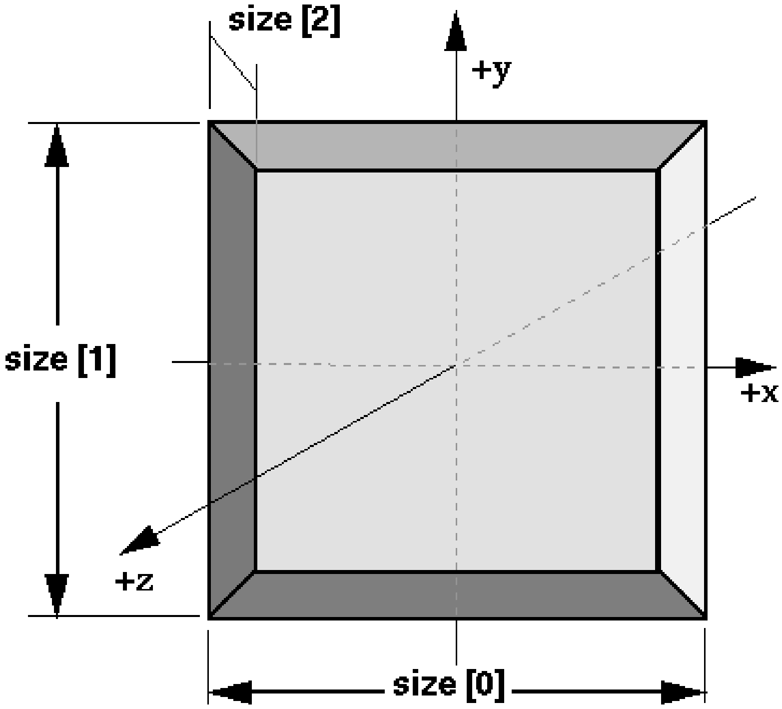
See the "*Concepts - Grouping and Children Nodes*" section for a description the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Billboard's children. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, then the results are undefined. A default *bboxSize* value, (-1 -1 -1), implies that the bounding box is not specified and if needed must be calculated by the browser. See "*Concepts - Bounding Boxes*" for a description of *bboxCenter* and *bboxSize* fields.

Box

```
Box {
  field    SFVec3f size  2 2 2
}
```

The Box node specifies a rectangular parallelepiped box in the local coordinate system centered at (0,0,0) in the local coordinate system and aligned with the coordinate axes. By default, the box measures 2 units in each dimension, from -1 to +1. The Box's *size* field specifies the extents of the the box along the X, Y, and Z axes respectively and must be greater than 0.0.



Textures are applied individually to each face of the box; the entire untransformed texture goes on each face. On the front, back, right, and left faces of the box, when viewed from the outside with the Y-axis up, the texture is mapped onto each face with the same orientation as the if image was displayed in normally 2D. On the top face of the box, when viewed from the outside along the +Y axis looking down with the -Z axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. On the bottom face of the box, when viewed from the outside along the -Y axis looking up with the +Z axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. TextureTransform affects the texture coordinates of the Box.

The Box geometry is considered to be solid and thus requires outside faces only. When viewed from the inside the results are undefined.

Collision

```
Collision {
  eventIn      MFNode   addChildren
  eventIn      MFNode   removeChildren
  exposedField MFNode   children        []
  exposedField SFBool   collide         TRUE
  field        SFVec3f  bboxCenter      0 0 0
  field        SFVec3f  bboxSize        -1 -1 -1
  field        SFNode   proxy           NULL
  eventOut     SFTime   collideTime
}
```

By default, **all objects in the scene are collidable**. Browser shall detect geometric collisions between the user's avatar (see NavigationInfo) and the scene's geometry, and prevent the avatar from 'entering' the geometry. The Collision node is grouping node that may turn off collision detection for its descendants, specify alternative objects to use for collision detection, and send events signaling that a collision has occurred between the user's avatar and the Collision group's geometry or alternate. If there are no Collision nodes specified in a scene, browsers shall detect collision with all objects during navigation.

See the "*Concepts - Grouping and Children Nodes*" section for a description the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The Collision node's *collide* field enables and disables collision detection. If *collide* is set to FALSE, the children and all descendants of the Collision node will not be checked for collision, even though they are drawn. This includes any descendant Collision nodes that have *collide* set to TRUE - (i.e. setting *collide* to FALSE turns it off for every node below it).

Collision nodes with the *collide* field set to TRUE detect the nearest collision with their descendant geometry (or proxies). Note that not all geometry is collidable - see each geometry node's sections for details. When the nearest collision is detected, the collided Collision node sends the time of the collision through its *collideTime* eventOut. This behavior is recursive - if a Collision node contains a child, descendant, or proxy (see below) that is a Collision node, and both Collisions detect that a collision has occurred, then both send a *collideTime* event out at the same time, and so on.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Collision's children. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, then the results are undefined. A default *bboxSize* value, (-1 -1 -1), implies that the bounding box is not specified and if needed must be calculated by the browser. See "*Concepts - Bounding Boxes*" for a description of the *bboxCenter* and *bboxSize* fields.

The collision proxy, defined in the *proxy* field, is a legal child node, (see "*Concepts - Grouping and Children Nodes*"), that is used as a substitute for the Collision's children during collision detection. The proxy is used strictly for collision detection - it is not drawn.

If the value of the *collide* field is FALSE, then collision detection is not performed with the children or proxy descendant nodes. If the root node of a scene is a Collision node with the *collide* field set to FALSE, then collision detection is disabled for the entire scene, regardless of whether descendent Collision nodes have set *collide* TRUE.

If the value of the *collide* field is TRUE and the *proxy* field is non-NULL, then the *proxy* field defines the scene which collision detection is performed. If the *proxy* value is NULL, the *children* of the collision node are collided against.

If *proxy* is specified, then any descendant children of the Collision node are ignored during collision detection. If *children* is empty, *collide* is TRUE and *proxy* is specified, then collision detection is done against the proxy but nothing is displayed (i.e. invisible collision objects).

The *collideTime* eventOut generates an event specifying the time when the user’s avatar (see NavigationInfo) intersects the collidable children or proxy of the Collision node. An ideal implementation computes the exact time of intersection. Implementations may approximate the ideal by sampling the positions of collidable objects and the user. Refer to the NavigationInfo node for parameters that control the user’s size.

Browsers are responsible for defining the navigation behavior when collisions occur. For example, when the user comes sufficiently close to an object to trigger a collision, the browser may have the user bounce off the object, come to a stop, or glide along the surface.

■Color

```
Color {
  exposedField MFColor color  []
}
```

This node defines a set of RGB colors to be used in the fields of another node.

Color nodes are only used to specify multiple colors for a single piece of geometry, such as a different color for each face or vertex of an IndexedFaceSet. A Material node is used to specify the overall material parameters of a lighted geometry. If both a Material and a Color node are specified for a geometry, the colors should ideally replace the diffuse component of the material.

Textures take precedence over colors; specifying both a Texture and a Color node for a geometry will result in the Color node being ignored. See "*Concepts - Lighting Model*" for details on lighting equations.

■ ColorInterpolator

```
ColorInterpolator {
  eventIn      SFFloat set_fraction
  exposedField MFFloat key           []
```

```
  exposedField MFColor keyValue      []
  eventOut     SFColor value_changed
}
```

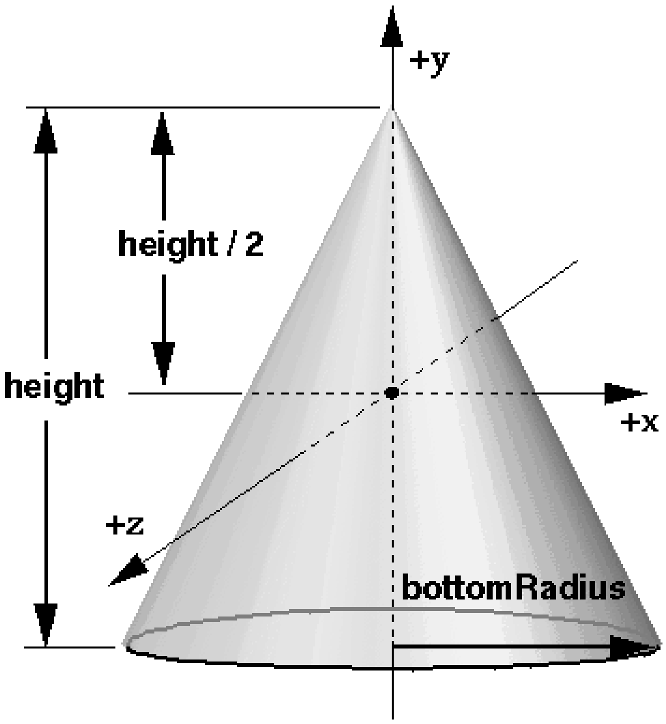
This node interpolates among a set of MFColor key values, to produce an SFColor (RGB) *value_changed* event. The number of colors in the *keyValue* field must be equal to the number of keyframes in the *key* field. The *keyValue* field and *value_changed* events are defined in RGB color space. A linear interpolation, using the value of *set_fraction* as input, is performed in HSV space.

Refer to "*Concepts - Interpolators*" for a more detailed discussion of interpolators.

Cone

```
Cone {
  field     SFFloat   bottomRadius 1
  field     SFFloat   height       2
  field     SFBool    side         TRUE
  field     SFBool    bottom       TRUE
}
```

The Cone node specifies a cone which is centered in the local coordinate system and whose central axis is aligned with the local Y-axis. The *bottonRadius* field specifies the radius of the cone's base, and the *height* field specifies the height of the cone from the center of the base to the apex. By default, the cone has a radius of 1.0 at the bottom and a height of 2.0, with its apex at y=1 and its bottom at y=-1. Both *bottomRadius* and *height* must be greater than 0.0.



The *side* field specifies whether sides of the cone are created, and the *bottom* field specifies whether the bottom cap of the cone is created. A value of TRUE specifies that this part of the cone exists, while a value of FALSE specifies that this part does not exist (not rendered). Parts with field values of FALSE are not collided with during collision detection.

When a texture is applied to the sides of the cone, the texture wraps counterclockwise (from above) starting at the back of the cone. The texture has a vertical seam at the back in the YZ plane, from the apex (0, *height*/2, 0) to the point (0, 0, -r). For the bottom cap, a circle is cut out of the unit texture square centered at (0, -*height*/2, 0) with dimensions (2 * *bottomRadius)* by (2 * *bottomRadius)*. The bottom cap texture appears right side up when the top of the cone is rotated towards the -Z axis. TextureTransform affects the texture coordinates of the Cone.

The Cone geometry is considered to be solid and thus requires outside faces only. When viewed from the inside the results are undefined.

Coordinate

```
Coordinate {
  exposedField MFVec3f point  []
}
```

This node defines a set of 3D coordinates to be used in the *coord* field of vertex-based geometry nodes (such as IndexedFaceSet, IndexedLineSet, and PointSet).

CoordinateInterpolator

```
CoordinateInterpolator {
  eventIn      SFFloat set_fraction
  exposedField MFFloat key           []
  exposedField MFVec3f keyValue      []
  eventOut     MFVec3f value_changed
}
```

This node linearly interpolates among a set of MFVec3f value. This would be appropriate for interpolating Coordinate positions for a geometric morph.

The number of coordinates in the *keyValue* field must be an integer multiple of the number of keyframes in the *key* field; that integer multiple defines how many coordinates will be contained in the *value_changed* events.

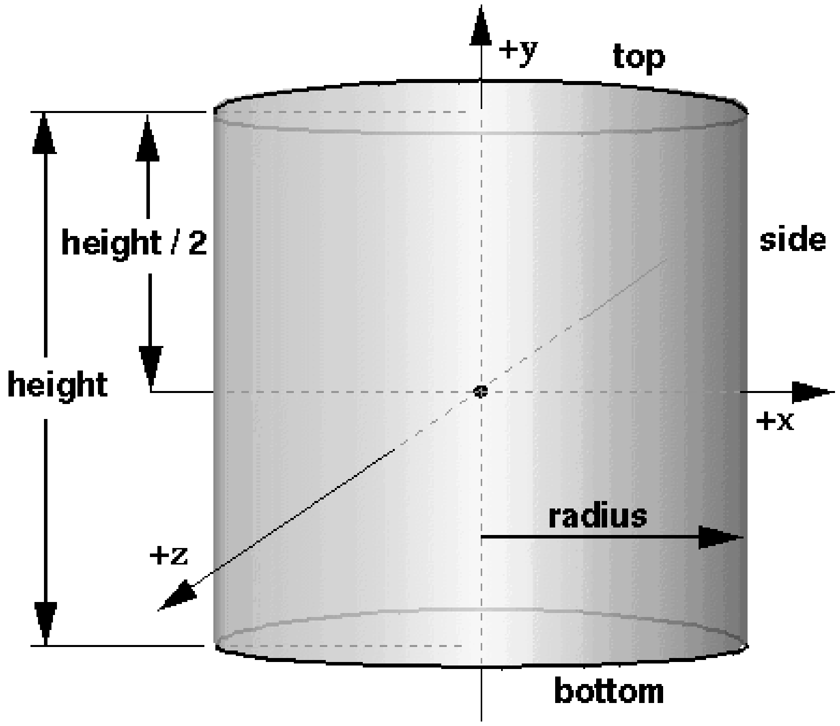
Refer to "*Concepts - Interpolators*" for a more detailed discussion of interpolators.

■Cylinder

```
Cylinder {
  field    SFBool     bottom  TRUE
  field    SFFloat    height  2
  field    SFFloat    radius  1
  field    SFBool     side    TRUE
  field    SFBool     top     TRUE
}
```

The Cylinder node specifies a capped cylinder centered at (0,0,0) in the local coordinate system and with a central axis oriented along the local Y-axis. By default, the cylinder is sized at -1 to +1 in all three dimensions. The *radius* field specifies the cylinder's radius and the *height* field specifies the cylinder's height along the central axis. Both *radius* and *height* must be greater than 0.0.

The cylinder has three *parts*: the *side*, the *top* (Y = +height) and the *bottom* (Y = -height). Each part has an associated SFBool field that indicates whether the part exists (TRUE) or does not exist (FALSE). If the parts do not exist, the they are not considered during collision detection.



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 caps, a circle is cut out of the unit texture square centered at (0, +/- *height*, 0) with dimensions 2**radius* by 2**radius*. 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. TextureTransform affects the texture coordinates of the Cylinder.

The Cylinder geometry is considered to be solid and thus requires outside faces only. When viewed from the inside the results are undefined.

CylinderSensor

```
CylinderSensor {
  exposedField SFBool     autoOffset TRUE
  exposedField SFFloat    diskAngle  0.262
  exposedField SFBool     enabled    TRUE
  exposedField SFFloat    maxAngle   -1
  exposedField SFFloat    minAngle   0
  exposedField SFFloat    offset     0
  eventOut     SFBool     isActive
  eventOut     SFRotation rotation_changed
  eventOut     SFVec3f    trackPoint_changed
}
```

The CylinderSensor maps pointing device (e.g. mouse or wand) motion into a rotation on an invisible cylinder that is aligned with the Y axis of its local space.

The *enabled* exposed field enables and disables the CylinderSensor - if TRUE, the sensor reacts appropriately to user events, if FALSE, the sensor does not track user input or send output events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event the sensor is enabled and ready for user activation.

The CylinderSensor generates events if the pointing device is activated while over any descendant geometry nodes of its parent group and then moved while activated. Typically, the pointing device is a 2D device such as a mouse. The pointing device is considered to be moving within a plane at a fixed distance from the viewer and perpendicular to the line of sight; this establishes a set of 3D coordinates for the pointer. If a 3D pointer is in use, then the sensor generates events only when the pointer is within the user’s field of view. In either case, the pointing device is considered to "pass over" geometry when that geometry is intersected by a line extending from the viewer and passing through the pointer’s 3D coordinates. If multiple sensors’ geometry intersect this line (hereafter called the bearing), only the nearest will be eligible to generate events.

Upon activation of the pointing device (e.g. mouse button down) over the sensor’s geometry, an *isActive* TRUE event is sent. The angle between the bearing vector and the local Y axis of the CylinderSensor determines whether the sides of the invisible cylinder or the caps (disks) are used for manipulation. If the angle is less than the *diskAngle*, then the geometry is treated as an infinitely large disk and dragging motion is mapped into a rotation around the local Y axis of the sensor’s coordinate system. The feel of the rotation is as if you were rotating a dial or crank. Using the right-hand rule, the X axis of the sensor’s local coordinate system, (defined by parents), represents the zero rotation value around the sensor’s local Y axis. For each subsequent position of the bearing, a *rotation_changed* event is output which corresponds to the angle between the local X axis and the vector defined by the intersection point and the nearest point on the local Y axis, plus the *offset* value. *trackPoint_changed* events reflect the unclamped drag position on the surface of this disk. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last rotation angle and an *offset_changed* event is generated. See "*Concepts - Drag Sensors*" for more details on *autoOffset* and *offset_changed*.

If angle between the bearing vector and the local Y axis of the CylinderSensor is greater than or equal to *diskAngle*, then the sensor behaves like a cylinder or rolling pin. The shortest distance between the point of intersection (between the bearing and the sensor's geometry) and the Y axis of the parent group's local coordinate system determines the radius of an invisible cylinder used to map pointing device motion, and mark the zero rotation value. For each subsequent position of the bearing, a *rotation_changed* event is output which corresponds to a relative rotation from the original intersection, plus the *offset* value. *trackPoint_changed* events reflect the unclamped drag position on the surface of this cylinder. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last rotation angle and an *offset_changed* event is generated. See "*Concepts - Drag Sensors*" for more details.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it releases and generates an *isActive* FALSE event (other pointing device sensors cannot generate events during this time). Motion of the pointing device while *isActive* is TRUE is referred to as a "drag". If a 2D pointing device is in use, *isActive* events will typically reflect the state of the primary button associated with the device (i.e. *isActive* is TRUE when the primary button is pressed, and FALSE when not released). If a 3D pointing device (e.g. wand) is in use, *isActive* events will typically reflect whether the pointer is within or in contact with the sensor's geometry.

While the pointing device is activated, *trackPoint_changed* and *rotation_changed* events are output and are interpreted from pointing device motion based on the sensor's local coordinate system at the time of activation. *trackPoint_changed* events represent the unclamped intersection points on the surface of the invisible cylinder or disk. If the initial angle results in cylinder rotation (as opposed to disk behavior) and if the pointing device is dragged off the cylinder while activated, browsers may interpret this in several ways (e.g. clamp all values to the cylinder, continue to rotate as the point is dragged away from the cylinder, etc.). Each movement of the pointing device, while *isActive* is TRUE, generates *trackPoint_changed* and *rotation_changed* events.

minAngle and *maxAngle* may be set to clamp *rotation_changed* events to a range of values (measured in radians about the local Z and Y axis as appropriate). If *minAngle* is greater than *maxAngle*, *rotation_changed* events are not clamped.

See "*Concepts - Pointing Device Sensors and Drag Sensors*" for more details.

DirectionalLight

```
DirectionalLight {
  exposedField SFFloat ambientIntensity  0
  exposedField SFColor color             1 1 1
  exposedField SFVec3f direction         0 0 -1
  exposedField SFFloat intensity         1
  exposedField SFBool  on                TRUE
}
```

The DirectionalLight node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector. See "*Concepts - Lights*" for a definition of the *ambientIntensity*, *color*, *intensity*, and *on* fields.

The *direction* field specifies the direction vector within the local coordinate system that the light illuminates in. Light is emitted along parallel rays from an infinite distance away. A directional light source illuminates only the objects in its enclosing parent group. The light may illuminate everything within this coordinate system, including all children and descendants of its parent group. The accumulated transformations of the parent nodes affect the light.

See "*Concepts - Lighting Model*"for a precise description of VRML's lighting equations.

Some low-end renderers do not support the concept of per-object lighting. This means that placing DirectionalLights inside local coordinate systems, which implies lighting only the objects beneath the Transform with that light, is not supported in all systems. For the broadest compatibility, lights should be placed at outermost scope.

ElevationGrid

```
ElevationGrid {
  eventIn      MFFloat  set_height
  exposedField SFNode   color            NULL
  exposedField SFNode   normal           NULL
  exposedField SFNode   texCoord         NULL
  field        MFFloat  height           []
  field        SFBool   ccw              TRUE
  field        SFBool   colorPerVertex   TRUE
  field        SFFloat  creaseAngle      0
  field        SFBool   normalPerVertex  TRUE
  field        SFBool   solid            TRUE
  field        SFInt32  xDimension       0
  field        SFFloat  xSpacing         0.0
  field        SFInt32  zDimension       0
  field        SFFloat  zSpacing         0.0
}
```

The ElevationGrid node specifies a uniform rectangular grid of varying height in the XZ plane of the local coordinate system. The geometry is described by a scalar array of height values that specify the height of a rectangular surface above each point of the grid.

The *xDimension* and *zDimension* fields indicate the number of dimensions of the grid *height* array in the X and Z directions. Both *xDimension* and *zDimension* must be > 1. The vertex locations for the rectangles are defined by the *height* field and the *xSpacing* and *zSpacing* fields:

- The *height* field is an *xDimension* by *zDimension* array of scalar values representing the height above the grid for each vertex the height values are stored in row major order.
- The *xSpacing* and *zSpacing* fields indicates the distance between vertices in the X and Z directions respectively, and must be >= 0.

Thus, the vertex corresponding to the point, P[*i,* j], on the grid is placed at:

```
P[i,j].x = xSpacing * i
P[i,j].y = height[ i + j * zDimension]
P[i,j].z = zSpacing * j

where 0<i<xDimension and 0<j<zDimension.
```

The *set_height* eventIn allows the height MFFloat field to be changed to allow animated ElevationGrids.

The default texture coordinates range from [0,0] at the first vertex to [1,1] at the last vertex. The S texture coordinate will be aligned with X, and the T texture coordinate with Z.

The *colorPerVertex* field determines whether colors (if specified in the color field) should be applied to each vertex or each quadrilateral of the ElevationGrid. If *colorPerVertex* is FALSE and the *color* field is not NULL, then the *color* field must contain a Color node containing at least (*xDimension-1)**(*zDimension-1)* colors. If *colorPerVertex* is TRUE and the *color* field is not NULL, then the *color* field must contain a Color node containing at least *xDimension***zDimension* colors.

See the "*Concepts - Geometry*" for a description of the *ccw*, *solid*, and *creaseAngle* fields.

By default, the rectangles are defined with a counterclockwise ordering, so the Y component of the normal is positive. Setting the *ccw* field to FALSE reverses the normal direction. Backface culling is enabled when the *ccw* field and the *solid* field are both TRUE (the default).

Extrusion

```
Extrusion {
  eventIn MFVec2f    set_crossSection
  eventIn MFRotation set_orientation
  eventIn MFVec2f    set_scale
  eventIn MFVec3f    set_spine
  field   SFBool     beginCap         TRUE
  field   SFBool     ccw              TRUE
  field   SFBool     convex           TRUE
  field   SFFloat    creaseAngle      0
  field   MFVec2f    crossSection     [ 1 1, 1 -1, -1 -1, -1 1, 1 1 ]
  field   SFBool     endCap           TRUE
  field   MFRotation orientation      0 0 1 0
  field   MFVec2f    scale            1 1
  field   SFBool     solid            TRUE
  field   MFVec3f    spine            [ 0 0 0, 0 1 0 ]
}
```

The Extrusion node specifies geometric shapes based on a two dimensional cross section extruded along a three dimensional spine. The cross section can be scaled and rotated at each spine point to produce a wide variety of shapes.

An Extrusion is defined by a 2D *crossSection* piecewise linear curve (described as a series of connected vertices), a 3D *spine* piecewise linear curve (also described as a series of connected vertices), a list of 2D *scale* parameters, and a list of 3D *orientation* parameters. Shapes are constructed as follows: The cross-section curve, which starts as a curve in the XZ plane, is first scaled about the origin by the first *scale* parameter (first value scales in X, second value scales in Z). It is then rotated about the origin by the first *orientation* parameter, and translated by the vector given as the first vertex of the *spine* curve. It is then extruded through space along the first segment of the *spine* curve. Next, it is scaled and rotated by the second *scale* and *orientation* parameters and extruded by the second segment of the *spine*, and so on. The number of *scale* and *orientation* values shall equal the number of spine points, or contain one value that is applied to all points. The *scale* values must be > 0.

A transformed cross section is found for each joint (that is, at each vertex of the *spine* curve, where segments of the extrusion connect), and the joints and segments are connected to form the surface. No check is made for self-penetration. Each transformed cross section is determined as follows:

1. Start with the cross section as specified, in the XZ plane.
2. Scale it about (0, 0, 0) by the value for *scale* given for the current joint.
3. Apply a rotation so that when the cross section is placed at its proper location on the spine it will be oriented properly. Essentially, this means that the cross section's Y axis (*up* vector coming out of the cross section) is rotated to align with an approximate tangent to the spine curve.

   For all points other than the first or last: The tangent for *spine*[*i*] is found by normalizing the vector defined by (*spine*[*i*+1] - *spine*[*i*-1]).

   If the spine curve is closed: The first and last points need to have the same tangent. This tangent is found as above, but using the points *spine*[0] for *spine*[*i*], *spine*[1] for *spine*[*i*+1] and *spine*[*n*-2] for *spine*[*i*-1], where *spine*[*n*-2] is the next to last point on the curve. The last point in the curve, *spine*[*n*-1], is the same as the first, *spine*[0].

   If the spine curve is not closed: The tangent used for the first point is just the direction from *spine*[0] to *spine*[1], and the tangent used for the last is the direction from *spine*[*n*-2] to *spine*[*n*-1].

   In the simple case where the spine curve is flat in the XY plane, these rotations are all just rotations about the Z axis. In the more general case where the spine curve is any 3D curve, you need to find the destinations for all 3 of the local X, Y, and Z axes so you can completely specify the rotation. The Z axis is found by taking the cross product of:

   (*spine*[*i*-1] - *spine*[*i*]) and (*spine*[*i*+1] - *spine*[*i*]).

   If the three points are collinear then this value is zero, so take the value from the previous point. Once you have the Z axis (from the cross product) and the Y axis (from the approximate tangent), calculate the X axis as the cross product of the Y and Z axes.

4. Given the plane computed in step 3, apply the *orientation* to the cross-section relative to this new plane. Rotate it counter-clockwise about the axis and by the angle specified in the *orientation* field at that joint.
5. Finally, the cross section is translated to the location of the *spine* point.

Surfaces of revolution: If the cross section is an approximation of a circle and the spine is straight, then the Extrusion is equivalent to a surface of revolution, where the *scale* parameters define the size of the cross section along the spine.

Cookie-cutter extrusions: If the scale is 1, 1 and the spine is straight, then the cross section acts like a cookie cutter, with the thickness of the cookie equal to the length of the spine.

Bend/twist/taper objects: These shapes are the result of using all fields. The spine curve bends the extruded shape defined by the cross section, the orientation parameters twist it around the spine, and the scale parameters taper it (by scaling about the spine).

Extrusion has three *parts*: the *sides*, the *beginCap* (the surface at the initial end of the spine) and the *endCap* (the surface at the final end of the spine). The caps have an associated SFBool field that indicates whether it exists (TRUE) or doesn't exist (FALSE).

When the *beginCap* or *endCap* fields are specified as TRUE, planar cap surfaces will be generated regardless of whether the *crossSection* is a closed curve. (If *crossSection* isn't a closed curve, the caps are generated as if it were -- equivalent to adding a final point to *crossSection* that's equal to the initial point. Note that an open surface can still have a cap, resulting (for a simple case) in a shape something like a soda can sliced in half vertically.) These surfaces are generated even if *spine* is also a closed curve. If a field value is FALSE, the corresponding cap is not generated.

Extrusion automatically generates its own normals. Orientation of the normals is determined by the vertex ordering of the triangles generated by Extrusion. The vertex ordering is in turn determined by the *crossSection* curve. If the *crossSection* is counterclockwise when viewed from the +Y axis, then the polygons will have counterclockwise ordering when viewed from 'outside' of the shape (and *vice versa* for clockwise ordered *crossSection* curves).

Texture coordinates are automatically generated by extrusions. Textures are mapped so that the coordinates range in the U direction from 0 to 1 along the *crossSection* curve (with 0 corresponding to the first point in *crossSection* and 1 to the last) and in the V direction from 0 to 1 along the *spine* curve (again with 0 corresponding to the first listed *spine* point and 1 to the last). When *crossSection* is closed, the texture has a seam that follows the line traced by the *crossSection*'s start/end point as it travels along the *spine*. If the *endCap* and/or *beginCap* exist, the *crossSection* curve is uniformly scaled and translated so that the largest dimension of the cross-section (X or Z) produces texture coordinates that range from 0.0 to 1.0. The *beginCap* and *endCap* textures' S and T directions correspond to the X and Z directions in which the *crossSection* coordinates are defined.

See "*Concepts - Geometry Nodes*" for a description of the *ccw*, *solid*, *convex*, and *creaseAngle* fields.

Fog

```
Fog {
  exposedField SFColor  color            1 1 1
  exposedField SFString fogType          "LINEAR"
  exposedField SFFloat  visibilityRange  0
  eventIn      SFBool   set_bind
  eventOut     SFBool   isBound
}
```

The Fog node provides a way to simulate atmospheric effects by blending objects with the color specified by the *color* field based on the objects' distances from the viewer. The distances are calculated in the coordinate space of the Fog node. The *visibilityRange* specifies the distance (in the Fog node's coordinate space) at which objects are totally obscured by the fog. Objects located *visibilityRange* meters or more away from the viewer are drawn with a constant color of *color*. Objects very close to the viewer are blended very little with the fog *color*. A *visibilityRange* of 0.0 or less disables the Fog node. Note that *visibilityRange* is affected by the scaling transformations of the Fog node's parents - translations and rotations have no affect on *visibilityRange*.

Fog nodes are "*Concepts - Bindable Children Nodes*" and thus there exists a Fog stack, in which the top-most Fog node on the stack is currently active. To push a Fog node onto the top of the stack, a TRUE value is sent to the *set_bind* eventIn. Once active, the Fog is then bound to the browsers view. A FALSE value of *set_bind*, pops the Fog from the stack and unbinds it from the browser viewer. See "*Concepts - Bindable Children Nodes*" for more details on the the Fog stack.

The *fogType* field controls how much of the fog color is blended with the object as a function of distance. If *fogType* is "LINEAR" (the default), then the amount of blending is a linear function of the distance, resulting in a depth cuing effect. If *fogType* is "EXPONENTIAL" then an exponential increase in blending should be used, resulting in a more natural fog appearance.

For best visual results, the Background node (which is unaffected by the Fog node) should be the same color as the fog node. The Fog node can also be used in conjunction with the *visibilityLimit* field of NavigationInfo node to provide a smooth fade out of objects as they approach the far clipping plane.

See the section "*Concepts - Lighting Model*" for details on lighting calculations.

FontStyle

```
FontStyle {
  field SFString family       "SERIF"
  field SFBool   horizontal   TRUE
  field MFString justify      "BEGIN"
  field SFString language     ""
  field SFBool   leftToRight  TRUE
  field SFFloat  size         1.0
  field SFFloat  spacing      1.0
  field SFString style        "PLAIN"
  field SFBool   topToBottom  TRUE
}
```

The FontStyle node defines the size, font family, and style of text's font, as well as the direction of the text strings and any specific language rendering techniques that must be used for non-English text. See Text node for application of FontStyle.

The *size* field specifies the height (in object space units) of glyphs rendered and determines the spacing of adjacent lines of text. All subsequent strings advance in either X or Y by -(*size* * *spacing*).

Font Family and Style

Font attributes are defined with the family and style fields. It is up to the browser to assign specific fonts to the various attribute combinations.

The *family* field specifies a case-sensitive SFString value that may be **"SERIF"** (the default) for a serif font such as Times Roman; **"SANS"** for a sans-serif font such as Helvetica; or **"TYPEWRITER"** for a fixed-pitch font such as Courier. A *family* value of empty quotes, **""**, is identical to **"SERIF"**.

The *style* field specifies a case-sensitive SFString value that may be **"PLAIN"** (the default) for default plain type; **"BOLD"** for boldface type; **"ITALIC"** for italic type; or **"BOLDITALIC"** for bold and italic type. A *style* value of empty quotes, **""**, is identical to **"PLAIN"**.

Direction, Justification and Spacing

The *horizontal*, *leftToRight*, and *topToBottom* fields indicate the direction of the text. The *horizontal* field indicates whether the text advances horizontally in its major direction (*horizontal* = TRUE, the default) or vertically in its major direction (*horizontal* = FALSE). The *leftToRight* and *topToBottom* fields indicate direction of text advance in the major (characters within a single string) and minor (successive strings) axes of layout. Which field is used for the major direction and which is used for the minor direction is determined by the *horizontal* field.

For horizontal text (*horizontal* = TRUE), characters on each line of text advance in the positive X direction if *leftToRight* is TRUE or in the negative X direction if *leftToRight* is FALSE. Characters are advanced according to their natural advance width. Then each line of characters is advanced in the negative Y direction if *topToBottom* is TRUE or in the positive Y direction if *topToBottom* is FALSE. Lines are advanced by the amount of *size* * *spacing*.

For vertical text (*horizontal* = FALSE), characters on each line of text advance in the negative Y direction if *topToBottom* is TRUE or in the positive Y direction if *topToBottom* is FALSE. Characters are advanced according to their natural advance height. Then each line of characters is advanced in the positive X direction if *leftToRight* is TRUE or in the negative X direction if *leftToRight* is FALSE. Lines are advanced by the amount of *size* * *spacing*.

The *justify* field determines alignment of the above text layout relative to the origin of the object coordinate system. It is an MFString which can contain 2 values. The first value specifies alignment along the major axis and the second value specifies alignment along the minor axis, as determined by the *horizontal* field. A *justify* value of **""** is equivalent to the default value. If the second string, minor alignment, is not specified then it defaults to the value **"FIRST"**. Thus, *justify* values of **""**, **"BEGIN"**, and **["BEGIN" "FIRST"]** are equivalent.

The major alignment is along the X axis when *horizontal* is TRUE and along the Y axis when *horizontal is* FALSE. The minor alignment is along the Y axis when *horizontal* is TRUE and along the X axis when *horizontal is* FALSE. The possible values for each enumerant of the *justify* field are **"FIRST"**, **"BEGIN"**, **"MIDDLE"**, and **"END"**. For major alignment, each line of text is positioned individually according to the major alignment enumerant. For minor alignment, the block of text representing all lines together is positioned according to the minor alignment enumerant. The following table describes the behavior in terms of which portion of the text is at the origin:

Major Alignment, *horizontal* = TRUE:

Enumerant	***leftToRight* = TRUE**	***leftToRight* = FALSE**
FIRST	Left edge of each line	Right edge of each line
BEGIN	Left edge of each line	Right edge of each line
MIDDLE	Centered about X-axis	Centered about X-axis
END	Right edge of each line	Left edge of each line

Major Alignment, *horizontal* = FALSE:

Enumerant	***topToBottom* = TRUE**	***topToBottom* = FALSE**
FIRST	Top edge of each line	Bottom edge of each line
BEGIN	Top edge of each line	Bottom edge of each line
MIDDLE	Centered about Y-axis	Center about Y-axis
END	Bottom edge of each line	Top edge of each line

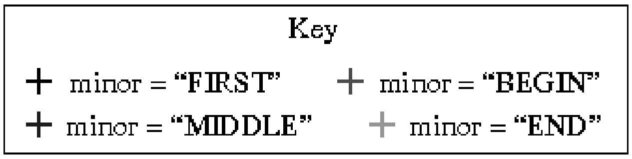
Minor Alignment, *horizontal* = TRUE:

Enumerant	***topToBottom* = TRUE**	***topToBottom* = FALSE**
FIRST	Baseline of first line	Baseline of first line
BEGIN	Top edge of first line	Bottom edge of first line
MIDDLE	Centered about Y-axis	Centered about Y-axis
END	Bottom edge of last line	Top edge of last line

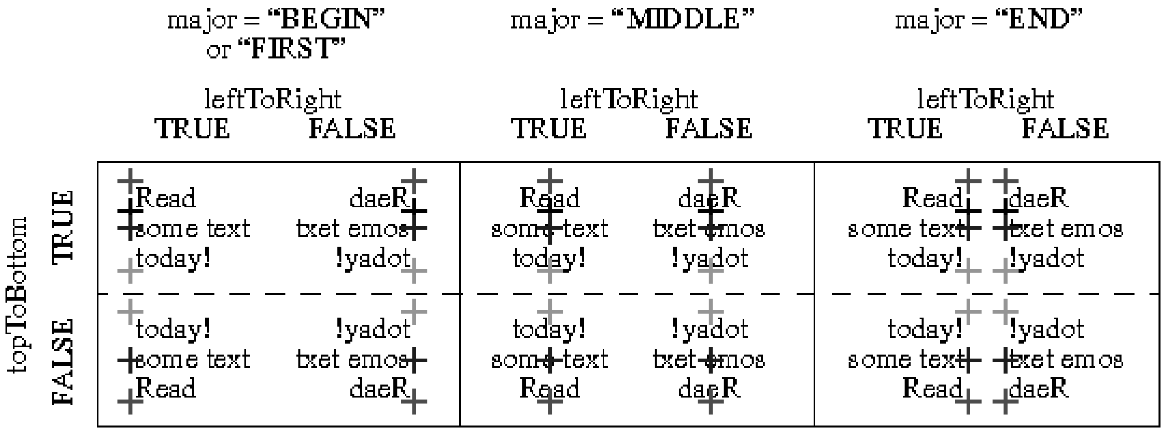
Minor Alignment, *horizontal* = FALSE:

Enumerant	***leftToRight* = TRUE**	***leftToRight* = FALSE**
FIRST	Left edge of first line	Right edge of first line
BEGIN	Left edge of first line	Right edge of first line
MIDDLE	Centered about X-axis	Centered about X-axis
END	Right edge of last line	Left edge of last line

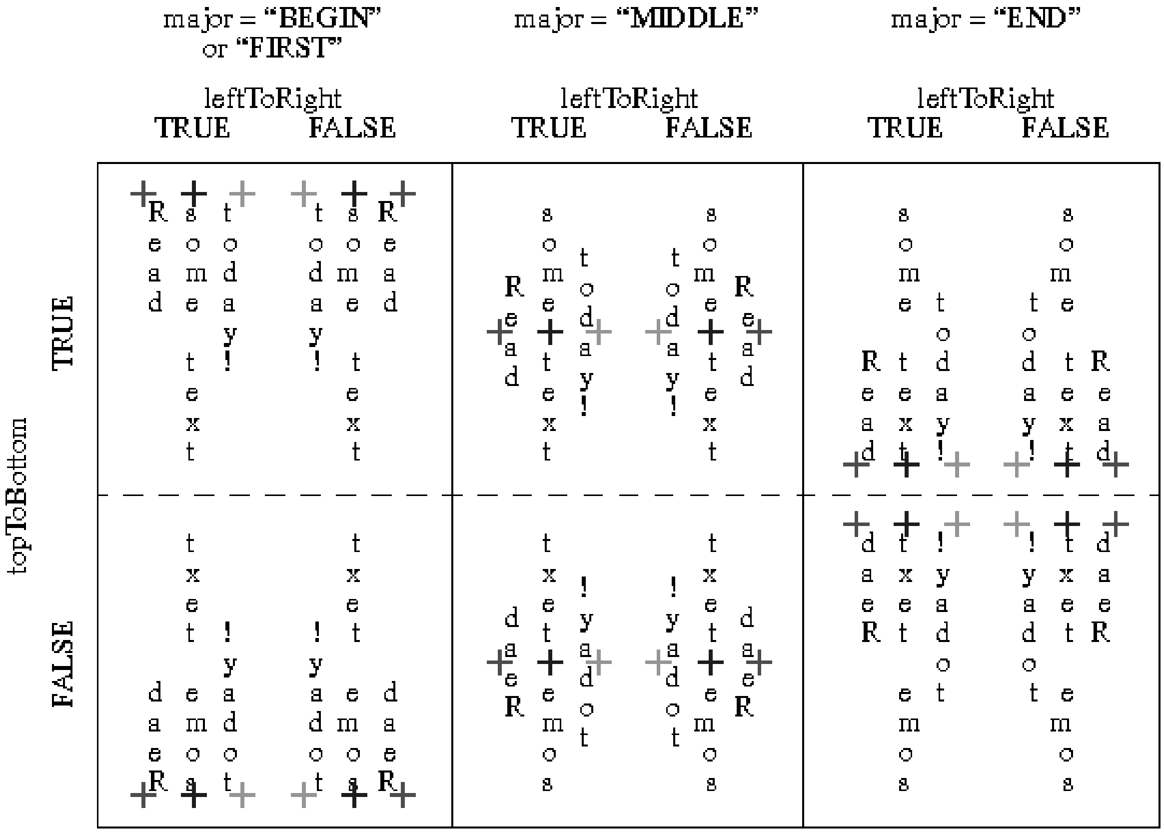
The default minor alignment is **"FIRST"**. This is a special case of minor alignment when *horizontal* is TRUE. Text starts at the baseline at the Y-axis. In all other cases, :**"FIRST"** is identical to **"BEGIN"**. In the following tables, each color-coded cross-hair indicates where the X and Y axes should be in relation to the text:



horizontal = TRUE:



horizontal = FALSE:



The *language* field specifies the context of the language for the text string. Due to the multilingual nature of the ISO 10646-1:1993, the *language* field is needed to provide a proper language attribute of the text string. The format is based on the POSIX locale specification as well as the RFC 1766: language[_territory]. The values for the language tag is based on the ISO 639, i.e. zh for Chinese, jp for Japanese, sc for Swedish. The territory tag is based on the ISO 3166 country code, i.e. TW is for Taiwan and CN for China for the "zh" Chinese language tag. If the *language* field is set to empty "", then local language bindings are used.

Please refer to these sites for more details:

```
http://www.chemie.fu-berlin.de/diverse/doc/ISO_639.html
http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html
```

■Group

```
Group {
  eventIn       MFNode  addChildren
  eventIn       MFNode  removeChildren
  exposedField  MFNode  children       []
  field         SFVec3f bboxCenter     0 0 0
  field         SFVec3f bboxSize       -1 -1 -1
}
```

A Group node is equivalent to a Transform node, without the transformation fields.

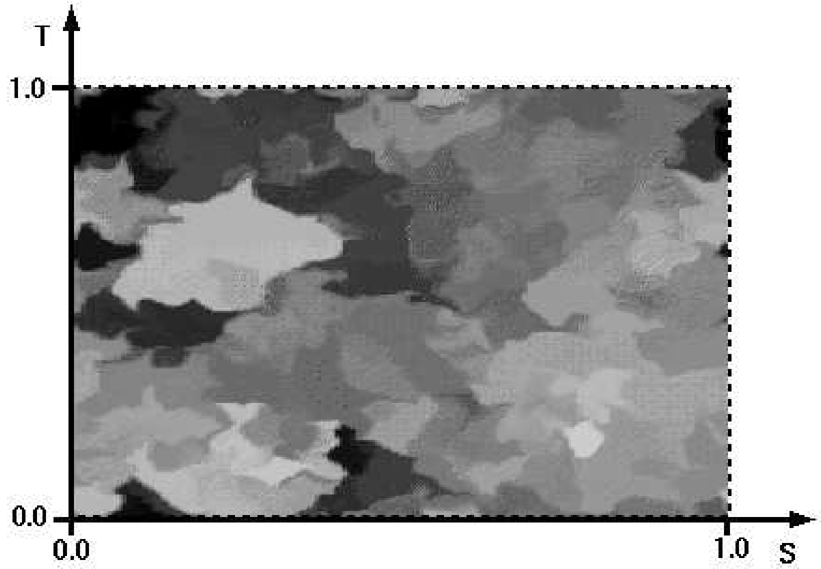
See the "*Concepts - Grouping and Children Nodes*" section for a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Group's children. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, then the results are undefined. A default *bboxSize* value, (-1 -1 -1), implies that the bounding box is not specified and if needed must be calculated by the browser. See "*Concepts - Bounding Boxes*" for a description of the *bboxCenter* and *bboxSize* fields.

ImageTexture

```
ImageTexture {
  exposedField MFString url      []
  field        SFBool   repeatS TRUE
  field        SFBool   repeatT TRUE
}
```

The ImageTexture node defines a texture map by specifying an image file and general parameters for mapping to geometry. Texture maps are defined in a 2D coordinate system, (s, t), that ranges from 0.0 to 1.0 in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to s=0, t=0, and the top-right pixel of the image corresponds to s=1, t=1.



The texture is read from the URL specified by the *url* field. To turn off texturing, set the *url* field to have no values ([]). Browsers are required to support the JPEG [JPEG] and PNG [PNG] image file formats, and in addition, may support any other image formats. Support for the GIF format [GIF] including transparent backgrounds is also recommended. See the section ""*Concepts - URLS and URNs*" for details on the *url* field.

Texture images may be one component (greyscale), two component (greyscale 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 perform 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 greyscale values in the texture image.
2. Diffuse color is multiplied by the greyscale 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.

See "*Concepts - Lighting Model*" for details on lighting equations and the interaction between textures, materials, and geometries.

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.

The *repeatS* and *repeatT* fields specify how the texture wraps in the S and T directions. If *repeatS* is TRUE (the default), the texture map is repeated outside the 0-to-1 texture coordinate range in the S direction so that it fills the shape. If *repeatS* is FALSE, the texture coordinates are clamped in the S direction to lie within the 0-to-1 range. The *repeatT* field is analogous to the *repeatS* field.

IndexedFaceSet

```
IndexedFaceSet {
  eventIn       MFInt32 set_colorIndex
  eventIn       MFInt32 set_coordIndex
  eventIn       MFInt32 set_normalIndex
  eventIn       MFInt32 set_texCoordIndex
  exposedField  SFNode  color             NULL
  exposedField  SFNode  coord             NULL
  exposedField  SFNode  normal            NULL
  exposedField  SFNode  texCoord          NULL
  field         SFBool  ccw               TRUE
  field         MFInt32 colorIndex        []
  field         SFBool  colorPerVertex    TRUE
  field         SFBool  convex            TRUE
  field         MFInt32 coordIndex        []
  field         SFFloat creaseAngle       0
  field         MFInt32 normalIndex       []
  field         SFBool  normalPerVertex   TRUE
  field         SFBool  solid             TRUE
```

```
  field          MFInt32 texCoordIndex     []
}
```

The IndexedFaceSet node represents a 3D shape formed by constructing faces (polygons) from vertices listed in the *coord* field. The *coord* field must contain a Coordinate node. IndexedFaceSet uses the indices in its *coordIndex* field to specify the polygonal faces. An index of -1 indicates that the current face has ended and the next one begins. The last face may (but does not have to be) followed by a -1. If the greatest index in the *coordIndex* field is N, then the Coordinate node must contain N+1 coordinates (indexed as 0-N). IndexedFaceSet is specified in the local coordinate system and is affected by parent transformations.

For descriptions of the *coord*, *normal*, and *texCoord* fields, see the Coordinate, Normal, and TextureCoordinate nodes.

See "*Concepts - Lighting Model*" for details on lighting equations and the interaction between textures, materials, and geometries.

If the color field is not NULL then it must contain a Color node, whose colors are applied to the vertices or faces of the IndexedFaceSet as follows:

- If *colorPerVertex* is FALSE, colors are applied to each face, as follows:
  - If the *colorIndex* field is not empty, then they are used to choose one color for each face of the IndexedFaceSet. There must be at least as many indices in the *colorIndex* field as there are faces in the IndexedFaceSet. If the greatest index in the *colorIndex* field is N, then there must be N+1 colors in the Color node. The *colorIndex* field must not contain any negative entries.
  - If the *colorIndex* field is empty, then the colors are applied to each face of the IndexedFaceSet in order. There must be at least as many colors in the Color node as there are faces.
- If *colorPerVertex* is TRUE, colors are applied to each vertex, as follows:
  - If the *colorIndex* field is not empty, then it is used to choose colors for each vertex of the IndexedFaceSet in exactly the same manner that the *coordIndex* field is used to choose coordinates for each vertex from the Coordinate node. The *colorIndex* field must contain at least as many indices as the *coordIndex* field, and must contain end-of-face markers (-1) in exactly the same places as the *coordIndex* field. If the greatest index in the *colorIndex* field is N, then there must be N+1 colors in the Color node.
  - If the *colorIndex* field is empty, then the *coordIndex* field is used to choose colors from the Color node. If the greatest index in the *coordIndex* field is N, then there must be N+1 colors in the Color node.

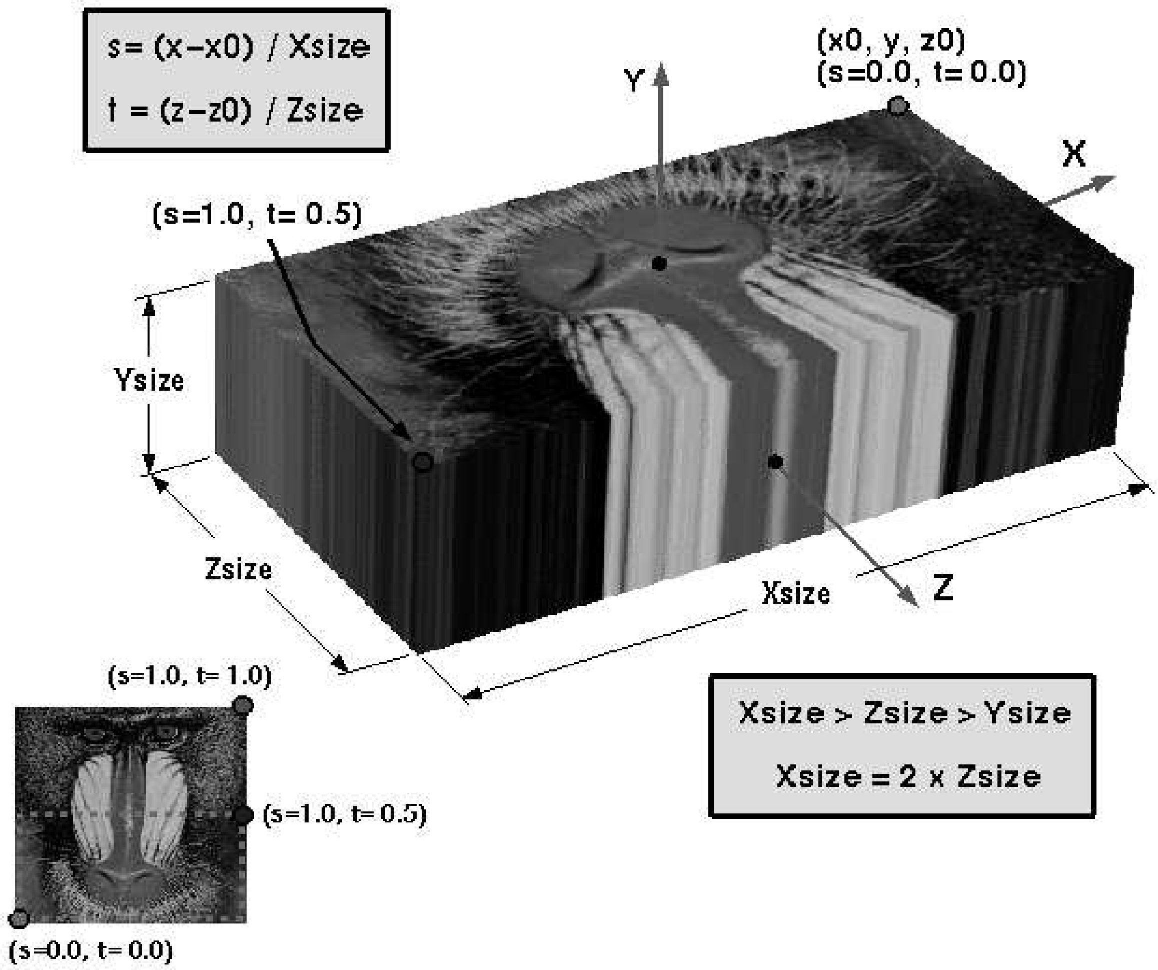
If the *normal* field is NULL, then the browser should automatically generate normals, using *creaseAngle* to determine if and how normals are smoothed across shared vertices.

If the *normal* field is not NULL, then it must contain a Normal node, whose normals are applied to the vertices or faces of the IndexedFaceSet in a manner exactly equivalent to that described above for applying colors to vertices/faces.

If the *texCoord* field is not NULL, then it must contain a TextureCoordinate node. The texture coordinates in that node are applied to the vertices of the IndexedFaceSet as follows:

- If the *texCoordIndex* field is not empty, then it is used to choose texture coordinates for each vertex of the IndexedFaceSet in exactly the same manner that the *coordIndex* field is used to choose coordinates for each vertex from the Coordinate node. The *texCoordIndex* field must contain at least as many indices as the *coordIndex* field, and must contain end-of-face markers (-1) in exactly the same places as the *coordIndex* field. If the greatest index in the *texCoordIndex* field is N, then there must be N+1 texture coordinates in the TextureCoordinate node.
- If the *texCoordIndex* field is empty, then the *coordIndex* array is used to choose texture coordinates from the TextureCoordinate node. If the greatest index in the *coordIndex* field is N, then there must be N+1 texture coordinates in the TextureCoordinate node.

If the *texCoord* field is NULL, 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. If two or all three dimensions of the bounding box are equal, then ties should be broken by choosing the X, Y, or Z dimension in that order of preference. 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. See the figure below for an illustration of default texture coordinates for a simple box shaped IndexedFaceSet with a bounding box with X dimension twice as large as the Z dimension which is twice as large as the Y dimension:



See the introductory "*Concepts - Geometry*" section for a description of the *ccw*, *solid*, *convex*, and *creaseAngle* fields.

IndexedLineSet

```
IndexedLineSet {
  eventIn       MFInt32 set_colorIndex
  eventIn       MFInt32 set_coordIndex
  exposedField  SFNode  color             NULL
  exposedField  SFNode  coord             NULL
  field         MFInt32 colorIndex        []
  field         SFBool  colorPerVertex    TRUE
  field         MFInt32 coordIndex        []
}
```

The IndexedLineSet node represents a 3D geometry formed by constructing polylines from 3D points specified in the *coord* field. IndexedLineSet uses the indices in its *coordIndex* field to specify the polylines by connecting together points from the *coord* field. An index of -1 indicates that the current polyline has ended and the next one begins. The last polyline may (but does not have to be) followed by a -1. IndexedLineSet is specified in the local coordinate system and is affected by parent transformations.

The *coord* field specifies the 3D vertices of the line set and is specified by a Coordinate node.

Lines are not lit, not texture-mapped, or not collided with during collision detection.

If the *color* field is not NULL, it must contain a Color node, and the colors are applied to the line(s) as follows:

- If *colorPerVertex* is FALSE:
    - If the *colorIndex* field is not empty, then one color is used for each polyline of the IndexedLineSet. There must be at least as many indices in the *colorIndex* field as there are polylines in the IndexedLineSet. If the greatest index in the *colorIndex* field is N, then there must be N+1 colors in the Color node. The *colorIndex* field must not contain any negative entries.
    - If the *colorIndex* field is empty, then the colors are applied to each polyline of the IndexedLineSet in order. There must be at least as many colors in the Color node as there are polylines.
- If *colorPerVertex* is TRUE:
    - If the *colorIndex* field is not empty, then colors are applied to each vertex of the IndexedLineSet in exactly the same manner that the *coordIndex* field is used to supply coordinates for each vertex from the Coordinate node. The *colorIndex* field must contain at least as many indices as the *coordIndex* field and must contain end-of-polyline markers (-1) in exactly the same places as the *coordIndex* field. If the greatest index in the *colorIndex* field is N, then there must be N+1 colors in the Color node.
    - If the *colorIndex* field is empty, then the *coordIndex* field is used to choose colors from the Color node. If the greatest index in the *coordIndex* field is N, then there must be N+1 colors

in the Color node.

If the *color* field is NULL and there is a Material defined for the Appearance affecting this IndexedLineSet, then use the *emissiveColor* of the Material to draw the lines. See "*Concepts - Lighting Model, Lighting Off*" for details on lighting equations.

Inline

```
Inline {
  exposedField MFString url        []
  field        SFVec3f  bboxCenter 0 0 0
  field        SFVec3f  bboxSize   -1 -1 -1
}
```

The Inline node is a grouping node that reads its children data from a location in the World Wide Web. Exactly when its children are read and displayed is not defined; reading the children may be delayed until the Inline is actually visible to the viewer. The *url* field specifies the URL containing the children. An Inline with an empty URL does nothing.

An Inline’s URLs shall refer to a valid VRML file that contains a list of children nodes at the top level. See "*Concepts - Grouping and Children Nodes*". The results are undefined if the URL refers to a file that is not VRML or if the file contains non-children nodes at the top level.

If multiple URLs are specified, the browser may display a URL of a lower preference file while it is obtaining, or if it is unable to obtain the higher preference file. See "*Concepts - URLS and URNs*" for details on the *url* field and preference order.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Inlines’s children. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, then the results are undefined. A default *bboxSize* value, (-1 -1 -1), implies that the bounding box is not specified and if needed must be calculated by the browser. See "*Concepts - Bounding Boxes*" for a description of the *bboxCenter* and *bboxSize* fields.

LOD

```
LOD {
  exposedField MFNode  level    []
  field        SFVec3f center   0 0 0
  field        MFFloat range    []
}
```

The LOD node specifies various levels of detail or complexity for a given object, and provides hints for browsers to automatically choose the appropriate version of the object based on the distance from the user. The *level* field contains a list of nodes that represent the same object or objects at varying levels of detail, from highest to the lowest level of detail, and the *range* field specifies the ideal distances at which to switch between the levels. See the "*Concepts - Grouping and Children Nodes*" section for a details on the types of nodes that are legal values for *level*.

The *center* field is a translation offset in the local coordinate system that specifies the center of the LOD object for distance calculations. In order to calculate which level to display, first the distance is calculated from the viewpoint, transformed into the local coordinate space of the LOD node, (including any scaling transformations), to the *center* point of the LOD. If the distance is less than the first value in the *range* field, then the first level of the LOD is drawn. If between the first and second values in the *range* field, the second level is drawn, and so on.

If there are N values in the *range* field, the LOD shall have N+1 nodes in its *level* field. Specifying too few levels will result in the last level being used repeatedly for the lowest levels of detail; if more levels than ranges are specified, the extra levels will be ignored. The exception to this rule is to leave the range field empty, which is a hint to the browser that it should choose a level automatically to maintain a constant display rate. Each value in the *range* field 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 smooth. Browsers may 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 an Inline node) is fetched, or might disregard the author-specified ranges for any other implementation-dependent reason. For best results, specify ranges only where necessary, and nest LOD nodes with and without ranges. Browsers should try to honor the hints given by authors, and authors should try to give browsers as much freedom as they can to choose levels of detail based on performance.

LOD nodes are evaluated top-down in the scene graph. Only the descendants of the currently selected level are rendered. Note that all nodes under an LOD node continue to receive and send events (i.e. routes) regardless of which LOD *level* is active. For example, if an active TimeSensor is contained within an inactive level of an LOD, the TimeSensor sends events regardless of the LOD’s state.

Material

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

The Material node specifies surface material properties for associated geometry nodes and are used by the VRML lighting equations during rendering. See "*Concepts - Lighting Model*" for a detailed description of the VRML lighting model equations.

All of the fields in the Material node range from 0.0 to 1.0.

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

- 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 *ambientIntensity* field specifies how much ambient light from light sources this surface should reflect. Ambient light is omni-directional and depends only on the number of light sources, not their positions with respect to the surface. Ambient color is calculated as *ambientIntensity* * *diffuseColor*.
- The *specularColor* and *shininess* determine the specular highlights--for example, 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.

[This section belong in the Conformance annex.]

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. Browsers need not support partial transparency, but should support at least fully transparent and fully opaque surfaces, treating transparency values >= 0.5 as fully transparent.

Issues for Low-End Rendering Systems. 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
ambientIntensity	No	Ignore
diffuseColor	Yes	Use
specularColor	No	Ignore
emissiveColor	No	If diffuse == 0.8 0.8 0.8, use emissive
shininess	Yes	Use
transparency	Yes	if < 0.5 then opaque else transparent

The emissive color field is used when all other colors are black (0 0 0). Rendering systems which do not support specular color may nevertheless support a specular intensity. This should be derived by taking the dot product of the specified RGB specular value with the vector [.32 .57 .11]. This adjusts the color value to compensate for the variable sensitivity of the eye to colors.

Likewise, if a system supports ambient intensity but not color, the same thing should be done with the ambient color values to generate the ambient intensity. 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.

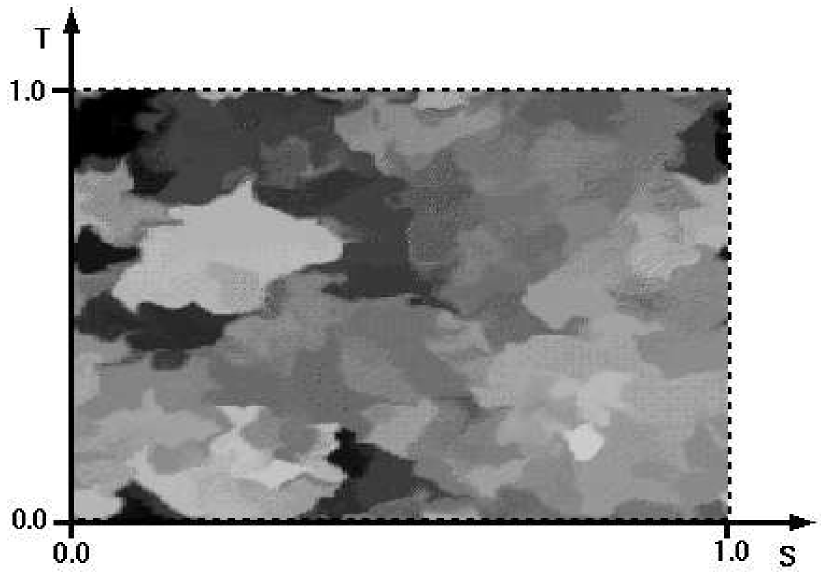
It is also expected that simpler rendering systems may be unable to support both diffuse and emissive objects in the same world. Also, many renderers will not support *ambientIntensity* with per-vertex colors specified with the Color node.

MovieTexture

```
MovieTexture {
  exposedField SFBool   loop             FALSE
  exposedField SFFloat  speed            1
  exposedField SFTime   startTime        0
  exposedField SFTime   stopTime         0
  exposedField MFString url              []
  field        SFBool   repeatS          TRUE
  field        SFBool   repeatT          TRUE
  eventOut     SFFloat  duration_changed
  eventOut     SFBool   isActive
}
```

The MovieTexture node defines a time dependent texture map (contained in a movie file) and parameters for controlling the movie and the texture mapping. A MovieTexture can also be used as the source of sound data for a Sound node, but in this special case are not used for rendering.

Texture maps are defined in a 2D coordinate system, (s, t), that ranges from 0.0 to 1.0 in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to s=0, t=0, and the top-right pixel of the image corresponds to s=1, t=1.



The *url* field that defines the movie data must support MPEG1-Systems (audio and video) or MPEG1-Video (video-only) movie file formats [MPEG]. See "*Concepts - URLS and URNs*" for details on the *url* field. It is recommended that implementations support greyscale or alpha transparency rendering if the specific movie format being used supports these features.

See "*Concepts - Lighting Model*" for details on lighting equations and the interaction between textures, materials, and geometries.

As soon as the movie is loaded, a *duration_changed* eventOut is sent. This indicates the duration of the movie, in seconds. This eventOut value can be read (for instance, by a Script) to determine the duration of a movie. A value of -1 implies the movie has not yet loaded or the value is unavailable for some reason.

The *loop, startTime,* and *stopTime* exposedFields and the *isActive* eventOut, and their affects on the MovieTexture node, are discussed in detail in the "*Concepts - Time Dependent Nodes*" section. The "*cycle"* of a MovieTexture is the length of time in seconds for one playing of the movie at the specified *speed*.

If a MovieTexture is inactive when the movie is first loaded, then frame 0 is shown in the texture if *speed* is non-negative, or the last frame of the movie if *speed* is negative. A MovieTexture will always display frame 0 if *speed* = 0. For positive values of *speed*, the frame an active MovieTexture will display at time `now` corresponds to the frame at movie time (i.e., in the movie's local time system with frame 0 at time 0, at *speed* = 1):

```
fmod (now - startTime, duration/speed)
```

If *speed* is negative, then the frame to display is the frame at movie time:

```
duration + fmod(now - startTime, duration/speed).
```

When a MovieTexture becomes inactive, the frame corresponding to the time at which the MovieTexture became inactive will remain as the texture.

The *speed* exposedField indicates how fast the movie should be played. A *speed* of 2 indicates the movie plays twice as fast. Note that the *duration_changed* output is not affected by the *speed* exposedField. *set_speed* events are ignored while the movie is playing. A negative *speed* implies that the movie will play backwards. However, content creators should note that this may not work for streaming movies or very large movie files.

MovieTextures can be referenced by an Appearance node's texture field (as a movie texture) and by a Sound node's source field (as an audio source only). A legal implementation of the MovieTexture node is not required to play audio if *speed* is not equal to 1.

NavigationInfo

```
NavigationInfo {
  eventIn      SFBool   set_bind
```

```
  exposedField MFFloat  avatarSize       [ 0.25, 1.6, 0.75 ]
  exposedField SFBool   headlight        TRUE
  exposedField SFFloat  speed            1.0
  exposedField MFString type             "WALK"
  exposedField SFFloat  visibilityLimit  0.0
  eventOut     SFBool   isBound
}
```

The NavigationInfo node contains information describing the physical characteristics of the viewer and viewing model. NavigationInfo is a bindable node (see "*Concepts - Bindable Children Nodes")* and thus there exists a NavigationInfo stack in the browser in which the top-most NavigationInfo on the stack is the currently active NavigationInfo. The current NavigationInfo is considered to be a child of the current Viewpoint - regardless of where it is initially located in the file. Whenever the current Viewpoint changes, the current NavigationInfo must be re-parented to it. Whenever the current NavigationInfo changes, the new NavigationInfo must be re-parented to the current Viewpoint.

If a TRUE value is sent to the *set_bind* eventIn of a NavigationInfo, it is pushed onto the NavigationInfo stack and activated. When a NavigationInfo is bound, the browser uses the fields of the NavigationInfo to set the navigation controls of its user interface and the NavigationInfo is conceptually re-parented under the currently bound Viewpoint. All subsequent scaling changes to the current Viewpoint's coordinate system automatically change aspects (see below) of the NavigationInfo values used in the browser (e.g. scale changes to any parent transformation). A FALSE value of *set_bind*, pops the NavigationInfo from the stack, results in an *isBound* FALSE event, and pops to the next entry in the stack which must be re-parented to the current Viewpoint. See "*Concepts - Bindable Children Nodes*" for more details on the the binding stacks.

The *type* field specifies a navigation paradigm to use. Minimally, browsers shall support the following navigation types: "WALK", "EXAMINE", "FLY", and "NONE". Walk navigation is used for exploring a virtual world. It is recommended that the browser should support a notion of gravity in walk mode. Fly navigation is similar to walk except that no notion of gravity should be enforced. There should still be some notion of "up" however. Examine navigation is typically used to view individual objects and often includes (but does not require) the ability to spin the object and move it closer or further away. The "none" choice removes all navigation controls - the user navigates using only controls provided in the scene, such as guided tours. Also allowed are browser specific navigation types. These should include a unique suffix (e.g. _sgi.com) to prevent conflicts. The *type* field is multi-valued so that authors can specify fallbacks in case a browser does not understand a given type. If none of the types are recognized by the browser, then the default "WALK" is used. These strings values are case sensitive ("walk" is not equal to "WALK").

The *speed* is the rate at which the viewer travels through a scene in meters per second. Since viewers may provide mechanisms to travel faster or slower, this should be the default or average speed of the viewer. If the NavigationInfo *type* is EXAMINE, *speed* should affect panning and dollying--it should have no effect on the rotation speed. The transformation hierarchy of the currently bound Viewpoint (see above) scales the *speed* - translations and rotations have no effect on *speed*. Speed must be >= 0.0 - where 0.0 specifies a stationary avatar.

The *avatarSize* field specifies the user's physical dimensions in the world for the purpose of collision detection and terrain following. It is a multi-value field to allow several dimensions to be specified. The first value should be the allowable distance between the user's position and any collision geometry (as specified by Collision) before a collision is detected. The second should be the height above the terrain the viewer should be maintained. The third should be the height of the tallest object over which the viewer can "step". This allows staircases to be built with dimensions that can be ascended by all browsers. Additional values are browser dependent and all values may be ignored, but if a browser interprets these values the first 3 should be interpreted as described above. The transformation hierarchy of the currently bound Viewpoint scales the *avatarSize* - translations and rotations have no effect on *avatarSize*.

For purposes of terrain following the browser needs a notion of the *down* direction (down vector), since gravity is applied in the direction of the down vector. This down vector should be along the negative Y-axis in the local coordinate system of the currently bound Viewpoint (i.e., the accumulation of the Viewpoint's parent transformations, not including the Viewpoint's orientation field).

The *visibilityLimit* field sets the furthest distance the user is able to see. The browser may clip all objects beyond this limit, fade them into the background or ignore this field. A value of 0.0 (the default) indicates an infinite visibility limit. *VisibilityLimit* is restricted to be >= 0.0.

The *speed*, *avatarSize* and *visibilityLimit* values are all scaled by the transformation being applied to currently bound Viewpoint. If there is no currently bound Viewpoint, they are interpreted in the world coordinate system. This allows these values to be automatically adjusted when binding to a Viewpoint that has a scaling transformation applied to it without requiring a new NavigationInfo node to be bound as well. If the scale applied to the Viewpoint is non-uniform the behavior is undefined.

The *headlight* field specifies whether a browser should turn a headlight on. A headlight is a directional light that always points in the direction the user is looking. Setting this field to TRUE allows the browser to provide a headlight, possibly with user interface controls to turn it on and off. Scenes that enlist pre-computed lighting (e.g. radiosity solutions) can turn the headlight off. The headlight shall have *intensity* = 1, *color* = 1 1 1, *ambientIntensity* = 0.0, and *direction* = 0 0 -1.

It is recommended that the near clipping plane should be set to one-half of the collision radius as specified in the *avatarSize* field. This recommendation may be ignored by the browser, but setting the near plane to this value prevents excessive clipping of objects just above the collision volume and provides a region inside the collision volume for content authors to include geometry that should remain fixed relative to the viewer, such as icons or a heads-up display, but that should not be occluded by geometry outside of the collision volume.

The first NavigationInfo node found during reading of the world is automatically bound (receives a *set_bind* TRUE event) and supplies the initial navigation parameters.

Normal

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

This node defines a set of 3D surface normal vectors to be used in the *vector* field of some geometry nodes (IndexedFaceSet, ElevationGrid). This node contains one multiple-valued field that contains the normal vectors. Normals should be unit-length or results are undefined.

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.

NormalInterpolator

```
NormalInterpolator {
  eventIn      SFFloat set_fraction
  exposedField MFFloat key           []
  exposedField MFVec3f keyValue      []
  eventOut     MFVec3f value_changed
}
```

This node interpolates among a set of multi-valued Vec3f values, suitable for transforming normal vectors. All output vectors will have been normalized by the interpolator.

The number of normals in the *keyValue* field must be an integer multiple of the number of keyframes in the *key* field; that integer multiple defines how many normals will be contained in the *value_changed* events.

Normal interpolation is to be performed on the surface of the unit sphere. That is, the output values for a linear interpolation from a point P on the unit sphere to a point Q also on unit sphere should lie along the shortest arc (on the unit sphere) connecting points P and Q. Also, equally spaced input fractions will result in arcs of equal length. Cases where P and Q are diagonally opposing allow an infinite number of arcs. The interpolation for this case can be along any one of these arcs.

Refer to "*Concepts - Interpolators*" for a more detailed discussion of interpolators.

OrientationInterpolator

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

This node interpolates among a set of SFRotation values. The rotations are absolute in object space and are, therefore, not cumulative. The *keyValue* field must contain exactly as many rotations as there are keyframes in the *key* field, or an error will be generated and results will be undefined.

An orientation represents the final position of an object after a rotation has been applied. An OrientationInterpolator will interpolate between two orientations by computing the shortest path on the unit sphere between the two orientations. The interpolation will be linear in arc length along this path. The path between two diagonally opposed orientations will be any one of the infinite possible paths with arc length PI.

If two consecutive keyValue values exist such that the arc length between them is greater than PI, then the interpolation will take place on the arc complement. For example, the interpolation between the orientations:

```
0 1 0 0 --> 0 1 0 5.0
```

is equivalent to the rotation between the two orientations:

```
0 1 0 2*PI --> 0 1 0 5.0
```

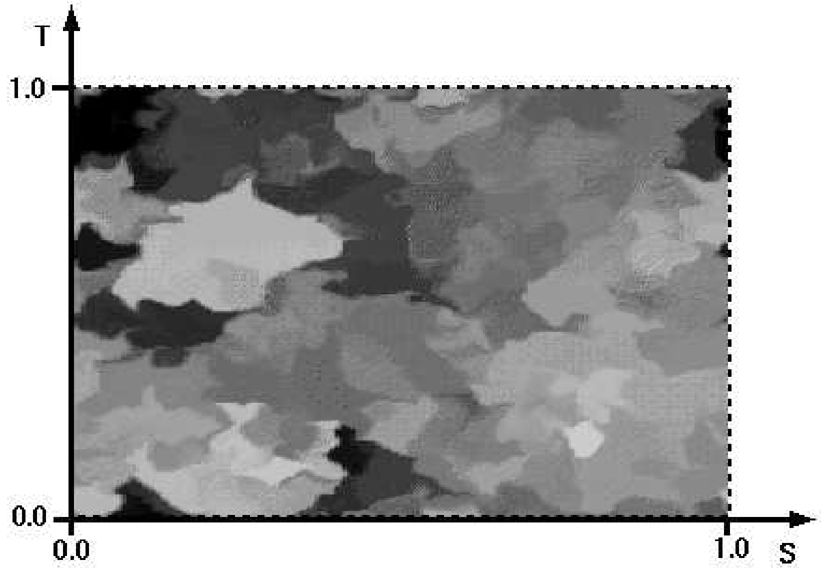
Refer to "*Concepts - Interpolators*" for a more detailed discussion of interpolators.

PixelTexture

```
PixelTexture {
  exposedField SFImage  image      0 0 0
  field        SFBool   repeatS    TRUE
  field        SFBool   repeatT    TRUE
}
```

The PixelTexture node defines a 2D image-based texture map as an explicit array of pixel values and parameters controlling tiling repetition of the texture onto geometry.

Texture maps are defined in a 2D coordinate system, (s, t), that ranges from 0.0 to 1.0 in both directions. The bottom edge of the pixel image corresponds to the S-axis of the texture map, and left edge of the pixel image corresponds to the T-axis of the texture map. The lower-left pixel of the pixel image corresponds to s=0, t=0, and the top-right pixel of the image corresponds to s=1, t=1.



Images may be one component (greyscale), two component (greyscale plus alpha opacity), three component (full RGB color), or four-component (full RGB color plus alpha opacity). An ideal VRML implementation will use the texture image to modify the diffuse color and transparency (= 1 - alpha opacity) of an object's material (specified in a Material node), then perform 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 greyscale values in the texture image.
2. Diffuse color is multiplied by the greyscale 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.

See "*Concepts - Lighting Model*" for details on the VRML lighting equations.

See the "*Field Reference - SFImage*" specification for details on how to specify an image.

The *repeatS* and *repeatT* fields specify how the texture wraps in the S and T directions. If *repeatS* is TRUE (the default), the texture map is repeated outside the 0-to-1 texture coordinate range in the S direction so that it fills the shape. If *repeatS* is FALSE, the texture coordinates are clamped in the S direction to lie within the 0-to-1 range. The *repeatT* field is analogous to the *repeatS* field.

PlaneSensor

```
PlaneSensor {
  exposedField SFBool  autoOffset          TRUE
  exposedField SFBool  enabled             TRUE
  exposedField SFVec2f maxPosition         -1 -1
  exposedField SFVec2f minPosition         0 0
  exposedField SFVec3f offset              0 0 0
  eventOut     SFBool  isActive
  eventOut     SFVec3f trackPoint_changed
  eventOut     SFVec3f translation_changed
}
```

The PlaneSensor maps pointing device (e.g. mouse or wand) motion into translation in two dimensions, in the XY plane of its local space. PlaneSensor uses the descendant geometry of its parent node to determine if a hit occurs.

The *enabled* exposed field enables and disables the PlaneSensor - if TRUE, the sensor reacts appropriately to user events, if FALSE, the sensor does not track user input or send output events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event the sensor is enabled and ready for user activation.

The PlaneSensor generates events if the pointing device is activated while over any descendant geometry nodes of its parent group and then moved while activated. Typically, the pointing device is a 2D device such as a mouse. The pointing device is considered to be moving within a plane at a fixed distance from the viewer and perpendicular to the line of sight; this establishes a set of 3D coordinates for the pointer. If a 3D pointer is in use, then the sensor generates events only when the pointer is within the user's field of view. In either case, the pointing device is considered to "pass over" geometry when that geometry is intersected by a line extending from the viewer and passing through the pointer's 3D coordinates. If multiple sensors' geometry intersect this line (hereafter called the bearing), only the nearest will be eligible to generate events.

Upon activation of the pointing device (e.g. mouse button down) over the sensor's geometry, an *isActive* TRUE event is sent. Dragging motion is mapped into a relative translation in the XY plane of the sensor's local coordinate system as it was defined at the time of activation. For each subsequent position of the bearing, a *translation_changed* event is output which corresponds to a relative translation from the original intersection point projected onto the XY plane, plus the *offset* value. The sign of the translation is defined by the XY plane of the sensor's coordinate system. *trackPoint_changed* events reflect the unclamped drag position on the surface of this plane. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last translation value and an *offset_changed* event is generated. See "*Concepts - Drag Sensors*" for more details.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it releases and generates an *isActive* FALSE event (other pointing device sensors cannot generate events during this time). Motion of the pointing device while *isActive* is TRUE is referred to as a "drag". If a 2D pointing device is in use, *isActive* events will typically reflect the state of the primary button associated with the device (i.e. *isActive* is TRUE when the primary button is pressed, and FALSE when not released). If a 3D pointing device (e.g. wand) is in use, *isActive* events will typically reflect whether the pointer is within or in contact with the sensor's geometry.

minPosition and *maxPosition* may be set to clamp *translation* events to a range of values as measured from the origin of the XY plane. If the X or Y component of *minPosition* is greater than the corresponding component of *maxPosition*, *translation_changed* events are not clamped in that dimension. If the X or Y component of *minPosition* is equal to the corresponding component of *maxPosition*, that component is constrained to the given value; this technique provides a way to implement a line sensor that maps dragging motion into a translation in one dimension.

While the pointing device is activated, *trackPoint_changed* and *translation_changed* events are output. *trackPoint_changed* events represent the unclamped intersection points on the surface of the local XY plane. If the pointing device is dragged off of the XY plane while activated (e.g. above horizon line), browsers may interpret this in several ways (e.g. clamp all values to the horizon). Each movement of the pointing device, while *isActive* is TRUE, generates *trackPoint_changed* and *translation_changed* events.

See "*Concepts - Pointing Device Sensors and Drag Sensors*" for more details.

PointLight

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

The PointLight node specifies a point light source at 3D location in the local coordinate system. A point source emits light equally in all directions; that is, it is omni-directional. PointLights are specified in their local coordinate system and are affected by parent transformations.

See "*Concepts - Light Sources*" for a detailed description of the *ambientIntensity*, *color*, and *intensity* fields.

A PointLight may illuminate geometry within *radius* (>= 0.0) meters of its *location*. Both radius and location are affected by parent transformations (scale *radius* and transform *location*).

A PointLight's illumination falls off with distance as specified by three *attenuation* coefficients. The attenuation factor is *1/(attenuation[0] + attenuation[1]*r + attenuation[2]*r^2)*, where *r* is the distance of the light to the surface being illuminated. The default is no attenuation. An *attenuation* value of **0 0 0** is identical to **1 0 0**. Attenuation values must be >= 0.0. Renderers that do not support a full attenuation model may approximate as necessary. See "*Concepts - Lighting Model*" for a detailed description of VRML's lighting equations.

PointSet

```
PointSet {
  exposedField  SFNode  color      NULL
  exposedField  SFNode  coord      NULL
}
```

The PointSet node specifies a set of 3D points in the local coordinate system with associated colors at each point. The *coord* field specifies a Coordinate node (or instance of a Coordinate node) - results are undefined if the *coord* field specifies any other type of node. PointSet uses the coordinates in order. If the *coord* field is NULL, then the PointSet is empty.

PointSets are not lit, not texture-mapped, or collided with during collision detection.

If the *color* field is not NULL, it must specify a Color node that contains at least the number of points contained in the *coord* node - results are undefined if the *color* field specifies any other type of node. Colors shall be applied to each point in order. The results are undefined if the number of values in the Color node is less than the number of values specified in the Coordinate node

If the *color* field is NULL and there is a Material defined for the Appearance affecting this PointSet, then use the *emissiveColor* of the Material to draw the points. See "*Concepts - Lighting Model, Lighting Off*" for details on lighting equations.

PositionInterpolator

```
PositionInterpolator {
  eventIn      SFFloat set_fraction
  exposedField MFFloat key            []
  exposedField MFVec3f keyValue       []
  eventOut     SFVec3f value_changed
}
```

This node linearly interpolates among a set of SFVec3f values. This is appropriate for interpolating a translation. The vectors are interpreted as absolute positions in object space. The *keyValue* field must contain exactly as many values as in the *key* field.

Refer to "*Concepts - Interpolators*" for a more detailed discussion of interpolators.

ProximitySensor

```
ProximitySensor {
  exposedField SFVec3f    center       0 0 0
  exposedField SFVec3f    size         0 0 0
  exposedField SFBool     enabled      TRUE
  eventOut     SFBool     isActive
  eventOut     SFVec3f    position_changed
```

```
  eventOut      SFRotation orientation_changed
  eventOut      SFTime     enterTime
  eventOut      SFTime     exitTime
}
```

The ProximitySensor generate events when the user enters, exits, and moves within a region in space (defined by a box). A proximity sensor can be enabled or disabled by sending it an *enabled* event with a value of TRUE or FALSE - a disabled sensor does not send output events.

A ProximitySensor generates *isActive* TRUE/FALSE events as the viewer enters and exits the rectangular box defined by its *center* and *size* fields. Browsers shall interpolate user positions and timestamp the *isActive* events with the exact time the user first intersected the proximity region. The *center* field defines the center point of the proximity region in object space, and the *size* field specifies a vector which defines the width (x), height (y), and depth (z) of the box bounding the region. ProximitySensor nodes are affected by the hierarchical transformations of its parents.

The *enterTime* event is generated whenever the *isActive* TRUE event is generated (user enters the box), and *exitTime* events are generated whenever *isActive* FALSE event is generated (user exits the box).

The *position_changed* and *orientation_changed* events send events whenever the position and orientation of the viewer changes with respect to the ProximitySensor's coordinate system - this includes enter and exit times. Note that the user movement may be as a result of a variety of circumstances (e.g. browser navigation, proximity sensor's coordinate system changes, bound Viewpoint's position or orientation changes, or the ProximitySensor's coordinate system changes).

Each ProximitySensor behaves independently of all other ProximitySensors - every enabled ProximitySensor that is effected by the user's movement receives and sends events, possibly resulting in multiple ProximitySensors receiving and sending events simultaneously. Unlike TouchSensors, there is no notion of a ProximitySensor lower in the scene graph "grabbing" events.

Instanced (DEF/USE) ProximitySensors use the union of all the boxes to check for enter and exit - an instanced ProximitySensor will detect enter and exit for all instances of the box and send output events appropriately.

A ProximitySensor that surrounds the entire world will have an enterTime equal to the time that the world was entered and can be used to start up animations or behaviors as soon as a world is loaded. A ProximitySensor with a (0 0 0) *size* field cannot generate events - this is equivalent to setting the *enabled* field to FALSE.

ScalarInterpolator

```
ScalarInterpolator {
  eventIn       SFFloat set_fraction
  exposedField MFFloat key            []
  exposedField MFFloat keyValue       []
  eventOut      SFFloat value_changed
}
```

This node linearly interpolates among a set of SFFloat values. This interpolator is appropriate for any parameter defined using a single floating point value, e.g., width, radius, intensity, etc. The *keyValue* field must contain exactly as many numbers as there are keyframes in the *key* field.

Refer to "*Concepts - Interpolators*" for a more detailed discussion of interpolators.

Script

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

The Script node is used to program behavior in a scene. Script nodes typically receive events that signify a change or user action, contain a program module that performs some computation, and effect change somewhere else in the scene by sending output events. Each Script node has associated programming language code, referenced by the *url* field, that is executed to carry out the Script node's function. That code will be referred to as "the script" in the rest of this description.

Browsers are not required to support any specific language. See the section in "*Concepts - Scripting*" for detailed information on scripting languages. Browsers are required to adhere to the language bindings of languages specified in annexes of the specification. See the section "*Concepts - URLS and URNs*" for details on the *url* field.

When the script is created, any language-dependent or user-defined initialization is performed. The script is able to receive and process events that are sent to it. Each event that can be received must be declared in the Script node using the same syntax as is used in a prototype definition:

```
eventIn type name
```

The *type* can be any of the standard VRML fields (see "Field Reference"), and *name* must be an identifier that is unique for this Script node.

The Script node should be able to generate events in response to the incoming events. Each event that can be generated must be declared in the Script node using the following syntax:

```
eventOut type name
```

Script nodes cannot have exposedFields. The implementation ramifications of exposedFields is far too complex and thus not allowed.

If the Script node's *mustEvaluate* field is FALSE, the browser can delay sending input events to the script until its outputs are needed by the browser. If the *mustEvaluate* field is TRUE, the browser should send input events to the script as soon as possible, regardless of whether the outputs are needed. The *mustEvaluate* field should be set to TRUE only if the Script has effects that are not known to the browser (such as sending information across the network); otherwise, poor performance may result.

Once the script has access to a VRML node (via an SFNode or MFNode value either in one of the Script node's fields or passed in as an eventIn), the script should be able to read the contents of that node's exposed field. If the Script node's *directOutput* field is TRUE, the script may also send events directly to any node to which it has access, and may dynamically establish or break routes. If *directOutput* is FALSE (the default), then the script may only affect the rest of the world via events sent through its eventOuts.

A script is able to communicate directly with the VRML browser to get the current time, the current world URL, and so on. This is strictly defined by the API for the specific language being used.

It is expected that all other functionality (such as networking capabilities, multi-threading capabilities, and so on) will be provided by the scripting language.

The location of the Script node in the scene graph has no affect on its operation. For example, if a parent of a Script node is a Switch node with *whichChoice* set to -1 (i.e. ignore its children), the Script continues to operate as specified (receives and sends events).

Shape

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

The Shape node has two fields: *appearance* and *geometry* which are used to create rendered objects in the world. The *appearance* field specifies an Appearance node that specifies the visual attributes (e.g. material and texture) to be applied to the geometry . The *geometry* field specifies a geometry node. The specified geometry node is rendered with the specified appearance nodes applied.

See "*Concepts - Lighting Model*" for details of the VRML lighting model and the interaction between Appearance and geometry nodes.

If the *geometry* field is NULL the object is not drawn.

Sound

```
Sound {
  exposedField SFVec3f  direction     0 0 1
  exposedField SFFloat  intensity     1
  exposedField SFVec3f  location      0 0 0
  exposedField SFFloat  maxBack       10
```

```
  exposedField SFFloat  maxFront      10
  exposedField SFFloat  minBack       1
  exposedField SFFloat  minFront      1
  exposedField SFFloat  priority      0
  exposedField SFNode   source        NULL
  field        SFBool   spatialize    TRUE
}
```

The Sound node describes the positioning and spatial presentation of a sound in a VRML scene. The sound may be located at a point and emit sound in a spherical or ellipsoid pattern, in the local coordinate system. The ellipsoid is pointed in a particular direction and may be shaped to provide more or less directional focus from the location of the sound. The sound node may also be used to describe an ambient sound which tapers off at a specified distance from the sound node.

The Sound node also enables ambient background sound to be created by setting of the maxFront and maxBack to the radius of the area for the ambient noise. If ambient noise is required for the whole scene then these values should be set to at least cover the distance from the location to the farthest point in scene from that point (including effects of transforms).

The *source* field specifies the sound source for the sound node. If there is no source specified the Sound will emit no audio. The source field shall specify either an AudioClip or a MovieTexture node. Furthermore, the MovieTexture node must refer to a movie format that supports sound (e.g. MPEG1-Systems [MPEG]).

The *intensity* field adjusts the volume of each sound source; The *intensity* is an SFFloat that ranges from 0.0 to 1.0. An *intensity* of 0 is silence, and an *intensity* of 1 is the full volume of the sound in the sample or the full volume of the MIDI clip.

The *priority* field gives the author some control over which sounds the browser will choose to play when there are more sounds active than sound channels available. The *priority* varies between 0.0 and 1.0, with 1.0 being the highest priority. For most applications priority 0.0 should be used for a normal sound and 1.0 should be used only for special event or cue sounds (usually of short duration) that the author wants the user to hear even if they are farther away and perhaps of lower intensity than some other ongoing sounds. Browsers should make as many sound channels available to the scene as is efficiently possible.

If the browser does not have enough sound channels to play all of the currently active sounds, it is recommended that the browser sort the active sounds into an ordered list using the following sort keys:

1. decreasing *priority*;
2. for sounds with *priority* > 0.5, increasing (now-*startTime*)
3. decreasing *intensity* at viewer location ((*intensity*/distance)**2);

where now represents the current time, and *startTime* is the *startTime* field of the audio source node specified in the *source* field.

It is important that sort key #2 be used for the high priority (event and cue) sounds so that new cues will be heard even when the channels are "full" of currently active high priority sounds. Sort key #2 should not be used for normal priority sounds so selection among them will be based on sort key #3 - intensity and distance from the viewer.

The browser should play as many sounds from the beginning of this sorted list as it has available channels. On most systems the number of concurrent sound channels is distinct from the number of concurrent MIDI streams. On these systems the browser may maintain separate ordered lists for sampled sounds and MIDI streams.

A sound's *location* in the scene graph determines its spatial location (the sound's location is transformed by the current transformation) and whether or not it can be heard. A sound can only be heard while it is part of the traversed scene; sound nodes that are descended from LOD, Switch, or any grouping or prototype node that disables traversal (i.e. drawing) of its children will not be audible unless they are traversed. If a sound is silenced for a time under a Switch or LOD node, and later it becomes part of the traversal again, the sound picks up where it would have been had it been playing continuously.

Around the *location* of the emitter, *minFront* and *minBack* determine the extent of the full intensity region in front of and behind the sound. If the location of the sound is taken as a focus of an ellipsoid, the *minBack* and *minFront* values, in combination with the *direction* vector determine the two foci of an ellipsoid bounding the ambient region of the sound. Similarly, *maxFront* and *maxBack* determine the limits of audibility in front of and behind the sound; they describe a second, outer ellipsoid. If *minFront* equals *minBack* and *maxFront* equals *maxBack*, the sound is omni-directional, the direction vector is ignored, and the min and max ellipsoids become spheres centered around the sound node. The fields *minFront*, *maxFront*, *minBack*, and *maxBack* are scaled by the parent transformations - these values must be >= 0.0.

The inner ellipsoid defines a space of full intensity for the sound. Within that space the sound will play at the intensity specified in the sound node. The outer ellipsoid determines the maximum extent of the sound. Outside that space, the sound cannot be heard at all. In between the two ellipsoids, the intensity drops off proportionally with inverse square of the distance. With this model, a Sound usually will have smooth changes in intensity over the entire extent is which it can be heard. However, if at any point the maximum is the same as or inside the minimum, the sound is cut off immediately at the edge of the minimum ellipsoid.

The ideal implementation of the sound attenuation between the inner and outer ellipsoids is an inverse power dropoff. A reasonable approximation to this ideal model is a linear dropoff in decibel value. Since an inverse power dropoff never actually reaches zero, it is necessary to select an appropriate cutoff value for the outer ellipsoid so that the outer ellipsoid contains the space in which the sound is truly audible and excludes space where it would be negligible. Keeping the outer ellipsoid as small as possible will help limit resources used by nearly inaudible sounds. Experimentation suggests that a 20dB dropoff from the maximum intensity is a reasonable cutoff value that makes the bounding volume (the outer ellipsoid) contain the truly audible range of the sound. Since actual physical sound dropoff in an anechoic environment follows the inverse square law, using this algorithm it is possible to mimic real-world sound attenuation by making the maximum ellipsoid ten times larger than the minimum ellipsoid. This will yield inverse square dropoff between them.

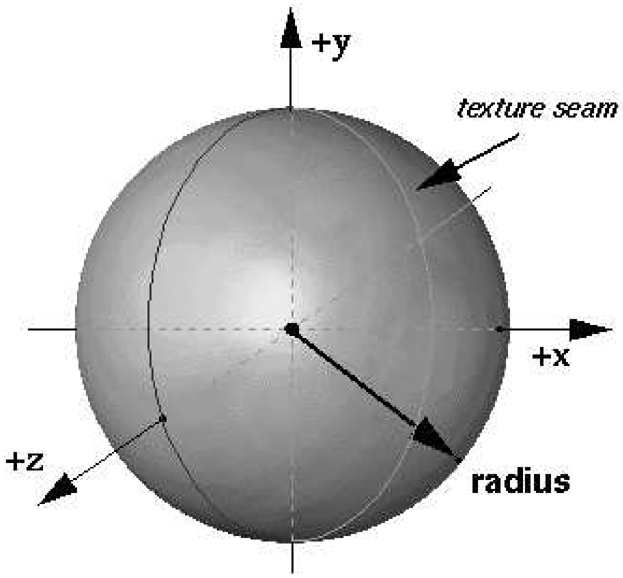
Browsers should support spatial localization of sound as well as their underlying sound libraries will allow. The *spatialize* field is used to indicate to browsers that they should try to locate this sound. If the *spatialize* field is TRUE, the sound should be treated as a monaural sound coming from a single point. A simple spatialization mechanism just places the sound properly in the pan of the stereo (or multichannel) sound output. Sounds are faded out over distance as described above. Browsers may use more elaborate sound spatialization algorithms if they wish.

Authors can create ambient sounds by setting the *spatialize* field to FALSE. In that case, stereo and multichannel sounds should be played using their normal separate channels. The distance to the sound and the minimum and maximum ellipsoids (discussed above) should affect the intensity in the normal way. Authors can create ambient sound over the entire scene by setting the *minFront* and *minBack* to the maximum extents of the scene.

Sphere

```
Sphere {
  field SFFloat radius  1
}
```

The Sphere node specifies a sphere centered at (0, 0, 0) in the local coordinate system. The *radius* field specifies the radius of the sphere and must be >= 0.0.



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 where the YZ plane intersects the sphere. TextureTransform affects the texture coordinates of the Sphere.

The Sphere geometry is considered to be solid and thus requires outside faces only. When viewed from the inside the results are undefined.

SphereSensor

```
SphereSensor {
  exposedField SFBool     autoOffset        TRUE
```

```
  exposedField SFBool     enabled          TRUE
  exposedField SFRotation offset           0 1 0 0
  eventOut     SFBool     isActive
  eventOut     SFRotation rotation_changed
  eventOut     SFVec3f    trackPoint_changed
}
```

The SphereSensor maps pointing device (e.g. mouse or wand) motion into spherical rotation about the center of its local space. SphereSensor uses the descendant geometry of its parent node to determine if a hit occurs. The feel of the rotation is as if you were rolling a ball.

The *enabled* exposed field enables and disables the SphereSensor - if TRUE, the sensor reacts appropriately to user events, if FALSE, the sensor does not track user input or send output events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event the sensor is enabled and ready for user activation.

The SphereSensor generates events if the pointing device is activated while over any descendant geometry nodes of its parent group and then moved while activated. Typically, the pointing device is a 2D device such as a mouse. The pointing device is considered to be moving within a plane at a fixed distance from the viewer and perpendicular to the line of sight; this establishes a set of 3D coordinates for the pointer. If a 3D pointer is in use, then the sensor generates events only when the pointer is within the user's field of view. In either case, the pointing device is considered to "pass over" geometry when that geometry is intersected by a line extending from the viewer and passing through the pointer's 3D coordinates. If multiple sensors' geometry intersect this line (hereafter called the bearing), only the nearest will be eligible to generate events.

Upon activation of the pointing device (e.g. mouse button down) over the sensor's geometry an *isActive* TRUE event is sent. The vector defined by the initial point of intersection on the SphereSensor's geometry and the local origin determines the radius of the sphere used to map subsequent pointing device motion while dragging. The virtual sphere defined by this radius and the local origin at the time of activation are used to interpret subsequent pointing device motion and is not affected by any changes to the sensor's coordinate system while the sensor is active. For each position of the bearing, a *rotation_changed* event is output which corresponds to a relative rotation from the original intersection, plus the *offset* value. The sign of the rotation is defined by the local coordinate system of the sensor. *trackPoint_changed* events reflect the unclamped drag position on the surface of this sphere. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last rotation value and an *offset_changed* event is generated. See "*Concepts - Drag Sensors"* for more details.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it releases and generates an *isActive* FALSE event (other pointing device sensors cannot generate events during this time). Motion of the pointing device while *isActive* is TRUE is referred to as a "drag". If a 2D pointing device is in use, *isActive* events will typically reflect the state of the primary button associated with the device (i.e. *isActive* is TRUE when the primary button is pressed and FALSE when released). If a 3D pointing device (e.g. wand) is in use, *isActive* events will typically reflect whether the pointer is within or in contact with the sensor's geometry.

While the pointing device is activated, *trackPoint_changed* and *rotation_changed* events are output. *trackPoint_changed* events represent the unclamped intersection points on the surface of the invisible sphere. If the pointing device is dragged off the sphere while activated, browsers may interpret this in several ways (e.g. clamp all values to the sphere, continue to rotate as the point is dragged away from the sphere, etc.). Each movement of the pointing device, while *isActive* is TRUE, generates *trackPoint_changed* and *rotation_changed* events.

See "*Concepts - Pointing Device Sensors and Drag Sensors*" for more details.

SpotLight

```
SpotLight {
  exposedField SFFloat ambientIntensity  0
  exposedField SFVec3f attenuation       1 0 0
  exposedField SFFloat beamWidth         1.570796
  exposedField SFColor color             1 1 1
  exposedField SFFloat cutOffAngle       0.785398
  exposedField SFVec3f direction         0 0 -1
  exposedField SFFloat intensity         1
  exposedField SFVec3f location          0 0 0
  exposedField SFBool  on                TRUE
  exposedField SFFloat radius            100
}
```

The SpotLight node defines a light source that emits light from a specific point along a specific direction vector and constrained within a solid angle. Spotlights may illuminate geometry nodes that respond to light sources and intersect the solid angle. Spotlights are specified in their local coordinate system and are affected by parent transformations.

See "*Concepts - Light Sources*" for a detailed description of *ambientIntensity, color*, *intensity*, and VRML's lighting equations. See "*Concepts - Lighting Model*" for a detailed description of the VRML lighting equations.

The *location* field specifies a translation offset of the center point of the light source from the light's local coordinate system origin. This point is the apex of the solid angle which bounds light emission from the given light source. The *direction* field specifies the direction vector of the light's central axis defined in its own local coordinate system. The *on* field specifies whether the light source emits light--if TRUE, then the light source is emitting light and may illuminate geometry in the scene, if FALSE it does not emit light and does not illuminate any geometry. The *radius* field specifies the radial extent of the solid angle and the maximum distance from *location* than may be illuminated by the light source - the light source does not emit light outside this radius. The *radius* must be >= 0.0.

The *cutOffAngle* field specifies the outer bound of the solid angle. The light source does not emit light outside of this solid angle. The *beamWidth* field specifies an inner solid angle in which the light source emits light at uniform full intensity. The light source's emission intensity drops off from the inner solid angle (*beamWidth*) to the outer solid angle (*cutOffAngle*). The drop off function from the inner angle to the outer angle is a cosine raised to a power function:

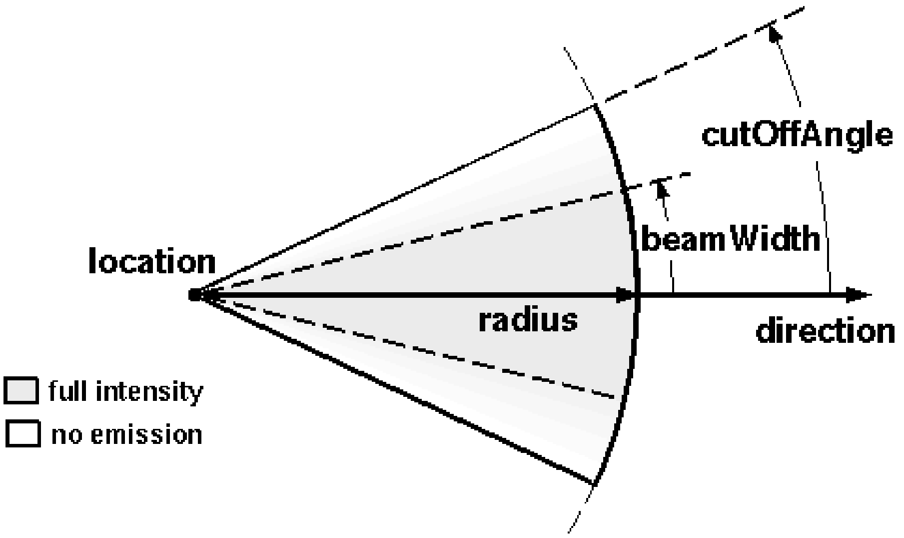
```
intensity(angle) = intensity * (cosine(angle) ** exponent)

where exponent = 0.5*log(0.5)/log(cos(beamWidth)),
      intensity is the SpotLight's field value,
```

```
intensity(angle) is the light intensity at an arbitrary
    angle from the direction vector,
and angle ranges from 0.0 at central axis to cutOffAngle.
```

If *beamWidth* > *cutOffAngle*, then *beamWidth* is assumed to be equal to *cutOffAngle* and the light source emits full intensity within the entire solid angle defined by *cutOffAngle*. Both *beamWidth* and *cutOffAngle* must be greater than 0.0 and less than or equal to PI/2. See figure below for an illustration of the SpotLight's field semantics (note: this example uses the default attenuation).

The light's illumination falls off with distance as specified by three *attenuation* coefficients. The attenuation factor is *1/(attenuation[0] + attenuation[1]*r + attenuation[2]*r^2)*, where *r* is the distance of the light to the surface being illuminated. The default is no attenuation. An *attenuation* value of **0 0 0** is identical to **1 0 0**. Attenuation values must be >= 0.0.



Switch

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

The Switch grouping node traverses zero or one of the nodes specified in the *choice* field.

See the "*Concepts - Grouping and Children Nodes*" section which describes "*children nodes*" for a details on the types of nodes that are legal values for *choice*.

The *whichChoice* field specifies the index of the child to traverse, where the first child has index 0. If *whichChoice* is less than zero or greater than the number of nodes in the *choice* field then nothing is chosen.

Note that all nodes under a Switch continue to receive and send events (i.e.routes) regardless of the value of *whichChoice*. For example, if an active TimeSensor is contained within an inactive choice of an Switch, the TimeSensor sends events regardless of the Switch's state.

Text

```
Text {
  exposedField  MFString string    []
  exposedField  SFNode   fontStyle NULL
  exposedField  MFFloat  length    []
  exposedField  SFFloat  maxExtent 0.0
}
```

The Text node specifies a two-sided, flat text string object positioned in the X-Y plane of the local coordinate system based on values defined in the fontStyle field (see FontStyle node). Text nodes may contain multiple text strings specified using the UTF-8 encoding as specified by the ISO 10646-1:1993 standard (http://www.iso.ch/cate/d18741.html). Due to the drastic changes in Korean Jamo language, the character set of the UTF-8 will be based on ISO 10646-1:1993 plus pDAM 1 - 5 (including the Korean changes). The text strings are stored in visual order.

The text strings are contained in the *string* field. The *fontStyle* field contains one FontStyle node that specifies the font size, font family and style, direction of the text strings, and any specific language rendering techniques that must be used for the text.

The *maxExtent* field limits and scales all of the text strings if the length of the maximum string is longer than the maximum extent, as measured in the local coordinate space. If the text string with the maximum length is shorter than the *maxExtent*, then there is no scaling. The maximum extent is measured horizontally for horizontal text (FontStyle node: *horizontal*=TRUE) and vertically for vertical text (FontStyle node: *horizontal*=FALSE). The *maxExtent* field must be >= 0.0.

The *length* field contains an MFFloat value that specifies the length of each text string in the local coordinate space. If the string is too short, it is stretched (either by scaling the text or by adding space between the characters). If the string is too long, it is compressed (either by scaling the text or by subtracting space between the characters). If a length value is missing--for example, if there are four strings but only three length values--the missing values are considered to be 0.

For both the *maxExtent* and *length* fields, specifying a value of 0 indicates to allow the string to be any length.

Textures are applied to 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, and T increases up.

ISO 10646-1:1993 Character Encodings

Characters in ISO 10646 are encoded in multiple octets. Code space is divided into four units, as follows:

```
+-------------+-------------+-----------+------------+
| Group-octet | Plane-octet | Row-octet | Cell-octet |
+-------------+-------------+-----------+------------+
```

The ISO 10646-1:1993 allows two basic forms for characters:

1. UCS-2 (Universal Coded Character Set-2). Also known as the Basic Multilingual Plane (BMP). Characters are encoded in the lower two octets (row and cell). Predictions are that this will be the most commonly used form of 10646.
2. UCS-4 (Universal Coded Character Set-4). Characters are encoded in the full four octets.

In addition, three transformation formats (UCS Transformation Format (UTF) are accepted: UTF-7, UTF-8, and UTF-16. Each represents the nature of the transformation - 7-bit, 8-bit, and 16-bit. The UTF-7 and UTF-16 can be referenced in the Unicode Standard 2.0 book.

The UTF-8 maintains transparency for all of the ASCII code values (0...127). It allows ASCII text (0x0..0x7F) to appear without any changes and encodes all characters from 0x80.. 0x7FFFFFFF into a series of six or fewer bytes.

If the most significant bit of the first character is 0, then the remaining seven bits are interpreted as an ASCII character. Otherwise, the number of leading 1 bits will indicate the number of bytes following. There is always a o bit between the count bits and any data.

First byte could be one of the following. The X indicates bits available to encode the character.

```
 0XXXXXXX only one byte   0..0x7F (ASCII)
 110XXXXX two bytes       Maximum character value is 0x7FF
 1110XXXX three bytes     Maximum character value is 0xFFFF
 11110XXX four bytes      Maximum character value is 0x1FFFFF
 111110XX five bytes      Maximum character value is 0x3FFFFFF
 1111110X six bytes       Maximum character value is 0x7FFFFFFF
```

All following bytes have this format: 10XXXXXX

A two byte example. The symbol for a register trade mark is "circled R registered sign" or 174 in ISO/Latin-1 (8859/1). It is encoded as 0x00AE in UCS-2 of the ISO 10646. In UTF-8 it is has the following two byte encoding 0xC2, 0xAE.

See "*Concepts - Lighting Model*" for details on VRML lighting equations and how Appearance, Material and textures interact with lighting.

The Text node does not perform collision detection.

TextureCoordinate

```
TextureCoordinate {
  exposedField MFVec2f point  []
```

```
}
```

The TextureCoordinate node specifies a set of 2D texture coordinates used by vertex-based geometry nodes (e.g. IndexedFaceSet and ElevationGrid) to map from textures to the vertices. Textures are two dimensional color functions that given an S and T pair return a color value. Texture maps parameter values range from 0.0 to 1.0 in S and T. However, TextureCoordinate values, specified by the *point* field, can range from -*infinity* to +*infinity*. Texture coordinates identify a location (and thus a color value) in the texture map. The horizontal coordinate, S, is specified first, followed by the vertical coordinate, T.

If the texture map is repeated in a given direction (S or T), then a texture coordinate C is mapped into a texture map that has N pixels in the given direction as follows:

```
        Location = (C - floor(C)) *
```

If the texture is not repeated:

```
        Location = (C > 1.0 ? 1.0 : (C < 0.0 ? 0.0 : C)) * N
```

See texture nodes for details on repeating textures (ImageTexture, MovieTexture, PixelTexture).

TextureTransform

```
TextureTransform {
  exposedField SFVec2f center      0 0
  exposedField SFFloat rotation    0
  exposedField SFVec2f scale       1 1
  exposedField SFVec2f translation 0 0
}
```

The TextureTransform node defines a 2D transformation that is applied to texture coordinates (see TextureCoordinate). This node affects the way textures are applied to the surface of geometry. The transformation consists of (in order) a non-uniform scale about an arbitrary center point, a rotation about the center point, and a translation. This allows for changes to the size, orientation, and position of textures on shapes. Note that these changes appear reversed when when viewed in the surface of geometry. For example, a *scale* value of **2 2** will scale the texture coordinates and have the net effect of shrinking the texture size by a factor of 2 (texture coordinates are twice as large and thus cause the texture to repeat). A translation of **0.5 0.0** translates the texture coordinates +.5 units along the S-axis and has the net effect of translating the texture -0.5 along the S-axis on the geometry’s surface. A rotation of PI/2 of the texture coordinates results in a -PI/2 rotation of the texture on the geometry.

The *center* field specifies a translation offset in texture coordinate space about which the *rotation* and *scale* fields are applied. The *scale* field specifies a scaling factor in S and T of the texture coordinates about the *center* point - *scale* values must be >= 0.0. The *rotation* field specifies a rotation in radians of the texture coordinates about the *center* point after the scale has taken place. The *translation* field specifies a translation of the texture coordinates.

Given a 2-dimensional texture coordinate **T** and a TextureTransform node, **T** is transformed into point

T′ by a series of intermediate transformations. In matrix-transformation notation, where C (center), T (translation), R (rotation), and S (scale) are the equivalent transformation matrices,

```
  T' = TT x C x R x S x -TC x T  (where T is a column vector)
```

Note that TextureTransforms cannot combine or accumulate.

TimeSensor

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

TimeSensors generate events as time passes. TimeSensors can be used to drive continuous simulations and animations, periodic activities (e.g., one per minute), and/or single occurrence events such as an alarm clock. TimeSensor discrete eventOuts include: *isActive*, which becomes TRUE when the TimeSensor begins running, and FALSE when it stops running, and *cycleTime*, a time event at *startTime* and at the beginning of each new cycle (useful for synchronization with other time-based objects). The remaining outputs generate continuous events and consist of *fraction_changed*, which is an SFFloat in the closed interval [0,1] representing the completed fraction of the current cycle, and *time*, an SFTime event specifying the absolute time for a given simulation tick.

If the *enabled* exposedField is TRUE, the TimeSensor is enabled and may be running. If a *set_enabled* FALSE event is received while the TimeSensor is running, then the sensor should evaluate and send all relevant outputs, send a FALSE value for *isActive,* and disable itself. However, events on the exposedFields of the TimeSensor (such as *set_startTime)* are processed and their corresponding eventOuts (*startTime_changed)* are sent regardless of the state of *enabled*. The remaining discussion assumes *enabled* is TRUE.

The *loop, startTime,* and *stopTime* exposedFields, and the *isActive* eventOut and their affects on the TimeSensor node, are discussed in detail in the "*Concepts - Time Dependent Nodes*" section. The "*cycle"* of an TimeSensor lasts for *cycleInterval* seconds. The value of *cycleInterval* must be greater than 0 (a value less than or equal to 0 produces undefined results). Because the TimeSensor is more complex than the abstract TimeDep node and generates continuous eventOuts, some of the information in the "Time Dependent Nodes" section is repeated here.

A *cycleTime* eventOut can be used for synchronization purposes, e.g., sound with animation. The value of a *cycleTime* eventOut will be equal to the time at the beginning of the current cycle. A *cycleTime* eventOut is generated at the beginning of every cycle, including the cycle starting at *startTime*. The first *cycleTime* eventOut for a TimeSensor node can be used as an alarm (single pulse at a specified time).

When a TimeSensor becomes active it will generate an *isActive* = TRUE event and begin generating *time, fraction_changed,* and *cycleTime* events, which may be routed to other nodes to drive animation or simulated behaviors - (see below for behavior at read time). The *time* event outputs the absolute time for a given tick of the TimeSensor (time fields and events represent the number of seconds since midnight GMT January 1, 1970). *fraction_changed* events output a floating point value in the closed interval [0, 1], where 0 corresponds to *startTime* and 1 corresponds to *startTime* + N**cycleInterval,* where N = 1, 2, ... That is, the *time* and *fraction_changed* eventOuts can be computed as:

```
time = now
f = fmod(now - startTime, cycleInterval)
if (f == 0.0 && now > startTime)
    fraction_changed = 1.0
else
    fraction_changed = f / cycleInterval
```

A TimeSensor can be set up to be active at read time by specifying *loop* TRUE (not the default) and *stopTime* <= *startTime* (satisfied by the default values). The *time* events output absolute times for each tick of the TimeSensor -- times must start at *startTime* and end with either *startTime+cycleInterval*, *stopTime*, or loop forever depending on the values of the other fields. An active TimeSensor must stop at the first simulation tick when time `now` >= *stopTime* > *startTime*.

No guarantees are made with respect to how often a TimeSensor will generate time events, but a TimeSensor should generate events at least at every simulation tick. TimeSensors are guaranteed to generate final *time* and *fraction_changed* events. If loop is FALSE, the final *time* event will be generated with a value of (*startTime+cycleInterval*) or *stopTime (*if *stopTime* > *startTime),* whichever value is less. If *loop* is TRUE at the completion of every cycle, then the final event will be generated as evaluated at *stopTime* (if *stopTime* > *startTime)* or never.

An active TimeSensor ignores *set_cycleInterval*, and *set_startTime* events. An active TimeSensor also ignores *set_stopTime* events for *set_stopTime* < *startTime*. For example, if a *set_startTime* event is received while a TimeSensor is active, then that *set_startTime* event is ignored (the *startTime* field is not changed, and a *startTime_changed* eventOut is not generated). If an active TimeSensor receives a *set_stopTime* event that is less than `now` and greater than or equal to *startTime*, it behaves as if the *stopTime* requested is `now` and sends the final events based on `now` (note that *stopTime* is set as specified in the eventIn).

TouchSensor

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

A TouchSensor tracks the location and state of the pointing device and detects when the user points at geometry contained by the TouchSensor's parent group. This sensor can be enabled or disabled by sending it an *enabled* event with a value of TRUE or FALSE. If the TouchSensor is disabled, it does not track user input or send output events.

The TouchSensor generates events as the pointing device "passes over" any geometry nodes that are descendants of the TouchSensor's parent group. Typically, the pointing device is a 2D device such as a mouse. In this case, the pointing device is considered to be moving within a plane a fixed distance from the viewer and perpendicular to the line of sight; this establishes a set of 3D coordinates for the pointer. If a 3D pointer is in use, then the TouchSensor generates events only when the pointer is within the user's field of view. In either case, the pointing device is considered to "pass over" geometry when that geometry is intersected by a line extending from the viewer and passing through the pointer's 3D coordinates. If multiple surfaces intersect this line (hereafter called the bearing), only the nearest will be eligible to generate events.

The *isOver* eventOut reflects the state of the pointing device with regard to whether it is over the TouchSensor's geometry or not. When the pointing device changes state from a position such that its bearing does not intersect any of the TouchSensor's geometry to one in which it does intersect geometry, an *isOver* TRUE event is generated. When the pointing device moves from a position such that its bearing intersects geometry to one in which it no longer intersects the geometry, or some other geometry is obstructing the TouchSensor's geometry, an *isOver* FALSE event is generated. These events are generated only when the pointing device has moved and changed 'over state; events are not generated if the geometry itself is animating and moving underneath the pointing device.

As the user moves the bearing over the TouchSensor's geometry, the point of intersection (if any) between the bearing and the geometry is determined. Each movement of the pointing device, while *isOver* is TRUE, generates *hitPoint_changed*, *hitNormal_changed*, and *hitTexCoord_changed* events. *hitPoint_changed* events contain the 3D point on the surface of the underlying geometry, given in the TouchSensor's coordinate system. *hitNormal_changed* events contain the surface normal vector at the hitPoint. *hitTexCoord_changed* events contain the texture coordinates of that surface at the hitPoint, which can be used to support the 3D equivalent of an image map.

If *isOver* is TRUE, the user may activate the pointing device to cause the TouchSensor to generate *isActive* events (e.g. press the primary mouse button). When the TouchSensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it releases and generates an *isActive* FALSE event (other pointing device sensors will not generate events during this time). Motion of the pointing device while *isActive* is TRUE is referred to as a "drag". If a 2D pointing device is in use, *isActive* events will typically reflect the state of the primary button associated with the device (i.e. *isActive* is TRUE when the primary button is pressed, and FALSE when not released). If a 3D pointing device is in use, *isActive* events will typically reflect whether the pointer is within or in contact with the TouchSensor's geometry.

The eventOut field *touchTime* is generated when all three of the following conditions are true:

- the pointing device was over the geometry when it was initially activated (*isActive* is TRUE),
- the pointing device is currently over the geometry (*isOver* is TRUE),
- and, the pointing device is deactivated (*isActive* FALSE event is also generated).

See "*Concepts - Pointing Device Sensors*" for more details.

Transform

```
Transform {
  eventIn      MFNode      addChildren
  eventIn      MFNode      removeChildren
  exposedField SFVec3f     center           0 0 0
  exposedField MFNode      children         []
  exposedField SFRotation  rotation         0 0 1  0
  exposedField SFVec3f     scale            1 1 1
  exposedField SFRotation  scaleOrientation 0 0 1  0
  exposedField SFVec3f     translation      0 0 0
  field        SFVec3f     bboxCenter       0 0 0
  field        SFVec3f     bboxSize         -1 -1 -1
}
```

A Transform is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its parents. See also "*Concepts - Coordinate Systems and Transformations*."

See the "*Concepts - Grouping and Children Nodes*" section for a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Transform's children. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, then the results are undefined. A default *bboxSize* value, (-1 -1 -1), implies that the bounding box is not specified and if needed must be calculated by the browser. See "*Concepts - Bounding Boxes*" for a description of the *bboxCenter* and *bboxSize* fields.

The *translation*, *rotation*, *scale*, *scaleOrientation* and *center* fields define 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 *center* field specifies a translation offset from the local coordinate system's origin, (0,0,0). The *rotation* field specifies a rotation of the coordinate system. The *scale* field specifies a non-uniform scale of the coordinate system - *scale* values must be >= 0.0. The *scaleOrientation* specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The *scaleOrientation* applies only to the scale operation. The *translation* field specifies a translation to the coordinate system.

Given a 3-dimensional point **P** and Transform node, **P** is transformed into point **P'** in its parent's coordinate system by a series of intermediate transformations. In matrix-transformation notation, where C (center), SR (scaleOrientation), T (translation), R (rotation), and S (scale) are the equivalent transformation matrices,

```
P' = T x C x R x SR x S x -SR x -TC x P  (where P is a column vector)
```

The Transform node:

```
Transform {
    center           C
    rotation         R
    scale            S
```

```
    scaleOrientation SR
    translation      T
    children         [...]
}
```

is equivalent to the nested sequence of:

```
Transform { translation T
 Transform { translation C
  Transform { rotation R
   Transform { rotation SR
    Transform { scale S
     Transform { rotation -SR
      Transform { translation -C
              ...
}}}}}}}
```

Viewpoint

```
Viewpoint {
  eventIn      SFBool     set_bind
  exposedField SFFloat    fieldOfView    0.785398
  exposedField SFBool     jump           TRUE
  exposedField SFRotation orientation    0 0 1  0
  exposedField SFVec3f    position       0 0 10
  field        SFString   description    ""
  eventOut     SFTime     bindTime
  eventOut     SFBool     isBound
}
```

The Viewpoint node defines a specific location in a local coordinate system from which the user might view the scene. Viewpoints are "*Concepts - Bindable Children Nodes*" and thus there exists a Viewpoint stack in the browser in which the top-most Viewpoint on the stack is the currently active Viewpoint. If a TRUE value is sent to the *set_bind* eventIn of a Viewpoint, it is moved to the top of the Viewpoint stack and thus activated. When a Viewpoint is at the top of the stack, the user's view is conceptually re-parented as a child of the Viewpoint. All subsequent changes to the Viewpoint's coordinate system change the user's view (e.g. changes to any parent transformation nodes or to the Viewpoint's position or orientation fields). Sending a *set_bind* FALSE event removes the Viewpoint from the stack and results in *isBound* FALSE and *bindTime* events. If the popped Viewpoint is at the top of the viewpoint stack the user's view is re-parented to the next entry in the stack. See "*Concepts - Bindable Children Nodes*" for more details on the the binding stacks. When a Viewpoint is moved to the top of the stack, the existing top of stack Viewpoint sends an *isBound* FALSE event and is pushed onto the stack.

Viewpoints have the additional requirement from other binding nodes in that they store the relative transformation from the user view to the current Viewpoint when they are moved to the top of stack. This is needed by the *jump* field, described below.

An author can automatically move the user's view through the world by binding the user to a Viewpoint and then animating either the Viewpoint or the transformations above it. Browsers shall allow the user view to be navigated relative to the coordinate system defined by the Viewpoint (and the transformations above it), even if the Viewpoint or its parent transformations are being animated.

The *bindTime* eventOut sends the time at which the Viewpoint is bound or unbound. This can happen during loading, when a *set_bind* event is sent to the Viewpoint, or when the browser binds to the Viewpoint via its user interface (see below).

The *position* and *orientation* fields of the Viewpoint node specify relative locations in the local coordinate system. *Position* is relative to the coordinate system's origin (0,0,0), while *orientation* specifies a rotation relative to the default orientation; the default orientation has the user looking down the -Z axis with +X to the right and +Y straight up. Viewpoints are affected by the transformation hierarchy.

Navigation types (see NavigationInfo) that require a definition of a *down* vector (e.g. terrain following) shall use the negative Y-axis of the coordinate system of the currently bound Viewpoint. Likewise navigation types (see NavigationInfo) that require a definition of an *up* vector shall use the positive Y-axis of the coordinate system of the currently bound Viewpoint. Note that the *orientation* field of the Viewpoint does not affect the definition of the down or up vectors. This allows the author to separate the viewing direction from the gravity direction.

The *jump* field specifies whether the user's view 'jumps' (or animates) to the position and orientation of a bound Viewpoint. Regardless of the value of *jump* at bind time, the relative viewing transformation between the user's view and the current Viewpoint shall be stored with the current Viewpoint for later use when *un-jumping*. The following is a re-write of the general bind stack rules described in "*Concepts - Bindable Child Nodes, Bind Stack Behavior*" with additional rules regarding Viewpoints (in **bold**):

1. During read:
    - the first encountered Viewpoint is bound by pushing it to the top of the Viewpoint stack,
        - nodes contained within Inlines are not candidates for the first encountered Viewpoint,
        - the first node within a prototype is a valid candidate for the first encountered Viewpoint;
    - the first encountered Viewpoint sends an *isBound* TRUE event.
2. When a *set_bind* TRUE eventIn is received by a Viewpoint:
    - if it is not on the top of the stack:
        - **the relative transformation from the current top of stack Viewpoint to the user's view is stored with the current top of stack Viewpoint,**
        - the current top of stack node sends an *isBound* eventOut FALSE,
        - the new node is moved to the top of the stack and becomes the currently bound Viewpoint,
        - the new Viewpoint (top of stack) sends an *isBound* TRUE eventOut,
        - **if *jump* is TRUE for the new Viewpoint, then the user's view is 'jumped' (or animated) to match the values in the *position* and *orientation* fields of the new Viewpoint;**
    - else if the node is already at the top of the stack, then this event has no affect.
3. When a *set_bind* FALSE eventIn is received by a Viewpoint:
    - it is removed from the stack,
    - if it is on the top of the stack:
        - it sends an *isBound* eventOut FALSE,
        - the next node in the stack becomes the currently bound Viewpoint (i.e. pop) and issues

an *isBound* TRUE eventOut,

- **if its *jump* is TRUE the user's view is 'jumped' (or animated) to the *position* and *orientation* of the next Viewpoint in the stack with the stored relative transformation for with this next Viewpoint applied,**

4. If a *set_bind* FALSE eventIn is received by a node not in the stack, the event is ignored and *isBound* events are not sent.
5. When a node replaces another node at the top of the stack, the *isBound* TRUE and FALSE eventOuts from the two nodes are sent simultaneously (i.e. identical timestamps).
6. If a bound node is deleted then it behaves as if it received a *set_bind* FALSE event (see #3).

Note that the *jump* field may change after a Viewpoint is bound - the rules described above still apply. If *jump* was TRUE when the Viewpoint is bound, but changed to FALSE before the *set_bind* FALSE is sent, then the Viewpoint does not *un-jump* during unbind. If *jump* was FALSE when the Viewpoint is bound, but changed to TRUE before the *set_bind* FALSE is sent, then the Viewpoint does perform the *un-jump* during unbind.

The *fieldOfView* field specifies a preferred field of view from this viewpoint, in radians. A small field of view roughly corresponds to a telephoto lens; a large field of view roughly corresponds to a wide-angle lens. The field of view should be greater than zero and smaller than PI; the default value corresponds to a 45 degree field of view. The value of *fieldOfView* represents the maximum viewing angle in any direction axis of the view. For example, a browser with a rectangular viewing projection shall use an angle of *fieldOfView* for the larger direction (depending on aspect ratio) and *fieldOfView* times *aspect ratio* in the smaller direction. If the aspect ratio is 2x1 (i.e. horizontal twice the vertical) and the fieldOfView is 1.0, then the horizontal viewing angle would be 1.0 and the vertical viewing angle would be 0.5. *fieldOfView* is a hint to the browser and may be ignored.

The *description* field identifies Viewpoints that are recommended to be publicly accessible through the browser's user interface (e.g. Viewpoints menu). The string in the *description* field should be displayed if this functionality is implemented. If *description* is empty, then the Viewpoint should not appear in any public user interface. It is recommended that the browser bind and move to a Viewpoint when its *description* is selected, either animating to the new position or jumping directly there. Once the new position is reached both the *isBound* and *bindTime* eventOuts are sent.

The URL syntax ".../scene.wrl#ViewpointName" specifies the user's initial view when entering "scene.wrl" to be the first Viewpoint in file "scene.wrl" that appears as "DEF ViewpointName Viewpoint { ... }" - this overrides the first Viewpoint in the file as the initial user view and receives a *set_bind* TRUE message. If the Viewpoint "ViewpointName" is not found, then assume that no Viewpoint was specified and use the first Viewpoint in the file. The URL syntax "#ViewpointName" specifies a view within the existing file. If this is loaded, then receives a *set_bind* TRUE message.

If a Viewpoint is bound (*set_bind*) and is the child of an LOD, Switch, or any node or prototype that disables its children, then the result is undefined. If a Viewpoint is bound that results in collision with geometry, then the browser performs its self-defined navigation adjustments as if the user navigated to this point (see Collision).

VisibilitySensor

```
VisibilitySensor {
  exposedField SFVec3f center   0 0 0
  exposedField SFBool  enabled  TRUE
  exposedField SFVec3f size     0 0 0
  eventOut     SFTime  enterTime
  eventOut     SFTime  exitTime
  eventOut     SFBool  isActive
}
```

The VisibilitySensor detects visibility changes of a rectangular box as the user navigates the world. VisibilitySensor is typically used to detect when the user can see a specific object or region in the scene, and to activate or deactivate some behavior or animation in order to attract the user or improve performance.

The *enabled* field enables and disables the VisibilitySensor. If *enabled* is set to FALSE, the VisibilitySensor does not send output events. If *enabled* is TRUE, then the VisibilitySensor detects changes to the visibility status of the box specified and sends events through the *isActive* eventOut. A TRUE event is output to *isActive* when any portion of the box impacts the rendered view, and a FALSE event is sent when the box has no effect on the view. Browsers shall guarantee that if *isActive* is FALSE that the box has absolutely no effect on the rendered view - browsers may error liberally when *isActive* is TRUE (e.g. maybe it does affect the rendering).

The exposed fields *center* and *size* specify the object space location of the box center and the extents of the box (i.e. width, height, and depth). The VisibilitySensor's box is effected by hierarchical transformations of its parents.

The *enterTime* event is generated whenever the *isActive* TRUE event is generated, and *exitTime* events are generated whenever *isActive* FALSE events are generated.

Each VisibilitySensor behaves independently of all other VisibilitySensors - every enabled VisibilitySensor that is affected by the user's movement receives and sends events, possibly resulting in multiple VisibilitySensors receiving and sending events simultaneously. Unlike TouchSensors, there is no notion of a Visibility Sensor lower in the scene graph "grabbing" events. Instanced (DEF/USE) VisibilitySensors use the union of all the boxes defined by their instances to check for enter and exit - an instanced VisibilitySensor will detect enter, motion, and exit for all instances of the box and send output events appropriately.

WorldInfo

```
WorldInfo {
  field MFString info  []
  field SFString title ""
}
```

The WorldInfo node contains information about the world. This node has no effect on the visual appearance or behavior of the world - it is strictly for documentation purposes. The *title* field is intended to store the name or title of the world so that browsers can present this to the user - for instance, in their window border. Any other information about the world can be stored in the *info* field - for instance, the scene author, copyright information, and public domain information.

The Virtual Reality Modeling Language Specification

6. Field and Event Reference

Version 2.0, ISO/IEC WD 14772

August 4, 1996

6.1 Introduction

This annex describes the syntax and general semantics of *fields* and *events,* the elemental data types used by VRML nodes to define objects (see "*Node Reference*"). Nodes are composed of fields and events (see "*Concepts - Nodes, Fields, and Events*"). The types defined in this annex are used by both fields and events.

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

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

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

6.2 SFBool

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

```
fooBool FALSE
```

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

The initial value of an SFBool eventOut is FALSE.

6.3 SFColor/MFColor

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

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

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

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

6.4 SFFloat/MFFloat

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

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

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

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

6.5 SFImage

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

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

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

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

For example,

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

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

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

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

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

6.6 SFInt32/MFInt32

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

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

is an MFInt32 field containing three values.

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

6.7 SFNode/MFNode

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

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

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

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

6.8 SFRotation/MFRotation

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

```
fooRot 0.0 1.0 0.0  3.14159265
```

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

6.9 SFString/MFString

The SFString and MFString fields and events contain strings formatted with the UTF-8 universal character set (ISO/IEC 10646-1:1993, http://www.iso.ch/cate/d18741.html). SFString specifies a single string, and the MFString specifies zero or more strings. Strings are written to file as a sequence of UTF-8 octets enclosed in double quotes (e.g. `"string"`).

Due to the drastic changes in Korean Jamo language, the character set of the UTF-8 will be based on ISO 10646-1:1993 plus pDAM 1 - 5 (including the Korean changes). The text strings are stored in visual order.

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

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

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

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

6.10 SFTime/MFTime

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

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

6.11 SFVec2f/MFVec2f

An SFVec2f field or event specifies a two-dimensional vector. An MFVec2f field or event specifies zero or more two-dimensional vectors. SFVec2fs and MFVec2fs are written to file as a pair of floating point values separated by whitespace. For example:

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

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

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

6.12 SFVec3f/MFVec3f

An SFVec3f field or event specifies a three-dimensional vector. An MFVec3f field or event specifies zero or more three-dimensional vectors. SFVec3fs and MFVec3fs are written to file as three floating point values separated by whitespace. For example:

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

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

The initial value of an SFVec3f eventOut is (0 0 0). The initial value of an MFVec3f eventOut is [ ].

Contact rikk@best.com , cmarrin@sgi.com, or gavin@acm.org with questions or comments.

This document's URL: http://vrml.sgi.com/moving-worlds/spec/part1/fieldsRef.html

The Virtual Reality Modeling Language

7. Conformance and Minimum Support Requirements

Version 2.0, ISO/IEC WD 14772

August 4, 1996

7.1 Introduction

7.1.1 Objectives

This clause provides rules for identifying conforming generators and interpreters of ISO/IEC 14772 along with specifications as to the minimum level of complexity which must be supported.

The primary objectives of these rules are:

1. to promote interoperability by eliminating arbitrary subsets of, or extensions to, ISO/IEC 14772;
2. to promote uniformity in the development of conformance tests;
3. to facilitate automated test generation.

7.1.2 Scope

This clause provides conformance criteria for metafiles, metafile generators, and metafile interpreters.

This clause addresses the VRML data stream and implementation requirements. Implementation requirements address the latitude allowed by VRML generators and interpreters. This clause does not directly address the environmental, performance, or resource requirements of the generator or interpreter.

This clause does not define the application requirements or dictate application functional content within a VRML file.

The scope of this clause is limited to rules for the open interchange of VRML content.

7.2 Conformance

7.2.1 Conformance of metafiles

Conformance of metafiles to ISO/IEC 14772 is defined in terms of the functionality and form specified in Part 1. In order to conform to ISO/IEC 14772, a metafile shall be a syntactically correct metafile.

A metafile is a syntactically correct version of ISO/IEC 14772 if the following conditions are met:

1. The metafile contains as its first element a VRML header comment node;
2. All nodes contained therein match the functional specification of the corresponding nodes of ISO/IEC 14772-1. The metafile shall obey the relationships defined in the formal grammar and all other syntactic requirements.
3. The sequence of nodes in the metafile obeys the relationships specified in ISO/IEC 14772-1 producing the structure specified in ISO/IEC 14772-1. For example, ...
4. No nodes appear in the metafile other than those specified in ISO/IEC 14772-1 unless required for the encoding technique. All nodes not defined in ISO/IEC 14772-1 are encoded using the PROTO or EXTERNPROTO nodes.
5. The metafile is encoded according to the rules in the standard clear text encoding in ISO/IEC 14772-1 or such other encodings that are standardized.

7.2.2 Conformance of metafile generators

Conformance of metafile generators is defined in terms of conformance to the functionality defined in ISO/IEC 14772-1. A metafile generator which conforms to ISO/IEC 14772 shall:

1. generate no syntax in violation of ISO/IEC 14772;
2. generate metafiles which conform to ISO/IEC 14772;
3. map the graphical characteristics of application pictures onto a set of VRML nodes which define those pictures within the latitude allowed in ISO/IEC 14772.

7.2.3 Conformance of metafile interpreters

Conformance of metafile interpreters is defined in terms of the functionality in ISO/IEC 14772. A metafile interpreter which conforms to ISO/IEC 14772 shall:

1. be able to read any metafile which conforms to ISO/IEC 14772;
2. render the graphical characteristics of the VRML nodes in any such metafile into a graphical

image or picture within the latitude defined in ISO/IEC 14772.

7.3 Minimum support requirements

7.3.1 Minimum support requirements for generators

There is no minimum complexity which must be supported by a conforming VRML generator except that the file must contain the required VRML header. Any compliant set of nodes may be generated of arbitrary complexity.

7.3.2 Minimum support requirements for interpreters

This subclause defines the minimum complexity which must be supported by a VRML interpreter. Interpreter implementations may choose to support greater limits but may not reduce the limits described in Table 7-1. When the metafile being interpreted contains nodes which exceed the latitude implemented by the interpreter, the interpreter will attempt to skip that node and continue at the next node. Where latitude is specified in this table for a particular node, full support is required for other aspects of that node.

Table 7-1: Minimum support criteria for VRML interpreters

Node	Minimum support
All groups	At least 512 children. Ignore bboxCenter and bboxSize
All interpolators	At least first 256 key-value pairs interpreted
All lights	At least 8 simultaneous lights.
All strings	At least 255 characters per string
All URL fields	At least 16 URL’s per field
Transformation stack	At least 32 levels in the transformation stack
Anchor	Ignore parameters. Ignore description
Appearance	Full support
AudioClip	Ignore description. At least 30 seconds duration. Wavefile in uncompressed PCM format

Background	Full support
Billboard	Full support except as for all groups
Box	Full support
Collision	Full support except as for all groups
Color	Full support
ColorInterpolator	Full support except as for all interpolators
Cone	Full support
Coordinate	At least first 16384 coordinates per coordinate node supported with indices to others ignored
CoordinateInterpolator	Full support except as for all interpolators
Cylinder	Full support
CylinderSensor	Full support except as for all interpolators
DirectionalLight	Global application of light source
ElevationGrid	At least 16384 heights per grid
Extrusion	At least 64 joints per extrusion. At least 1024 vertices in cross-section.
Fog	Full support
FontStyle	If non-Latin characters, family can be ignored.
Group	Full support except as for all groups
ImageTexture	Point sampling. At least JPEG and PNG formats
IndexedFaceSet	At least 1024 vertices per face. At least 1024 faces. Ignore ccw. Ignore convex. Ignore solid.
IndexedLineSet	At least 1024 vertices per polyline At least 1024 polylines per set
Inline	Full support except as for all groups

LOD	At least first 4 level/range combinations shall be interpreted
Material	Ignore ambient intensity Ignore specular colour Ignore emissive colour At least transparent and opaque with values less than 0.5 opaque
MovieTexture	At least one simultaneously active movie texture At least MPEG1-Systems and MPEG1-Video
NavigationInfo	Ignore avatarSize Ignore types other than "WALK", "FLY", "EXAMINE", and "NONE" Ignore visibilityLimit
Normal	At least first 16384 normals per normal node supported with indices to others ignored
NormalInterpolator	Full support except as for all interpolators
OrientationInterpolator	Full support except as for all interpolators
PixelTexture	At least 256256 image size
PlaneSensor	Full support
PointLight	Full support
PointSet	At least 4096 points per point set
PositionInterpolator	Full support except as for all interpolators
ProximitySensor	Full support
ScalarInterpolator	Full support except as for all interpolators
Script	At least 32 eventIns At least 32 fields At least 32 eventOuts

Shape	Full support
Sound	At least 3 simultaneously active sounds At least linear sound attenuation between inner and outer ellipsoids At least spatialization across the panorama being viewed At least 2 priorities
Sphere	Full support
SphereSensor	Full support
SpotLight	Beam width can be ignored
Switch	Full support except as for all groups
Text	At least UTF-8 character encoding transformation format
TextureCoordinate	At least first 16384 texture coordinates per texture coordinate node supported with indices to others ignored
TextureTransform	Full support
TimeSensor	Full support
TouchSensor	Full support
Transform	Full support except as for all groups
Viewpoint	Ignore fieldOfView Ignore description
VisibilitySensor	Full support
WorldInfo	Full support

The Virtual Reality Modeling Language Specification

Appendix A. Grammar Definition

Version 2.0, ISO/IEC 14772

August 4, 1996

This section provides a detailed description of the grammar for each node in VRML 2.0. There are four sections: Introduction, General, Nodes, and Fields.

A.1 Introduction

VRML grammar is ambiguous; semantic knowledge of the names and types of fields, eventIns, and eventOuts for each node type (either builtIn or user-defined using **PROTO** or **EXTERNROTO**) must be used during parsing so that the parser knows which field type is being parsed.

The ’#’ (0x23) character begins a comment wherever it appears outside of quoted SFString or MFString fields. The ’#’ character and all characters until the next carriage-return or newline make up the comment and are treated as whitespace.

The carriage return (0x0d), newline (0x0a), space (0x20), tab (0x09), and comma (0x2c) characters are whitespace characters wherever they appear outside of quoted SFString or MFString fields. Any number of whitespace characters and comments may be used to separate the syntactic entities of a VRML file.

Please see the Nodes Reference section of the Moving Worlds specification for a description of the allowed fields, eventIns and eventOuts for all pre-defined node types. Also note that some of the basic types that will typically be handled by a lexical analyzer (*sffloatValue*, *sftimeValue*, *sfint32Value*, and *sfstringValue*) have not been formally specified; please see the Fields Reference section of the spec for a more complete description of their syntax.

A.2 General

vrmlScene:
　　declarations
declarations:
　　declaration
　　declaration declarations
declaration:
　　nodeDeclaration
　　protoDeclaration
　　routeDeclaration
　　NULL
nodeDeclaration:
　　node
　　DEF *nodeNameId node*
　　USE *nodeNameId*
protoDeclaration:
　　proto
　　externproto
proto:
　　PROTO *nodeTypeId* **[** *interface_declarations* **] {** *vrmlScene* **}**
interfaceDeclarations:
　　interfaceDeclaration
　　interfaceDeclaration interfaceDeclarations
restrictedInterfaceDeclaration:
　　eventIn *fieldType eventInId*
　　eventOut *fieldType eventOutId*
　　field *fieldType fieldId fieldValue*
interfaceDeclaration:
　　restrictedInterfaceDeclaration
　　exposedField *fieldType fieldId fieldValue*
externproto:
　　EXTERNPROTO *nodeTypeId* **[** *externInterfaceDeclarations* **]** *mfstringValue*
externInterfaceDeclarations:
　　externInterfaceDeclaration
　　externInterfaceDeclaration externInterfaceDeclarations
externInterfaceDeclaration:
　　eventIn *fieldType eventInId*
　　eventOut *fieldType eventOutId*
　　field *fieldType fieldId*
　　exposedField *fieldType fieldId*
routeDeclaration:
　　ROUTE *nodeNameId* **.** *eventOutId* **TO** *nodeNameId* **.** *eventInId*

A.3 Nodes

node:
nodeTypeId **{** *nodeGuts* **}**
Script { *scriptGuts* **}**
nodeGuts:
nodeGut
nodeGut nodeGuts
scriptGuts:
scriptGut
scriptGut scriptGuts
scriptGut:
nodeGut
restrictedInterfaceDeclaration
eventIn *fieldType eventInId* **IS** *eventInId*
eventOut *fieldType eventOutId* **IS** *eventOutId*
field *fieldType fieldId* **IS** *fieldId*
nodeGut:
fieldId fieldValue
fieldId **IS** *fieldId*
eventInId **IS** *eventInId*
eventOutId **IS** *eventOutId*
routeDeclaration
protoDeclaration
nodeNameId:
Id
nodeTypeId:
Id
fieldId:
Id
eventInId:
Id
eventOutId:
Id
Id:
IdFirstChar
IdFirstChar IdRestChars
IdFirstChar:
Any ISO-10646 character encoded using UTF-8 except: 0x30-0x39, 0x0-0x20, 0x22, 0x23, 0x27, 0x2c, 0x2e, 0x5b, 0x5c, 0x5d, 0x7b, 0x7d.
IdRestChars:
Any number of ISO-10646 characters except: 0x0-0x20, 0x22, 0x23, 0x27, 0x2c, 0x2e, 0x5b, 0x5c, 0x5d, 0x7b, 0x7d.

A.4 Fields

fieldType:
MFColor
MFFloat
MFInt32
MFNode
MFRotation
MFString
MFVec2f
MFVec3f
SFBool
SFColor
SFFloat
SFImage
SFInt32
SFNode
SFRotation
SFString
SFTime
SFVec2f
SFVec3f

fieldValue:
sfboolValue
sfcolorValue
sffloatValue
sfimageValue
sfint32Value
sfnodeValue
sfrotationValue
sfstringValue
sftimeValue
sfvec2fValue
sfvec3fValue
mfcolorValue
mffloatValue
mfint32Value
mfnodeValue
mfrotationValue
mfstringValue
mfvec2fValue
mfvec3fValue

sfboolValue:
TRUE
FALSE

sfcolorValue:
float float float

sffloatValue:
... floating point number in ANSI C floating point format...
sfimageValue:
int32 int32 int32 int32s...
sfint32Value:
[0-9]+
0x*[0-9A-F]*+
sfnodeValue:
nodeDeclaration
NULL
sfrotationValue:
float float float float
sfstringValue:
".*" ... double-quotes must be \", backslashes must be \\...
sftimeValue:
... double-precision number in ANSI C floating point format...
sfvec2fValue:
float float
sfvec3fValue:
float float float
mfcolorValue:
sfcolorValue
[ ]
[*sfcolorValues* **]**
sfcolorValues:
sfcolorValue
sfcolorValue sfcolorValues
mffloatValue:
sffloatValue
[ ]
[*sffloatValues* **]**
sffloatValues:
sffloatValue
sffloatValue sffloatValues
mfint32Value:
sfint32Value
[ ]
[*sfint32Values* **]**
sfint32Values:
sfint32Value
sfint32Value sfint32Values
mfnodeValue:
nodeDeclaration
[ ]
[*nodeDeclarations* **]**
nodeDeclarations:
nodeDeclaration
nodeDeclaration nodeDeclarations

mfrotationValue:
  sfrotationValue
  [ ]
  [*sfrotationValues* **]**
sfrotationValues:
  sfrotationValue
  sfrotationValue sfrotationValues
mfstringValue:
  sfstringValue
  [ ]
  [*sfstringValues* **]**
sfstringValues:
  sfstringValue
  sfstringValue sfstringValues
mfvec2fValue:
  sfvec2fValue
  [ ]
  [*sfvec2fValues***]**
sfvec2fValues:
  sfvec2fValue
  sfvec2fValue sfvec2fValues
mfvec3fValue:
  sfvec3fValue
  [ ]
  [*sfvec3fValues* **]**
sfvec3fValues:
  sfvec3fValue
  sfvec3fValue sfvec3fValues

The Virtual Reality Modeling Language Specification

Appendix B. Examples

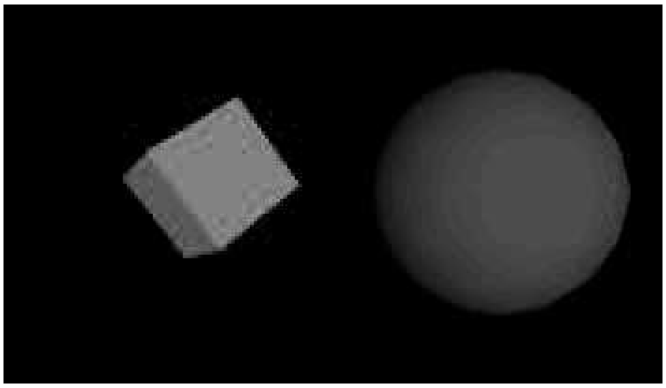
Version 2.0, ISO/IEC WD 14772

August 4, 1996

This appendix provides a variety of examples of VRML 2.0.

Simple example: "red sphere meets blue box"

This file contains a simple scene defining a view of a red sphere and a blue box, lit by a directional light:



```
#VRML V2.0 utf8
Transform {
  children [
    NavigationInfo { headlight FALSE } # We'll add our own light

    DirectionalLight {        # First child
        direction 0 0 -1      # Light illuminating the scene
    }

    Transform {               # Second child - a red sphere
      translation 3 0 1
```

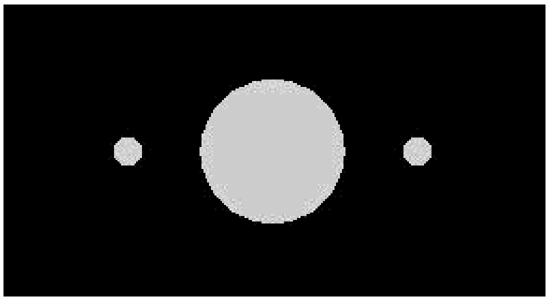
```
      children [
        Shape {
          geometry Sphere { radius 2.3 }
          appearance Appearance {
            material Material { diffuseColor 1 0 0 }   # Red
         }
        }
      ]
    }

    Transform {                # Third child - a blue box
      translation -2.4 .2 1
      rotation     0 1 1  .9
      children [
        Shape {
          geometry Box {}
          appearance Appearance {
            material Material { diffuseColor 0 0 1 }  # Blue
         }
        }
      ]
    }

  ] # end of children for world
}
```

Instancing (Sharing)

Reading the following file results in three spheres being drawn. The first sphere defines a unit sphere at the original named "Joe", the second sphere defines a smaller sphere translated along the +x axis, the third sphere is a reference to the second sphere and is translated along the -x axis. If any changes occur to the second sphere (e.g. radius changes), then the third sphere, (which is not really a reference to the second) will change too:



```
#VRML V2.0 utf8
Transform {
  children [
    DEF Joe Shape { geometry Sphere {} }
    Transform {
      translation 2 0 0
      children    DEF Joe Shape { geometry Sphere { radius .2 } }
    }
    Transform {
```

```
      translation -2 0 0
      children    USE Joe
    }

  ]
}
```

(Note that the spheres are unlit because no appearance was specified.)

Prototype example

A simple chair with variable colors for the leg and seat might be prototyped as:

```
#VRML V2.0 utf8

PROTO TwoColorStool [ field SFColor legColor  .8 .4 .7
                      field SFColor seatColor .6 .6 .1 ]
{
  Transform {
    children [
      Transform {   # stool seat
       translation 0 0.6 0
        children
          Shape {
            appearance Appearance {
              material Material { diffuseColor IS seatColor }
            }
            geometry Box { size 1.2 0.2 1.2 }
          }
      }

      Transform {   # first stool leg
       translation -.5 0 -.5
        children
          DEF Leg Shape {
            appearance Appearance {
              material Material { diffuseColor IS legColor }
            }
```

```
            geometry Cylinder { height 1 radius .1 }
          }
      }
      Transform {   # another stool leg
       translation .5 0 -.5
        children USE Leg
      }
      Transform {   # another stool leg
       translation -.5 0 .5
        children USE Leg
      }
      Transform {   # another stool leg
       translation .5 0 .5
        children USE Leg
      }
    ] # End of root Transform's children
  } # End of root Transform
} # End of prototype

# The prototype is now defined. Although it contains a number of nodes,
# only the legColor and seatColor fields are public. Instead of using the
# default legColor and seatColor, this instance of the stool has red legs
# and a green seat:


TwoColorStool {
  legColor 1 0 0 seatColor 0 1 0
}
NavigationInfo { type "EXAMINE" }      # Use the Examine viewer
```

Scripting Example

This Script node decides whether or not to open a bank vault given openVault and combinationEntered messages To do this it remembers whether or not the correct combination has been entered:

```
DEF OpenVault Script {
    # Declarations of what's in this Script node:
    eventIn SFTime  openVault
    eventIn SFBool  combinationEntered
    eventOut SFTime vaultUnlocked
    field SFBool    unlocked FALSE

    # Implementation of the logic:
    url "javascript:
        function combinationEntered(value) { unlocked = value; }
        function openVault(value) {
            if (unlocked) vaultUnlocked = value;
        }"
}
```

Note that the *openVault* eventIn and the *vaultUnlocked* eventOut are or type SFTime. This is so they can be wired directly to a TouchSensor and TimeSensor, respectively. The TimeSensor can output into an interpolator which performs an opening door animation.

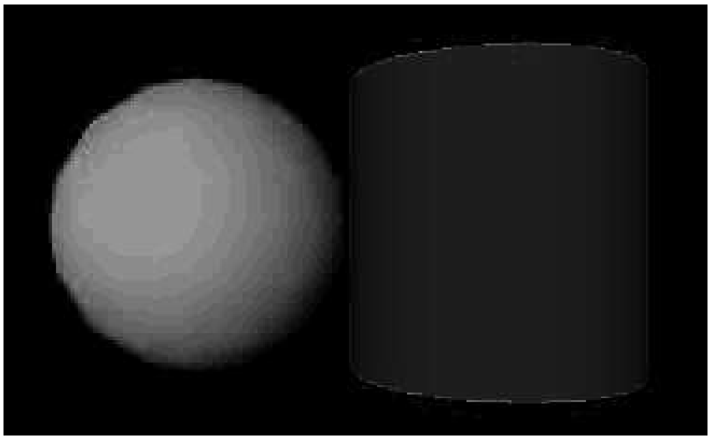
Geometric Properties

For example, the following IndexedFaceSet (contained in a Shape node) uses all four of the geometric property nodes to specify vertex coordinates, colors per vertex, normals per vertex, and texture coordinates per vertex (note that the material sets the overall transparency):

```
Shape {
  geometry IndexedFaceSet {
     coordIndex  [ 0, 1, 3, -1, 0, 2, 5, -1, ...]
     coord       Coordinate        { point [0.0 5.0 3.0, ...] }
     color       Color             { rgb [ 0.2 0.7 0.8, ...] }
     normal      Normal            { vector [0.0 1.0 0.0, ...] }
     texCoord    TextureCoordinate { point [0 1.0, ...] }
  }
  appearance Appearance { material Material { transparency 0.5 } }
}
```

Transforms and Leaves

This example has 2 parts. First is an example of a simple VRML 1.0 scene. It contains a red cone, a blue sphere, and a green cylinder with a hierarchical transformation structure. Next is the same example using the Moving Worlds Transforms and leaves syntax.



VRML 1.0

```
#VRML V1.0 ascii
Separator {
    Transform {
        translation 0 2 0
    }
    Material {
        diffuseColor 1 0 0
    }
    Cone { }
```

```
    Separator {
        Transform {
            scaleFactor 2 2 2
        }
        Material {
            diffuseColor 0 0 1
        }
        Sphere { }

        Transform {
            translation 2 0 0
        }
        Material {
            diffuseColor 0 1 0
        }
        Cylinder { }
    }
}
```

VRML 2.0

```
#VRML V2.0 utf8
Transform {
    translation 0 2 0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 1 0 0
                }
            }
            geometry Cone { }
        },

        Transform {
            scale 2 2 2
            children [
                Shape {
                    appearance Appearance {
                        material Material {
                            diffuseColor 0 0 1
                        }
                    }
                    geometry Sphere { }
                },

                Transform {
                    translation 2 0 0
                    children [
                        Shape {
                            appearance Appearance {
                                material Material {
                                    diffuseColor 0 1 0
                                }
                            }
                            geometry Cylinder { }
                        }
                    ]
                }
            ]
```

```
        }
    ]
}
```

Note that the default Viewpoint will not have the objects centered in the view as shown above.

Transform: VRML 1.0 vs. VRML 2.0

Here is an example that illustrates the order in which the elements of a Transform are applied:

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

is equivalent to the nested sequence of:

```
Transform { translation T1
 children [ Transform { translation T2
  children [ Transform { rotation R1
   children [ Transform { rotation R2
    children [ Transform { scale S
     children [ Transform { rotation -R2
      children [ Transform { translation -T2
              ...
       }
      ]}
     ]}
    ]}
   ]}
  ]}
 ]
}
```

Prototypes and Alternate Representations

Moving Worlds has the capability to define new nodes. VRML 1.0 had the ability to add nodes using the *fields* field and *isA* keyword. The prototype feature can duplicate all the features of the 1.0 node definition capabilities, as well as the alternate representation feature proposed in the VRML 1.1 draft spec. Take the example of a RefractiveMaterial. This is just like a Material node but adds an indexOfRefraction field. This field can be ignored if the browser cannot render refraction. In VRML 1.0 this would be written like this:

```
...
RefractiveMaterial {
```

```
    fields [ SFColor ambientColor,  MFColor diffuseColor,
             SFColor specularColor, MFColor emissiveColor,
             SFFloat shininess,     MFFloat transparency,
             SFFloat indexOfRefraction, MFString isA ]

    isA "Material"
}
```

If the browser had been hardcoded to understand a RefractiveMaterial the indexOfRefraction would be used, otherwise it would be ignored and RefractiveMaterial would behave just like a Material node.

In VRML 2.0 this is written like this:

```
...
PROTO RefractiveMaterial [
            field SFFloat ambientIntensity  0 0 0
            field MFColor diffuseColor      0.5 0.5 0.5
            field SFColor specularColor     0 0 0
            field MFColor emissiveColor     0 0 0
            field SFFloat shininess         0
            field MFFloat transparency      0 0 0
            field SFFloat indexOfRefraction 0.1 ]
{
    Material {
            ambientIntensity  IS ambientIntensity
            diffuseColor      IS diffuseColor
            specularColor     IS specularColor
            emissiveColor     IS emissiveColor
            shininess         IS shininess
            transparency      IS transparency
    }
}
```

While this is more wordy, notice that the default values were given in the prototype. These are different than the defaults for the standard Material. So this allows you to change defaults on a standard node. The EXTERNPROTO capability allows the use of alternative implementations of a node:

```
...
EXTERNPROTO RefractiveMaterial [
            field SFFloat ambientIntensity
            field MFColor diffuseColor
            field SFColor specularColor
            field MFColor emissiveColor
            field SFFloat shininess
            field MFFloat transparency
            field SFFloat indexOfRefraction ]

    http://www.myCompany.com/vrmlNodes/RefractiveMaterial.wrl,
    http://somewhere.else/MyRefractiveMaterial.wrl
```

This will choose from one of three possible sources of RefractiveMaterial. If the browser has this node hardcoded, it will be used. Otherwise the first URL will be requested and a prototype of the node will used from there. If that fails, the second will be tried.

Anchor

The *target* parameter can be used by the anchor node to send a request to load a URL into another frame:

```
Anchor {
  url "http://somehost/somefile.html"
  parameters [ "target=name_of_frame" ]
  ...
}
```

An Anchor may be used to bind the viewer to a particular *viewpoint* in a virtual world by specifying a URL ending with "#viewpointName", where "viewpointName" is the DEF name of a viewpoint defined in the world. For example:

```
Anchor {
  url "http://www.school.edu/vrml/someScene.wrl#OverView"
  children Shape { geometry Box {} }
}
```

specifies an anchor that puts the viewer in the "someScene" world bound to the viewpoint named "OverView" when the Box is chosen (note that "OverView" is the name of the viewpoint, not the value of the viewpoint's description field). If no world is specified, then the current scene is implied; for example:

```
Anchor {
  url "#Doorway"
  children Shape { Sphere {} }
}
```

binds you to the Viewpoint with the DEF name "Doorway" in the current scene.

Directional Light

A directional light source illuminates only the objects in its enclosing grouping node. The light illuminates everything within this coordinate system, including the objects that precede it in the scene graph--for example:

```
Transform {
  children [
    DEF UnlitShapeOne Shape { ... }
    DEF LitParent Transform {
      children [
        DEF LitShapeOne Shape { ... }
        DirectionalLight { .... } # lights the shapes under LitParent
        DEF LitShapeTwo Shape { ... }
      ]
    }
    DEF UnlitShapeTwo Shape { ... }
  ]
}
```

PointSet

This simple example defines a PointSet composed of 3 points. The first point is red (1 0 0), the second point is green (0 1 0), and the third point is blue (0 0 1). The second PointSet instances the Coordinate node defined in the first PointSet, but defines different colors:

```
Shape {
  geometry PointSet {
    coord DEF mypts Coordinate { point [ 0 0 0, 2 2 2, 3 3 3 ] }
    color Color { color [ 1 0 0, 0 1 0, 0 0 1 ] }
  }
}
Shape {
  geometry PointSet {
    coord USE mypts
    color Color { color [ .5 .5 0, 0 .5 .5, 1 1 1 ] }
  }
}
```

This simple example defines a PointSet composed of 3 points. The first point is red (1 0 0), the second point is green (0 1 0), and the third point is blue (0 0 1). The second PointSet instances the Coordinate node defined in the first PointSet, but defines different colors:

Level of Detail

The LOD node is typically used for switching between different versions of geometry at specified distances from the viewer. But if the range field is left at its default value the browser selects the most appropriate child from the list given. It can make this selection based on performance or perceived importance of the object. Children should be listed with most detailed version first just as for the normal case. This "performance LOD" feature can be combined with the normal LOD function to give the browser a selection of children from which to choose at each distance.

In this example, the browser is free to choose either a detailed or a less-detailed version of the object when the viewer is closer than 100 meters (as measured in the coordinate space of the LOD). The browser should display the less-detailed version of the object if the viewer is between 100 and 1,000 meters and should display nothing at all if the viewer is farther than 1,000 meters. Browsers should try to honor the hints given by authors, and authors should try to give browsers as much freedom as they can to choose levels of detail based on performance.

```
LOD {
  range [100, 1000]
  levels [
    LOD {
      levels [
        Transform { ... detailed version...  }
        DEF LoRes Transform { ... less detailed version... }
```

```
      ]
    }
    USE LoRes,
    Shape { } # Display nothing
  ]
}
```

For best results, specify ranges only where necessary, and nest LOD nodes with and without ranges.

Color Interpolator

This example interpolates from red to green to blue in a 10 second cycle:

```
DEF myColor ColorInterpolator {
  key        [   0.0,    0.5,    1.0 ]
  keyValue   [ 1 0 0,  0 1 0,  0 0 1 ] # red, green, blue
}

DEF myClock TimeSensor {
  cycleInterval 10.0      # 10 second animation
  loop          TRUE      # infinitely cycling animation
}

ROUTE myClock.fraction_changed TO myColor.set_fraction
```

TimeSensor

The TimeSensor is very flexible. Here are some of the many ways in which it can be used:

- a TimeSensor can be triggered to run continuously by setting *cycleInterval* > 0, and *loop* = TRUE, and then routing a time output from another node that triggers the loop (e.g. the *touchTime* eventOut of a TouchSensor can then be routed to the TimeSensor's *startTime* to start the TimeSensor running).
- a TimeSensor can be made to run continuously upon reading by setting *cycleInterval* > 0, *startTime* > 0, *stopTime* = 0, and *loop* = TRUE. (This use is not recommended.)

1. Animate a box when the user clicks on it:

```
DEF XForm Transform { children [
  Shape { geometry Box {} }
  DEF Clicker TouchSensor {}
  DEF TimeSource TimeSensor { cycleInterval 2.0 } # Run once for 2 sec.
  # Animate one full turn about Y axis:
  DEF Animation OrientationInterpolator {
       key      [ 0,      .33,       .66,        1.0 ]
       keyValue [ 0 1 0 0, 0 1 0 2.1, 0 1 0 4.2, 0 1 0 0 ]
  }
]}
ROUTE Clicker.touchTime TO TimeSource.startTime
```

```
ROUTE TimeSource.fraction_changed TO Animation.set_fraction
ROUTE Animation.value_changed TO XForm.rotation
```

2. Play Westminster Chimes once an hour:

```
#VRML V2.0 utf8

Group { children [
  DEF Hour TimeSensor {
    loop          TRUE
    cycleInterval 3600.0         # 60*60 seconds == 1 hour
  }
  Sound {
    source DEF Sounder AudioClip {
      url "http://...../westminster.mid" }
    }
  }
]}
ROUTE Hour.cycleTime TO Sounder.startTime
```

3. Make a grunting noise when the user runs into a wall:

```
DEF Walls Collision { children [
  Transform {
    #... geometry of walls...
  }
  Sound {
    source DEF Grunt AudioClip {
      url "http://...../grunt.wav"
    }
  }
]}
ROUTE Walls.collision TO Grunt.startTime
```

Shuttles and Pendulums

Shuttles and pendulums are great building blocks for composing interesting animations. This shuttle translates its children back and forth along the X axis, from -1 to 1. The pendulum rotates its children about the Y axis, from 0 to 3.14159 radians and back again.

```
PROTO Shuttle [
    exposedField SFBool enabled TRUE
    field SFFloat rate 1
    eventIn SFBool moveRight
    eventOut SFBool isAtLeft
    field MFNode children ]
{
    DEF F Transform { children IS children }
    DEF T TimeSensor {
        cycleInterval IS rate
        enabled IS enabled
    }
    DEF S Script {
        eventIn  SFBool  enabled IS set_enabled
        field    SFFloat rate IS rate
```

```
        eventIn  SFBool  moveRight IS moveRight
        eventIn  SFBool  isActive
        eventOut SFBool  isAtLeft IS isAtLeft
        eventOut SFTime  start
        eventOut SFTime  stop
        field    SFNode  timeSensor USE T

        url "javascript:
            // constructor: send initial isAtLeft eventOut
            function initialize() {
                isAtLeft = true;
            }

            function moveRight(move, ts) {
                if (move) {
                    // want to start move right
                    start = ts;
                    stop = ts + rate / 2;
                }
                else {
                    // want to start move left
                    start = ts - rate / 2;
                    stop = ts + rate / 2;
                }
            }

            function isActive(active) {
                if (!active) isAtLeft = !moveRight;
            }

            function set_enabled(value, ts) {
                if (value) {
                    // continue from where we left off
                    start = ts - (timeSensor.time - start);
                    stop  = ts - (timeSensor.time - stop);
                }
            }"
    }

    DEF I PositionInterpolator {
        keys [ 0, 0.5, 1 ]
        values [ -1 0 0, 1 0 0, -1 0 0 ]
    }

    ROUTE T.fraction_changed TO I.set_fraction
    ROUTE T.isActive TO S.isActive
    ROUTE I.value_changed TO F.set_translation
    ROUTE S.start TO T.set_startTime
    ROUTE S.stop TO T.set_stopTime
}


PROTO Pendulum [
    exposedField SFBool enabled TRUE
    field SFFloat rate 1
    field SFFloat maxAngle
    eventIn SFBool moveCCW
    eventOut SFBool isAtCW
    field MFNode children ]
{
    DEF F Transform { children IS children }
```

```
DEF T TimeSensor {
    cycleInterval IS rate
    enabled IS enabled
}
DEF S Script {
    eventIn  SFBool     enabled IS set_enabled
    field    SFFloat    rate IS rate
    field    SFFloat    maxAngle IS maxAngle
    eventIn  SFBool     moveCCW IS moveCCW
    eventIn  SFBool     isActive
    eventOut SFBool     isAtCW IS isAtCW
    eventOut SFTime     start
    eventOut SFTime     stop
    eventOut MFRotation rotation
    field    SFNode     timeSensor USE T

    url "javascript:
        function initialize() {
            // constructor:setup interpolator,
            // send initial isAtCW eventOut
            isAtCW = true;

            rot[0] = 0; rot[1] = 1; rot[2] = 0;
            rot[3] = 0;
            rotation[0] = rot;
            rotation[2] = rot;

            rot[3] = maxAngle;
            rotation[1] = rot;
        }

        function moveCCW(move, ts) {
            if (move) {
                // want to start CCW half (0.0 - 0.5) of move
                start = ts;
                stop = start + rate / 2;
            }
            else {
                // want to start CW half (0.5 - 1.0) of move
                start = ts - rate / 2;
                stop = ts + rate / 2;
            }
        }

        function isActive(active) {
            if (!active) isAtCW = !moveCCW;
        }

        function set_enabled(value, ts) {
            if (value) {
                // continue from where we left off
                start = ts - (timeSensor.time - start);
                stop  = ts - (timeSensor.time - stop);
            }
        }"
}
DEF I OrientationInterpolator {
    keys [ 0, 0.5, 1 ]
}
ROUTE T.fraction_changed TO I.set_fraction
ROUTE I.value_changed TO F.set_rotation
```

```
    ROUTE T.isActive TO S.isActive
    ROUTE S.start TO T.set_startTime
    ROUTE S.stop TO T.set_stopTime
    ROUTE S.rotation TO I.set_values
}
```

In use, the Shuttle can have its isAtRight output wired to its moveLeft input to give a continuous shuttle. The Pendulum can have its isAtCCW output wired to its moveCW input to give a continuous Pendulum effect.

Robot

Robots are very popular in in VRML discussion groups. Here's a simple implementation of one. This robot has very simple body parts: a cube for his head, a sphere for his body and cylinders for arms (he hovers so he has no feet!). He is something of a sentry - he walks forward, turns around, and walks back. He does this whenever you are near. This makes use of the Shuttle and Pendulum above.

```
DEF Walk Shuttle {
    enabled FALSE
    rate 10
    children [
        DEF Near ProximitySensor { size 10 10 10 }
        DEF Turn Pendulum {
            enabled FALSE

            children [
                # The Robot
                Shape {
                    geometry Box { } # head
                }
                Transform {
                    scale 1 5 1
                    translation 0 -5 0
                    children [ Shape { geometry Sphere { } } ] # body
                }
                DEF Arm Pendulum {
                    maxAngle 0.52 # 30 degrees
                    enabled FALSE

                    children [
                        Transform {
                            scale 1 7 1
                            translation 1 -5 0
                            rotation 1 0 0 4.45 # rotate so swing
                                                # centers on Y axis
                            center 0 3.5 0

                            children [
                                Shape { geometry Cylinder { } }
                            ]
                        }
                    ]
                }

                # duplicate arm on other side and flip so it swings
```

```
                # in opposition
                Transform {
                    rotation 0 1 0 3.14159
                    translation 10 0 0
                    children [ USE Arm ]
                }
            ]
        }
    ]
}

# hook up the sentry.  The arms will swing infinitely.  He walks
# along the shuttle path, then turns, then walks back, etc.
ROUTE Near.isActive TO Arm.enabled
ROUTE Near.isActive TO Walk.enabled
ROUTE Arm.isAtCW TO Arm.moveCCW
ROUTE Walk.isAtLeft TO Turn.moveCCW
ROUTE Turn.isAtCW TO Walk.moveRight
```

Chopper

Here is a simple example of how to do simple animation triggered by a touchsensor. It uses an EXTERNPROTO to include a Rotor node from the net which will do the actual animation.

```
EXTERNPROTO Rotor [
    eventIn MFFloat Spin
    field MFNode children ]
 "http://somewhere/Rotor.wrl" # Where to look for implementation


PROTO Chopper [
    field SFFloat maxAltitude 30
       field SFFloat rotorSpeed 1 ]
{
    Group {
        children [
            DEF Touch TouchSensor { }, # Gotta get touch events
            Shape { ... body... },
            DEF Top Rotor { ... geometry ... },
            DEF Back Rotor { ... geometry ... }
        ]
    }

    DEF SCRIPT Script {
        eventIn SFBool startOrStopEngines
        field maxAltitude IS maxAltitude
        field rotorSpeed IS rotorSpeed
        field SFNode topRotor USE Top
        field SFNode backRotor USE Back
        field SFBool bEngineStarted FALSE

        url "chopper.vs"
    }

    ROUTE Touch.isActive -> SCRIPT.startOrStopEngines
}
```

```
DEF MyScene Group {
    DEF MikesChopper Chopper { maxAltitude 40 }
}


chopper.vs:
-------------
    function startOrStopEngines(value, ts) {
        // Don't do anything on mouse-down:
        if (value) return;

        // Otherwise, start or stop engines:
        if (!bEngineStarted) {
            StartEngine();
        }
        else {
            StopEngine();
        }
    }

    function SpinRotors(fInRotorSpeed, fSeconds) {
        rp[0] = 0;
        rp[1] = fInRotorSpeed;
        rp[2] = 0;
        rp[3] = fSeconds;
        TopRotor.Spin = rp;

        rp[0] = fInRotorSpeed;
        rp[1] = 0;
        rp[2] = 0;
        rp[3] = fSeconds;
        BackRotor.Spin = rp;
    }

    function StartEngine() {
        // Sound could be done either by controlling a PointSound node
        // (put into another SFNode field) OR by adding/removing a
        // PointSound from the Separator (in which case the Separator
        // would need to be passed in an SFNode field).

        SpinRotors(fRotorSpeed, 3);
        bEngineStarted = TRUE;
    }

    function StopEngine() {
        SpinRotors(0, 6);
        bEngineStarted = FALSE;
    }
}
```

Guided Tour

Moving Worlds has great facilities to put the viewer's camera under control of a script. This is useful for things such as guided tours, merry-go-round rides, and transportation devices such as busses and elevators. These next 2 examples show a couple of ways to use this feature.

The first example is a simple guided tour through the world. Upon entry, a guide orb hovers in front of you. Click on this and your tour through the world begins. The orb follows you around on your tour. Perhaps a PointSound node can be embedded inside to point out the sights. A ProximitySensor ensures that the tour is started only if the user is close to the initial starting point. Note that this is done without scripts thanks to the touchTime output of the TouchSensor.

```
Group {
    children [
        <geometry for the world>,

        DEF GuideTransform Transform {
            children [
                DEF TourGuide Viewpoint { jump FALSE },
                DEF ProxSensor ProximitySensor { size 10 10 10 }
                DEF StartTour TouchSensor { },
                Shape { geometry Sphere { } }, # the guide orb
            ]
        }
    ]
}

DEF GuidePI PositionInterpolator {
    keys [ ... ]
    values [ ... ]
}

DEF GuideRI RotationInterpolator {
    keys [ ... ]
    values [ ... ]
}

DEF TS TimeSensor { cycleInterval 60 } # 60 second tour

ROUTE ProxSensor.isActive TO StartTour.enabled
ROUTE StartTour.touchTime TO TS.startTime
ROUTE TS.isActive TO TourGuide.bind
ROUTE TS.fraction TO GuidePI.set_fraction
ROUTE TS.fraction TO GuideRI.set_fraction
ROUTE GuidePI.outValue TO GuideTransform.set_translation
ROUTE GuideRI.outValue TO GuideTransform.set_rotation
```

Elevator

Here's another example of animating the camera. This time it's an elevator to ease access to a multistory building. For this example I'll just show a 2 story building and I'll assume that the elevator is already at the ground floor. To go up you just step inside. A ProximitySensor fires and starts the elevator up automatically. I'll leave call buttons for outside the elevator, elevator doors and floor selector buttons as an exercise for the reader!

```
Group {
    children [

        DEF ETransform Transform {
```

```
            children [
                DEF EViewpoint Viewpoint { }
                DEF EProximity ProximitySensor { size 2 2 2 }
                <geometry for the elevator,
                 a unit cube about the origin with a doorway>
            ]
        }
    ]
}
DEF ElevatorPI PositionInterpolator {
    keys [ 0, 1 ]
    values [ 0 0 0, 0 4 0 ] # a floor is 4 meters high
}
DEF TS TimeSensor { cycleInterval 10 } # 10 second travel time

DEF S Script {
    field SFNode viewpoint USE EViewpoint
    eventIn SFBool active
    eventIn SFBool done
    eventOut SFTime start
    behavior "Elevator.java"
}

ROUTE EProximity.enterTime TO TS.startTime
ROUTE TS.isActive TO EViewpoint.bind
ROUTE TS.fraction_changed TO ElevatorPI.set_fraction
ROUTE ElevatorPI.value_changed TO ETransform.set_translation
```

Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm.org with questions or comments.

The Virtual Reality Modeling Language

Appendix C. Java Scripting Reference

Version 2.0, ISO/IEC WD 14772

August 4, 1996

This annex describes the Java classes and methods that allow Script nodes (see "*Nodes Reference - Script*") to interact with associated scenes. See "*Concepts - Scripting*" for a general description of scripting languages in VRML.

C.1 Language

Java(TM) is an object-oriented, platform-independent, multi-threaded, general-purpose programming environment developed at Sun Microsystems, Inc. See the Java web site for a full description of the Java programming language (http://java.sun.com/). This appendix describes the Java bindings of VRML to the Script node.

C.2 Supported Protocol in the Script Node's *url* Field

The *url* field of the Script node contains the URL of a file containing the Java byte code, for example:

```
Script {
    url "http://foo.co.jp/Example.class"
    eventIn SFBool start
}
```

C.2.1 File Extension

The file extension for Java byte code is **.class**.

C.2.2 MIME Type

The MIME type for Java byte code is defined as follows:

```
application/x-java
```

C.3 EventIn Handling

Events to the Script node are passed to the corresponding Java method (processEvent or processEvents) in the script. It is necessary to specify the script in the url field of the Script node.

If a Java byte code file is specified in the url field, the following two conditions must hold:

- it must contain the class definition whose name is exactly the same as the body of the file name, and
- it must be a subclass of 'Script' class in "*vrml.node Package*".

For example, the following Script node has one eventIn field whose name is 'start'.

```
Script {
       url "http://foo.co.jp/Example.class"
       eventIn SFBool start
}
```

This node points to the script file 'Example.class' - its source ('Example.java') looks like this:

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

class Example extends Script {
    ...
    // This method is called when any event is received
    public void processEvent (Event e) {
       // ... perform some operation ...
    }
}
```

In the above example, when the start eventIn is sent the processEvent() method is executed and receives the eventIn.

C.3.1 Parameter Passing and the EventIn Field/Method

When a Script node receives an eventIn, a processEvent() or processEvents() method in the file specified in url field of the Script node is called, which receives the eventIn as Java objects (Event object). See "*processEvent() or processEvents() Method*".

The Event object has three information associated with it: name, value and timestamp of the eventIn. These can be retrieved using the corresponding method on the Event object.

```
class Event {
   public String getName();
   public ConstField getValue();
   public double getTimeStamp();
}
```

Suppose that the eventIn type is **SFXXX** and eventIn name is **eventInYYY**, then

- the getName() should return "**eventInYYY**"
- the getValue() should return ConstField.
- getTimeStamp() should return the timestamp when the eventIn was occurred

In the above example, the eventIn name would be "start" and eventIn value could be casted to be ConstSFBool. Also, the timestamp for the time when the eventIn was occurred is available as a double. These are passed as an Event object to processEvent() method:

```
public void processEvent (Event e) {
    if(e.getName().equals("start")){
          ConstSFBool v = (ConstSFBool)e.getValue();
          if(v.getValue()==true){
               // ... perform some operation ...
          }
     }
}
```

C.3.2 *processEvents()* and *processEvent()* Method

Authors can define a processEvents method within a class that is called when the script receives some set of events. The prototype of processEvents method is

```
public void processEvents(int count, Event events[]);
```

count indicates the number of events delivered. *events* is the array of events delivered. Its default behavior is to iterate over each event, calling processEvent() on each one as follows:

```
public void processEvents(int count, Event events[])
{
    for (int i = 0; i < count; i++) {
        processEvent( events[i] );
    }
}
```

Although authors could change this operation by giving a user-defined processEvents() method, in most cases, they would only change processEvent() method and eventsProcessed() method described below.

When multiple eventIns are routed from single node to single script node and then the eventIns which have the same timestamp are occurred, the processEvents() receives multiple events as the event array. Otherwise, each coming event invokes separate processEvents().

For example, processEvents() method receives two events in the following case :

```
Transform {
    children [
        DEF TS TouchSensor {}
        Shape { geometry Cone {} }
    ]
}
DEF SC Script {
    url     "Example.class"
    eventIn SFBool isActive
    eventIn SFTime touchTime
}
ROUTE TS.isActive  TO SC.isActive
ROUTE TS.touchTime TO SC.touchTime
```

Authors can define a processEvent method within a class. The prototype of processEvent is

```
public void processEvent(Event event);
```

Its default behavior is no operation.

C.3.3 *eventsProcessed()* Method

Authors can define an *eventsProcessed* method within a class that is called after some set of events has been received. It allows Scripts that do not rely on the ordering of events received to generate fewer events than an equivalent Script that generates events whenever events are received. It is called after every invocation of processEvents().

Events generated from an eventsProcessed routine are given the timestamp of the last event processed.

The prototype of eventsProcessed method is

```
public void eventsProcessed();
```

Its default behavior is no operation.

C.3.4 *shutdown()* Method

Authors can define a *shutdown* method within a class that is called when the corresponding Script node is deleted.

The prototype of shutdown method is

```
public void shutdown();
```

Its default behavior is no operation.

C.3.5 *initialize()* Method

Authors can define an initialize method within a class that is called before any event is generated. The various methods on Script such as getEventIn, getEventOut, getExposedField, and getField are not guaranteed to return correct values before the call to initialize (i.e. in the constructor). initialize() is called once during the life of Script object.

The prototype of initialize method is

```
public void initialize();
```

Its default behavior is no operation.

C.4 Accessing Fields and Events

The fields, eventIns and eventOuts of a Script node are accessible from their corresponding Java classes.

C.4.1 Accessing Fields, EventIns and EventOuts of the Script

Each field defined in the Script node is available to the script by using its name. Its value can be read or written. This value is persistent across function calls. EventOuts defined in the script node can also be read.

Accessing fields of Script node can be done by using Script class methods. Script class has several methods to do that: getField(), getEventOut(), getEventIn() and getExposedField().

- Field **getField**(String *fieldName*)
  is the method to get the reference to the Script node's 'field' whose name is *fieldName*. The return value can be converted to an appropriate Java "*Field Class*".
- Field **getEventOut**(String *eventName*)
  is the method to get the reference to the Script node's 'eventOut' whose name is *eventName*. The return value can be converted to an appropriate Java "*Field Class*".
- Field **getEventIn**(String *eventName*)
  is the method to get the reference to the Script node's 'eventIn' whose name is *eventName*. The return value can be converted to an appropriate Java "*Field Class*". When you call getValue() method on a 'field' object obtained by getEventIn() method, the return value is unspecified. Therefore, you can not rely on it. 'EventIn' is a write-only field.

When you call setValue(), set1Value(), addValue() or insertValue() method on a 'field' object obtained by **getField()** method, the value specified as an argument is stored in the corresponding VRML node's field.

When you call setValue(), set1Value(), addValue() or insertValue() method on a 'field' object obtained by **getEventOut()** method, the value specified as an argument generates an event in VRML scene. The effect of this event is specified by the associated Route in the VRML scene.

When you call setValue(), set1Value(), addValue() or insertValue() methods on a 'field' object obtained by **getEventIn()** method, the value specified an argument generates an event to the Script node. For example, the following Script node defines an eventIn, *start*, a field, *state*, and an eventOut, *on*. The method *initialize()* is invoked before any events are received, and the method *processEvent()* is invoked when *start* receives an event:

```
Script {
    url       "Example.class"
    eventIn   SFBool start
    eventOut  SFBool on
    field     SFBool state TRUE
}
```

```
Example.class:

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

  class Example extends Script {
     private SFBool state;
     private SFBool on;

     public void initialize(){
          state = (SFBool) getField("state");
          on = (SFBool) getEventOut("on");
      }

      public void processEvent(Event e) {
          if(state.getValue()==true){
              on.setValue(true); // set true to eventOut 'on'
              state.setValue(false);
          }
          else {
              on.setValue(false); // set false to eventOut 'on'
              state.setValue(true);
          }
      }
  }
```

C.4.2 Accessing Fields, EventIns and EventOuts of Other Nodes

If a script program has an access to any node, any eventIn, eventOut or exposedField of that node is accessible by using the getEventIn(), getEventOut() method or getExposedField() method defined on the node's class (see "*Exposed Classes and Methods for Nodes and Fields*").

The typical way for a script program to have an access to another VRML node is to have an SFNode field which provides a reference to the other node. The following example shows how this is done:

```
DEF SomeNode Transform { }
Script {
```

```
        field SFNode node USE SomeNode
        eventIn SFVec3f pos
        url "Example.class"
    }
```

Example.class:

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

    class Example extends Script {
        private SFNode node;
        private SFVec3 trans;

       public void initialize(){
            private SFNode node = (SFNode) getField("node");
       }

        public void processEvent(Event e) {
             // gets the ref to 'translation' field of Transform node

             trans = (SFVec3f)(node.getValue())
                                         .getExposedField("translation");
             trans.setValue((ConstSFVec3f)e.getValue());
        }
    }
```

C.4.3 Sending EventIns or EventOuts

Sending eventOuts from script is done by setting value to the reference to the 'eventOut' of the script by setValue(), set1Value(), addValue() or insertValue() method. Sending eventIns from script is done by setting value to the reference to the 'eventIn' by setValue(), set1Value(), addValue() or insertValue() method.

C.5 Exposed Classes and Methods for Nodes and Fields

Java classes for VRML are defined in the packages: *vrml, vrml.node* and *vrml.field*.

The Field class extends Java's Object class by default; thus, Field has the full functionality of the Object class, including the **getClass()** method. The rest of the package defines a "Const" read-only class for each VRML field type, with a **getValue()** method for each class; and another read/write class for each VRML field type, with both **getValue()** and **setValue()** methods for each class. A getValue() method converts a VRML-type value into a Java-type value. A setValue() method converts a Java-type value into a VRML-type value and sets it to the VRML field.

Most of the **setValue()** methods and **set1Value()** methods are listed as "throws exception," meaning that errors are possible -- you may need to write exception handlers (using Java's **catch()** method) when you use those methods. Any method not listed as "throws exception" is guaranteed to generate no exceptions. Each method that throws an exception includes a prototype showing which exception(s) can be thrown.

C.5.1 Field Class and ConstField Class

All VRML data types have an equivalent classes in Java.

```
class Field {
}
```

Field class is the root of each field types. This class has two subclasses : read-only class and writable class

- **Read-only class**
  This class supports **getValue()** method. In addition, some classes support some convenient methods to get value from the object.

  ConstSFBool, ConstSFColor, ConstMFColor, ConstSFFloat, ConstMFFloat, ConstSFImage, ConstSFInt32, ConstMFInt32, ConstSFNode, ConstMFNode, ConstSFRotation, ConstMFRotation, ConstSFString, ConstMFString, ConstSFVec2f, ConstMFVec2f, ConstSFVec3f, ConstMFVec3f, ConstSFTime, ConstMFTime

- **Writable class**
  This type of classes support both **getValue()** and **setValue()** methods. If the class name is prefixed with **MF** meaning that it is a multiple valued field class, the class also supports the **set1Value(), addValue() and insertValue()** method.
  In addition, some classes support some convenient methods to get and set value from the object.

  SFBool, SFColor, MFColor, SFFloat, MFFloat, SFImage, SFInt32, MFInt32, SFNode, MFNode, SFRotation, MFRotation, SFString, MFString, SFVec2f, MFVec2f, SFVec3f, MFVec3f, SFTime, MFTime

The Java *Field* class and its subclasses have several methods to get and set values: getSize(), getValue(), get1Value(), setValue(), set1Value(), addValue() or insertValue().

- **getSize**()
  is the method to return the number of elements of each multiple value field class(MF class).
- **getValue**()
  is the method to convert a VRML-type value into a Java-type value and returns it.
- **get1Value**(int *index*)
  is the method to convert a VRML-type value (*index*-th element) into a Java-type value and returns it. The index of the first element is 0. Getting the element beyond the existing elements throws exception.
- **setValue**(*value*)
  is the method to convert a Java-type *value* into a VRML-type value and sets it to the VRML field.
- **set1Value**(int *index*, *value*)
  is the method to convert from a Java-type *value* to a VRML-type value and set it to the *index*-th element.
- **addValue**(*value*)
  is the method to convert from a Java-type *value* to a VRML-type value and add it to the last element.

- **insertValue**(int *index*, *value*)
  is the method to convert from a Java-type *value* to a VRML-type value and insert it to the *index*-th element. The index of the first element is 0. Setting the element beyond the existing elements throws exception.

In these methods, getSize(), get1Value(), set1Value(), addValue() and insertValue() are only available for multiple value field classes(MF classes). See "*vrml Package*" for each classes' methods definition.

C.5.2 *Node* Class

Node class has several methods: getType(), getEventOut(), getEventIn(), getExposedField(), getBrowser()

- String **getType**()
  is the method to returns the type of the node.
- ConstField **getEventOut**(String *eventName*)
  is the method to get the reference to the node's 'eventOut' whose name is *eventName*. The return value can be converted to an appropriate Java "*Field Class*".
- Field **getEventIn**(String *eventName*)
  is the method to get the reference to the node's 'eventIn' whose name is *eventName*. The return value can be converted to an appropriate Java "*Field Class*". When you call getValue() method on a 'field' object obtained by getEventIn() method, the return value is unspecified. Therefore, you can not rely on it. 'EventIn' is a write-only field.
- Field **getExposedField**(String *eventName*)
  is the method to get the reference to the node's 'exposedField' whose name is *eventName*. The return value can be converted to an appropriate Java "*Field Class*".
- Browser **getBrowser**()
  is method to get the browser that this node is contained in. See "*Browser Class*".

When you call setValue(), set1Value(), addValue() or insertValue() method on a 'field' object obtained by **getEventIn()** method, the value specified as an argument generates an event to the node.

When you call setValue(), set1Value(), addValue() or insertValue() method on a 'field' object obtained by **getExposedField()** method, the value specified as an argument generates an event in VRML scene. The effect of this event is specified by the associated Route in the VRML scene.

C.5.3 *Browser* Class

This section lists the public Java interfaces to the *Browser* class, which allows scripts to get and set browser information. For descriptions of the following methods see the "*Concepts - Scripting - Browser Interface*".

Return value	Method name
String	***getName***()
String	***getVersion***()
float	***getCurrentSpeed***()
float	***getCurrentFrameRate***()
String	***getWorldURL***()
void	***replaceWorld***(*Node[] nodes*)
Node[]	***createVrmlFromString***(*String vrmlSyntax*)
void	***createVrmlFromURL***(*String[] url, Node node, String event*)
void	***addRoute***(*Node fromNode, String fromEventOut, Node toNode, String toEventIn*)
void	***deleteRoute***(*Node fromNode, String fromEventOut, Node toNode, String toEventIn*)
void	***loadURL***(*String[] url, String[] parameter*)
void	***setDescription***(*String description*)

See "*vrml Package*"for each method's definition.

Conversion table from the types used in Browser class to Java type.

VRML type	Java type
SFString	*String*
SFFloat	*float*
MFString	*String[]*
MFNode	*Node[]*

C.5.4 User-defined Classes and Packages

The Java classes defined by a user can be used in the Java program. They are searched from the directory where the Java program is placed.

If the Java class is in a package, this package is searched from the directory where the Java program is placed.

C.5.5 Standard Java Packages

Java programs have access to the full set of classes available in java.* The handling of these classes - especially *AWT*, and the security model of networking - will be browser specific. Threads are required to work as normal for Java.

C.6 Exceptions

Java methods may throw the following exceptions:

- **InvalidFieldException**
  is thrown at the time getField() is executed and the field name is invalid.
- **InvalidEventInException**
  is thrown at the time getEventIn() is executed and the eventIn name is invalid.
- **InvalidEventOutException**
  is thrown at the time getEventOut() or getEventOut() is executed and the eventOut name is invalid.
- **InvalidExposedFieldException**
  is thrown at the time getExposedField() is executed and the exposedField name is invalid.
- **InvalidVRMLSyntaxException**
  is thrown at the time createVrmlFromString(), createVrmlFromURL() or loadURL() is executed and the vrml string is invalid.
- **InvalidRouteException**
  is thrown at the time addRoute() or deleteRoute() is executed and one or more the arguments is invalid.
- **InvalidFieldChangeException**
  may be thrown as a result of all sorts of illegal field changes, for example:
    - Adding a node from one World as the child of a node in another World.
    - Creating a circularity in a scene graph.
    - Setting an invalid string on enumerated fields, such as the fogType field of the Fog node.

  It is not guaranteed that such exceptions will be thrown, but a browser should do the best job it can.

- **InvalidNavigationTypeException**
  is thrown at the time setNavigationType() is executed and the argument is invalid.
- **ArrayIndexOutOfBoundsException**
  is generated at the time setValue(), set1Value(), addValue() or insertValue() is executed and the index is out of bound. This is the standard exception defined in the Java Array class.

If exceptions are not redefined by authors, a browser's behavior is unspecified - see "*Example of Exception Class*".

C.7 Example

Here's an example of a Script node which determines whether a given color contains a lot of red. The Script node exposes a color field, an eventIn, and an eventOut:

```
Script {
    field    SFColor currentColor 0 0 0
    eventIn  SFColor colorIn
    eventOut SFBool  isRed
    url "ExampleScript.class"
}
```

And here's the source code for the "ExampleScript.java" file that gets called every time an eventIn is routed to the above Script node:

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

class ExampleScript extends Script {
    // Declare field(s)
    private SFColor currentColor;

    // Declare eventOut field(s)
    private SFBool isRed;

    public void initialize(){
       currentColor = (SFColor) getField("currentColor");
       isRed = (SFBool) getEventOut("isRed");
    }

    public void processEvent(Event e){
        // This method is called when a colorIn event is received
        currentColor.setValue((ConstSFColor)e.getValue());
    }

    public void eventsProcessed() {
        if (currentColor.getValue()[0] >= 0.5) // if red is at or above 50%
            isRed.setValue(TRUE);
    }
}
```

For details on when the methods defined in ExampleScript are called - see "*Concepts -Execution Model*".

Browser class examples:

- createVrmlFromUrl method

```
DEF Example Script {
    url "Example.class"
    field   MFString target_url "foo.wrl"
    eventIn MFNode   nodesLoaded
    eventIn SFBool   trigger_event
  }
```

```
Example.class:
  import vrml.*;
  import vrml.field.*;
```

```
import vrml.node.*;

class Example extends Script {
    private MFString target_url;
    private Browser browser;

    public void initialize(){
        target_url = (MFString)getField("target_url");
        browser = this.getBrowser();
    }

    public void processEvent(Event e){
        if(e.getName().equals("trigger_event")){
            // do something and then fetch values
            browser.createVRMLFromURL(target_url.getValue(), this, "nodesLoade
       }
        if(e.getName().equals("nodesLoaded")){
            // do something
        }
    }
}
```

- addRoute method

```
DEF Sensor TouchSensor {}
DEF Example Script {
  url "Example.class"
  field   SFNode fromNode USE Sensor
  eventIn SFBool clicked
  eventIn SFBool trigger_event
}
```

```
Example.class:
  import vrml.*;
  import vrml.field.*;
  import vrml.node.*;

  class Example extends Script {
      private SFNode fromNode;
      private Browser browser;

      public void initialize(){
          fromNode = (SFNode) getField("fromNode");
          browser = this.getBrowser();
      }

      public void processEvent(Event e){
         if(e.getName().equals("trigger_event")){
              // do something and then add routing
              browser.addRoute(fromNode.getValue(), "isActive", this, "clicked")
         }
         if(e.getName().equals("clicked")){
              // do something
          }
      }
  }
```

C.8 Class Definitions

C.8.1 Class Hierarchy

The classes are divided into three packages: *vrml*, *vrml.field* and *vrml.node*.

```
java.lang.Object
     |
     +- vrml.Event
     +- vrml.Browser
     +- vrml.Field
     |       +- vrml.field.SFBool
     |       +- vrml.field.SFColor
     |       +- vrml.field.SFFloat
     |       +- vrml.field.SFImage
     |       +- vrml.field.SFInt32
     |       +- vrml.field.SFNode
     |       +- vrml.field.SFRotation
     |       +- vrml.field.SFString
     |       +- vrml.field.SFTime
     |       +- vrml.field.SFVec2f
     |       +- vrml.field.SFVec3f
     |       |
     |       +- vrml.MField
     |       |       +- vrml.field.MFColor
     |       |       +- vrml.field.MFFloat
     |       |       +- vrml.field.MFInt32
     |       |       +- vrml.field.MFNode
     |       |       +- vrml.field.MFRotation
     |       |       +- vrml.field.MFString
     |       |       +- vrml.field.MFTime
     |       |       +- vrml.field.MFVec2f
     |       |       +- vrml.field.MFVec3f
     |       |
     |       +- vrml.ConstField
     |               +- vrml.field.ConstSFBool
     |               +- vrml.field.ConstSFColor
     |               +- vrml.field.ConstSFFloat
     |               +- vrml.field.ConstSFImage
     |               +- vrml.field.ConstSFInt32
     |               +- vrml.field.ConstSFNode
     |               +- vrml.field.ConstSFRotation
     |               +- vrml.field.ConstSFString
     |               +- vrml.field.ConstSFTime
     |               +- vrml.field.ConstSFVec2f
     |               +- vrml.field.ConstSFVec3f
     |               |
     |               +- vrml.ConstMField
     |                       +- vrml.field.ConstMFColor
     |                       +- vrml.field.ConstMFFloat
     |                       +- vrml.field.ConstMFInt32
     |                       +- vrml.field.ConstMFNode
     |                       +- vrml.field.ConstMFRotation
     |                       +- vrml.field.ConstMFString
     |                       +- vrml.field.ConstMFTime
     |                       +- vrml.field.ConstMFVec2f
     |                       +- vrml.field.ConstMFVec3f
     |
     +- vrml.BaseNode
```

```
            +- vrml.node.Node
            +- vrml.node.Script

java.lang.Exception
        java.lang.RuntimeException
                vrml.InvalidRouteException
                vrml.InvalidFieldException
                vrml.InvalidEventInException
                vrml.InvalidEventOutException
                vrml.InvalidExposedFieldException
                vrml.InvalidNavigationTypeException
                vrml.InvalidFieldChangeException
        vrml.InvalidVRMLSyntaxException
```

C.8.2 vrml Packages

C.8.2.1 *vrml* Package

```
package vrml;

public abstract class Field implements Cloneable
{
   public Object clone();
}

public abstract class ConstField extends Field
{
}

public class ConstMField extends ConstField
{
   public int getSize();
}

public class MField extends Field
{
   public int getSize();
   public void clear();
   public void delete(int index);
}

public class Event implements Cloneable {
  public String getName();
  public double getTimeStamp();
  public ConstField getValue();
  public Object clone();
}

public class Browser {
  // Browser interface
  public String getName();
  public String getVersion();

  public float getCurrentSpeed();

  public float getCurrentFrameRate();

  public String getWorldURL();
  public void replaceWorld(Node[] nodes);
```

```
  public Node[] createVrmlFromString(String vrmlSyntax)
    throws InvalidVRMLSyntaxException;

  public void createVrmlFromURL(String[] url, Node node, String event)
    throws InvalidVRMLSyntaxException;

  public void addRoute(Node fromNode, String fromEventOut,
    Node toNode, String toEventIn);
  public void deleteRoute(Node fromNode, String fromEventOut,
    Node toNode, String toEventIn);

  public void loadURL(String[] url, String[] parameter)
    throws InvalidVRMLSyntaxException;
  public void setDescription(String description);
}

//
// This is the general BaseNode class
//
public abstract class BaseNode
{
  // Returns the type of the node.  If the node is a prototype
  //   it returns the name of the prototype.
  public String getType();

  // Get the Browser that this node is contained in.
  public Browser getBrowser();
}
```

C.8.2.2 *vrml.field* Package

```
package vrml.field;

public class ConstSFBool extends ConstField
{
   public boolean getValue();
}

public class ConstSFColor extends ConstField
{
   public void getValue(float color[]);
   public float getRed();
   public float getGreen();
   public float getBlue();
}

public class ConstSFFloat extends ConstField
{
   public float getValue();
}

public class ConstSFImage extends ConstField
{
   public int getWidth();
   public int getHeight();
   public int getComponents();
   public void getPixels(byte pixels[]);
}

public class ConstSFInt32 extends ConstField
```

```
{
   public int getValue();
}

public class ConstSFNode extends ConstField
{
  /* ****************************************
   * Return value of getValue() must extend Node class.
   * The concrete class is implementation dependent
   * and up to browser implementation.
   ****************************************** */
   public Node getValue();
}

public class ConstSFRotation extends ConstField
{
   public void getValue(float[] rotation);
}

public class ConstSFString extends ConstField
{
   public String getValue();
}

public class ConstSFTime extends ConstField
{
   public double getValue();
}

public class ConstSFVec2f extends ConstField
{
   public void getValue(float vec2[]);
   public float getX();
   public float getY();
}

public class ConstSFVec3f extends ConstField
{
   public void getValue(float vec3[]);
   public float getX();
   public float getY();
   public float getZ();
}

public class ConstMFColor extends ConstMField
{
   public void getValue(float colors[][]);
   public void getValue(float colors[]);
   public void get1Value(int index, float color[]);
   public void get1Value(int index, SFColor color);
}

public class ConstMFFloat extends ConstMField
{
   public void getValue(float values[]);
   public float get1Value(int index);
}

public class ConstMFInt32 extends ConstMField
{
   public void getValue(int values[]);
```

```
   public int get1Value(int index);
}

public class ConstMFNode extends ConstMField
{
  /****************************************
   * Return value of getValue() must extend Node class.
   * The concrete class is implementation dependent
   * and up to browser implementation.
   ****************************************/
   public void getValue(Node values[]);
   public Node get1Value(int index);
}

public class ConstMFRotation extends ConstMField
{
   public void getValue(float rotations[][]);
   public void getValue(float rotations[]);
   public void get1Value(int index, float rotation[]);
   public void get1Value(int index, SFRotation rotation);
}

public class ConstMFString extends ConstMField
{
   public void getValue(String values[]);
   public String get1Value(int index);
}

public class ConstMFTime extends ConstMField
{
   public void getValue(double times[]);
   public double get1Value(int index);
}

public class ConstMFVec2f extends ConstMField
{
   public void getValue(float vecs[][]);
   public void getValue(float vecs[]);
   public void get1Value(int index, float vec[]);
   public void get1Value(int index, SFVec2f vec);
}

public class ConstMFVec3f extends ConstMField
{
   public void getValue(float vecs[][]);
   public void getValue(float vecs[]);
   public void get1Value(int index, float vec[]);
   public void get1Value(int index, SFVec3f vec);
}

public class SFBool extends Field
{
   public SFBool(boolean value);
   public boolean getValue();
   public void setValue(boolean b);
   public void setValue(ConstSFBool b);
   public void setValue(SFBool b);
}

public class SFColor extends Field
{
```

```
   public SFColor(float red, float green, float blue);
   public void getValue(float color[]);
   public float getRed();
   public float getGreen();
   public float getBlue();
   public void setValue(float color[]);
   public void setValue(float red, float green, float blue);
   public void setValue(ConstSFColor color);
   public void setValue(SFColor color);
}

public class SFFloat extends Field
{
   public SFFloat(float f);
   public float getValue();
   public void setValue(float f);
   public void setValue(ConstSFFloat f);
   public void setValue(SFFloat f);
}

public class SFImage extends Field
{
   public SFImage(int width, int height, int components, byte pixels[]);
   public int getWidth();
   public int getHeight();
   public int getComponents();
   public void getPixels(byte pixels[]);
   public void setValue(int width, int height, int components,
                        byte pixels[]);
   public void setValue(ConstSFImage image);
   public void setValue(SFImage image);
}

public class SFInt32 extends Field
{
   public SFInt32(int value);
   public int getValue();
   public void setValue(int i);
   public void setValue(ConstSFInt32 i);
   public void setValue(SFInt32 i);
}

public class SFNode extends Field
{
   public SFNode(Node node);

  /******************************************
   * Return value of getValue() must extend Node class.
   * The concrete class is implementation dependent
   * and up to browser implementation.
   ******************************************/
  public Node getValue();
   public void setValue(Node node);
   public void setValue(ConstSFNode node);
   public void setValue(SFNode node);
}

public class SFRotation extends Field
{
   public SFRotation(float axisX, float axisY, float axisZ, float rotation);
   public void getValue(float[] rotation);
```

```
   public void setValue(float[] rotation);
   public void setValue(float axisX, float axisY, float axisZ, float rotation);
   public void setValue(ConstSFRotation rotation);
   public void setValue(SFRotation rotation);
}

public class SFString extends Field
{
   public SFString(String s);
   public String getValue();
   public void setValue(String s);
   public void setValue(ConstSFString s);
   public void setValue(SFString s);
}

public class SFTime extends Field
{
   public SFTime(double time);
   public double getValue();
   public void setValue(double time);
   public void setValue(ConstSFTime time);
   public void setValue(SFTime time);
}

public class SFVec2f extends Field
{
   public SFVec2f(float x, float y);
   public void getValue(float vec[]);
   public float getX();
   public float getY();
   public void setValue(float vec[]);
   public void setValue(float x, float y);
   public void setValue(ConstSFVec2f vec);
   public void setValue(SFVec2f vec);
}

public class SFVec3f extends Field
{
   public SFVec3f(float x, float y, float z);
   public void getValue(float vec[]);
   public float getX();
   public float getY();
   public float getZ();
   public void setValue(float vec[]);
   public void setValue(float x, float y, float z);
   public void setValue(ConstSFVec3f vec);
   public void setValue(SFVec3f vec);
}

public class MFColor extends MField
{
   public MFColor(float value[][]);
   public MFColor(float value[]);
   public MFColor(int size, float value[]);

   public void getValue(float colors[][]);
   public void getValue(float colors[]);

   public void setValue(float colors[][]);
   public void setValue(int size, float colors[]);
   /**************************************************
```

```
    color[0] ... color[size - 1] are used as color data
    in the way that color[0], color[1], and color[2]
    represent the first color. The number of colors
    is defined as "size / 3".
    ***************************************************/

   public void setValue(ConstMFColor colors);

   public void get1Value(int index, float color[]);
   public void get1Value(int index, SFColor color);

   public void set1Value(int index, ConstSFColor color);
   public void set1Value(int index, SFColor color);
   public void set1Value(int index, float red, float green, float blue);

   public void addValue(ConstSFColor color);
   public void addValue(SFColor color);
   public void addValue(float red, float green, float blue);

   public void insertValue(int index, ConstSFColor color);
   public void insertValue(int index, SFColor color);
   public void insertValue(int index, float red, float green, float blue);
}

public class MFFloat extends MField
{
   public MFFloat(float values[]);

   public void getValue(float values[]);

   public void setValue(float values[]);
   public void setValue(int size, float values[]);
   public void setValue(ConstMFFloat value);

   public float get1Value(int index);

   public void set1Value(int index, float f);
   public void set1Value(int index, ConstSFFloat f);
   public void set1Value(int index, SFFloat f);

   public void addValue(float f);
   public void addValue(ConstSFFloat f);
   public void addValue(SFFloat f);

   public void insertValue(int index, float f);
   public void insertValue(int index, ConstSFFloat f);
   public void insertValue(int index, SFFloat f);
}

public class MFInt32 extends MField
{
   public MFInt32(int values[]);

   public void getValue(int values[]);

   public void setValue(int values[]);
   public void setValue(int size, int values[]);
   public void setValue(ConstMFInt32 value);

   public int get1Value(int index);
```

```
   public void set1Value(int index, int i);
   public void set1Value(int index, ConstSFInt32 i);
   public void set1Value(int index, SFInt32 i);

   public void addValue(int i);
   public void addValue(ConstSFInt32 i);
   public void addValue(SFInt32 i);

   public void insertValue(int index, int i);
   public void insertValue(int index, ConstSFInt32 i);
   public void insertValue(int index, SFInt32 i);
}

public class MFNode extends MField
{
   public MFNode(Node node[]);

  /******************************************
   * Return value of getValue() must extend Node class.
   * The concrete class is implementation dependent
   * and up to browser implementation.
   ******************************************/
   public void getValue(Node node[]);

   public void setValue(Node node[]);
   public void setValue(int size, Node node[]);
   public void setValue(ConstMFNode node);

   public Node get1Value(int index);

   public void set1Value(int index, Node node);
   public void set1Value(int index, ConstSFNode node);
   public void set1Value(int index, SFNode node);

   public void addValue(Node node);
   public void addValue(ConstSFNode node);
   public void addValue(SFNode node);

   public void insertValue(int index, Node node);
   public void insertValue(int index, ConstSFNode node);
   public void insertValue(int index, SFNode node);
}

public class MFRotation extends MField
{
   public MFRotation(float rotations[][]);
   public MFRotation(float rotations[]);
   public MFRotation(int size, float rotations[]);

   public void getValue(float rotations[][]);
   public void getValue(float rotations[]);

   public void setValue(float rotations[][])
   public void setValue(int size, float rotations[]);
   public void setValue(ConstMFRotation rotations);

   public void get1Value(int index, float rotation[]);
   public void get1Value(int index, SFRotation rotation);

   public void set1Value(int index, ConstSFRotation rotation);
   public void set1Value(int index, SFRotation rotation);
```

```
   public void set1Value(int index, float ax, float ay, float az, float angle);

   public void addValue(ConstSFRotation rotation);
   public void addValue(SFRotation rotation);
   public void addValue(float ax, float ay, float az, float angle);

   public void insertValue(int index, ConstSFRotation rotation);
   public void insertValue(int index, SFRotation rotation);
   public void insertValue(int index, float ax, float ay, float az, float angle);
}

public class MFString extends MField
{
   public MFString(String s[]);

   public void getValue(String s[]);

   public void setValue(String s[]);
   public void setValue(int size, String s[]);
   public void setValue(ConstMFString s);

   public String get1Value(int index);

   public void set1Value(int index, String s);
   public void set1Value(int index, ConstSFString s);
   public void set1Value(int index, SFString s);

   public void addValue(String s);
   public void addValue(ConstSFString s);
   public void addValue(SFString s);

   public void insertValue(int index, String s);
   public void insertValue(int index, ConstSFString s);
   public void insertValue(int index, SFString s);
}

public class MFTime extends MField
{
   public MFTime(double times[]);

   public void getValue(double times[]);

   public void setValue(double times[]);
   public void setValue(int size, double times[]);
   public void setValue(ConstMFTime times);

   public double get1Value(int index);

   public void set1Value(int index, double time);
   public void set1Value(int index, ConstSFTime time);
   public void set1Value(int index, SFTime time);

   public void addValue(double time);
   public void addValue(ConstSFTime time);
   public void addValue(SFTime time);

   public void insertValue(int index, double time);
   public void insertValue(int index, ConstSFTime time);
   public void insertValue(int index, SFTime time);
}
```

```
public class MFVec2f extends MField
{
   public MFVec2f(float vecs[][]);
   public MFVec2f(float vecs[]);
   public MFVec2f(int size, float vecs[]);

   public void getValue(float vecs[][]);
   public void getValue(float vecs[]);

   public void setValue(float vecs[][]);
   public void setValue(int size, vecs[]);
   public void setValue(ConstMFVec2f vecs);

   public void get1Value(int index, float vec[]);
   public void get1Value(int index, SFVec2f vec);

   public void set1Value(int index, float x, float y);
   public void set1Value(int index, ConstSFVec2f vec);
   public void set1Value(int index, SFVec2f vec);

   public void addValue(float x, float y);
   public void addValue(ConstSFVec2f vec);
   public void addValue(SFVec2f vec);

   public void insertValue(int index, float x, float y);
   public void insertValue(int index, ConstSFVec2f vec);
   public void insertValue(int index, SFVec2f vec);
}

public class MFVec3f extends MField
{
   public MFVec3f(float vecs[][]);
   public MFVec3f(float vecs[]);
   public MFVec3f(int size, float vecs[]);

   public void getValue(float vecs[][]);
   public void getValue(float vecs[]);

   public void setValue(float vecs[][]);
   public void setValue(int size, float vecs[]);
   public void setValue(ConstMFVec3f vecs);

   public void get1Value(int index, float vec[]);
   public void get1Value(int index, SFVec3f vec);

   public void set1Value(int index, float x, float y, float z);
   public void set1Value(int index, ConstSFVec3f vec);
   public void set1Value(int index, SFVec3f vec);

   public void addValue(float x, float y, float z);
   public void addValue(ConstSFVec3f vec);
   public void addValue(SFVec3f vec);

   public void insertValue(int index, float x, float y, float z);
   public void insertValue(int index, ConstSFVec3f vec);
   public void insertValue(int index, SFVec3f vec);
}
```

C.8.2.3 *vrml.node* Package

package **vrml.node**;

```
//
// This is the general Node class
//
public abstract class Node extends BaseNode {

  // Get an EventIn by name. Return value is write-only.
  //   Throws an InvalidEventInException if eventInName isn't a valid
  //   event in name for a node of this type.
  public final Field getEventIn(String fieldName);

  // Get an EventOut by name. Return value is read-only.
  //   Throws an InvalidEventOutException if eventOutName isn't a valid
  //   event out name for a node of this type.
  public final ConstField getEventOut(String fieldName);

  // Get an exposed field by name.
  //   Throws an InvalidExposedFieldException if fieldName isn't a valid
  //   exposed field name for a node of this type.
  public final Field getExposedField(String fieldName);
}

//
// This is the general Script class, to be subclassed by all scripts.
// Note that the provided methods allow the script author to explicitly
// throw tailored exceptions in case something goes wrong in the
// script.
//
public abstract class Script extends BaseNode {

  // This method is called before any event is generated
  public void initialize();

  // Get a Field by name.
  //   Throws an InvalidFieldException if fieldName isn't a valid
  //   event in name for a node of this type.
  protected final Field getField(String fieldName);

  // Get an EventOut by name.
  //   Throws an InvalidEventOutException if eventOutName isn't a valid
  //   event out name for a node of this type.
  protected final Field getEventOut(String fieldName);

  // processEvents() is called automatically when the script receives
  //   some set of events. It should not be called directly except by its subclass.
  //   count indicates the number of events delivered.
  public void processEvents(int count, Event events[]);

  // processEvent() is called automatically when the script receives
  // an event.
  public void processEvent(Event event);

  // eventsProcessed() is called after every invocation of processEvents().
  public void eventsProcessed()

  // shutdown() is called when this Script node is deleted.
  public void shutdown();
}
```

C.9 Example of Exception Class

```
public class InvalidEventInException extends IllegalArgumentException
{
   /**
    * Constructs an InvalidEventInException with no detail message.
    */
   public InvalidEventInException(){
      super();
   }
   /**
    * Constructs an InvalidEventInException with the specified detail message.
    * A detail message is a String that describes this particular exception.
    * @param s the detail message
    */
   public InvalidEventInException(String s){
      super(s);
   }
}

public class InvalidEventOutException extends IllegalArgumentException
{
   public InvalidEventOutException(){
      super();
   }
   public InvalidEventOutException(String s){
      super(s);
   }
}

public class InvalidFieldException extends IllegalArgumentException
{
   public InvalidFieldException(){
      super();
   }
   public InvalidFieldException(String s){
      super(s);
   }
}

public class InvalidExposedFieldException extends IllegalArgumentException
{
   public InvalidExposedFieldException(){
      super();
   }
   public InvalidExposedFieldException(String s){
      super(s);
   }
}

public class InvalidVRMLSyntaxException extends Exception
{
   public InvalidVRMLSyntaxException(){
      super();
   }
   public InvalidVRMLSyntaxException(String s){
```

```
      super(s);
   }
}

public class InvalidRouteException extends IllegalArgumentException
{
   public InvalidRouteException(){
      super();
   }
   public InvalidRouteException(String s){
      super(s);
   }
}

public class InvalidNavigationTypeException extends IllegalArgumentException
{
   public InvalidNavigationTypeException(){
      super();
   }
   public InvalidNavigationTypeException(String s){
      super(s);
   }
}

public class InvalidFieldChangeException extends IllegalArgumentException
{
   public InvalidFieldChangeException(){
      super();
   }
   public InvalidFieldChangeException(String s){
      super(s);
   }
}
```

Contact matsuda@arch.sony.co.jp(Kou1 Ma2da), sugino@ssd.sony.co.jp, or honda@arch.sony.co.jp with questions or comments.

This URL: http://vrml.sgi.com/moving-worlds/spec/part1/java.html.

The Virtual Reality Modeling Language

Appendix D. JavaScript Scripting Reference

Version 2.0, ISO/IEC WD 14772

August 4, 1996

This appendix describes the use of JavaScript with the Script node. See "*Concepts - Scripting*" for a general overview of scripting in VRML, and see "*Nodes Reference - Script*" for a description of the Script node.

D.1 Language

Netscape JavaScript was created by Netscape Communications Corporation (http://home.netscape.com). JavaScript is a programmable API that allows cross-platform scripting of events, objects, and actions. A full description of JavaScript can be found at: http://home.netscape.com/comprod/products/navigator/version_2.0/script/script_info/. This appendix describes the use of JavaScript as the scripting language of a Script node.

D.2 Supported Protocol in the Script Node's *url* Field

The url field of the Script node may contain a URL that references JavaScript code:

```
 Script {  url "http://foo.com/myScript.js"  }
```

The javascript: protocol allows the script to be placed inline as follows:

```
    Script {  url "javascript: function foo() { ... }"   }
```

The *url* field may contain multiple URLs and thus reference a remote file or in-line code:

```
    Script {
        url [ "http://foo.com/myScript.js",
              "javascript: function foo() { ... }" ]
    }
```

D.2.1 File Extension

The file extension for JavaScript source code is **.js**.

D.2.2 MIME Type

The MIME type for JavaScript source code is defined as follows:

```
        application/x-javascript
```

D.3 EventIn Handling

Events sent to the Script node are passed to the corresponding JavaScript function in the script. It is necessary to specify the script in the *url* field of the Script node. The function's name is the same as the eventIn and is passed two arguments, the event value and its timestamp (See "*Parameter passing and the EventIn function*"). If there isn't a corresponding JavaScript function in the script, the browser's behavior is undefined.

For example, the following Script node has one eventIn field whose name is *start*:

```
Script {
    eventIn SFBool start
    url "javascript: function start(value, timestamp) { ... }"
}
```

In the above example, when the *start* eventIn is sent the start() function is executed.

D.3.1 Parameter Passing and the EventIn Function

When a Script node receives an eventIn, a corresponding method in the file specified in *url* field of the Script node is called, which has two arguments. The value of the eventIn is passed as the first argument and timestamp of the eventIn is passed as the second argument. The type of the value is the same as the type of the EventIn and the type of the timestamp is **SFTime**. See "*Mapping between JavaScript types and VRML types*" for a description of how VRML types appear in JavaScript.

D.3.2 eventsProcessed() Method

Authors may define a function named *eventsProcessed* which will be called after some set of events has been received. Some implementations will call this function after the return from each EventIn function, while others will call it only after processing a number of EventIn functions. In the latter case an author can improve performance by placing lengthy processing algorithms which do not need to execute for every event received into the *eventsProcessed* function.

Example:

The author needs to compute a complex inverse kinematics operation at each time step of an animation sequence. The sequence is single-stepped using a TouchSensor and button geometry. Normally the author would have an EventIn function execute whenever the button is pressed. This function would increment the time step then run the inverse kinematics algorithm. But this would execute the complex algorithm at every button press and the user could easily get ahead of the algorithm by clicking on the button rapidly. To solve this the EventIn function can be changed to simply increment the time step and the IK algorithm can be moved to an eventsProcessed function. In an efficient implementation the clicks would be queued. When the user clicks quickly the time step would be incremented once for each button click but the complex algorithm will be executed only once. This way the animation sequence will keep up with the user.

The *eventsProcessed* function takes no parameters. Events generated from it are given the timestamp of the last event processed.

D.3.3 initialize() Method

Authors may define a function named *initialize* which is called when the corresponding Script node has been loaded and before any events are processed. This can be used to prepare for processing before events are received, such as construct geometry or initialize external mechanisms.

The *initialize* function takes no parameters. Events generated from it are given the timestamp of when the Script node was loaded.

D.3.3 shutdown() Method

Authors may define a function named *shutdown* which is called when the corresponding Script node is deleted or the world containing the Script node is unloaded or replaced by another world. This can be used to send events informing external mechanisms that the Script node is being deleted so they can clean up files, etc.

The *shutdown* function takes no parameters. Events generated from it are given the timestamp of when the Script node was deleted.

D.4 Accessing Fields

The fields, eventIns and eventOuts of a Script node are accessible from its JavaScript functions. As in all other nodes the fields are accessible only within the Script. The Script's eventIns can be routed to and its eventOuts can be routed from. Another Script node with a pointer to this node can access its eventIns and eventOuts just like any other node.

D.4.1 Accessing Fields and EventOuts of the Script

Fields defined in the Script node are available to the script by using its name. It's value can be read or written. This value is persistent across function calls. EventOuts defined in the script node can also be read. The value is the last value sent.

D.4.2 Accessing Fields and EventOuts of Other Nodes

The script can access any exposedField, eventIn or eventOut of any node to which it has a pointer:

```
DEF SomeNode Transform { }
Script {
    field SFNode node USE SomeNode
    eventIn SFVec3f pos
    directOutput TRUE
    url "...
        function pos(value) {
            node.set_translation = value;
```

```
                }"
        }
```

This sends a set_translation eventIn to the Transform node. An eventIn on a passed node can appear only on the left side of the assignment. An eventOut in the passed node can appear only on the right side, which reads the last value sent out. Fields in the passed node cannot be accessed, but exposedFields can either send an event to the "set_..." eventIn, or read the current value of the "..._changed" eventOut. This follows the routing model of the rest of VRML.

D.4.3 Sending EventOuts

Assigning to an eventOut sends that event at the completion of the currently executing function. This implies that assigning to the eventOut multiple times during one execution of the function still only sends one event and that event is the last value assigned.

D.5 JavaScript Objects

D.5.1 Browser Object

This section lists the functions available in the *browser* object, which allows scripts to get and set browser information. Return values and parameters are shown typed using VRML data types for clarity. For descriptions of the methods, see the Browser Interface topic of the Scripting section of the spec.

Return value	Method Name
SFString	**getName**()
SFString	**getVersion**()
SFFloat	**getCurrentSpeed()**
SFFloat	**getCurrentFrameRate**()
SFString	**getWorldURL**()
void	**replaceWorld**(MFNode nodes)
SFNode	**createVrmlFromString**(SFString vrmlSyntax)
SFNode	**createVrmlFromURL**(MFString url, Node node, SFString event)
void	**addRoute**(SFNode fromNode, SFString fromEventOut, SFNode toNode, SFString toEventIn)
void	**deleteRoute**(SFNode fromNode, SFString fromEventOut, SFNode toNode, SFString toEventIn)
void	**loadURL**(MFString url, MFString parameter)
void	**setDescription**(SFString description)

D.5.2 Mapping between JavaScript types and VRML types

JavaScript has few native types. It has strings, booleans, a numeric type and objects. Objects have members which can be any of the three simple types, a function, or another object. VRML types are mapped into JavaScript by considering MF field types as objects containing one member for each value in the MF field. These are accessed using array dereferencing operations. For instance getting the third member of an MFFloat field named *foo* in JavaScript is done like this:

```
bar = foo[3];
```

After this operation *bar* contains a single numeric value. Note that array indexing in JavaScript starts at index 1.

Simple SF field type map directly into JavaScript. SFString becomes a JavaScript string, SFBool becomes a boolean, and SFInt32 and SFFloat become the numeric type. SF fields with more than one numeric value are considered as objects containing the numeric values of the field. For instance an SFVec3f is an object containing 3 numeric values, accessed using array dereferencing. To access the y component of an SFVec3f named *foo* do this:

```
bar = foo[2];
```

After this operation *bar* contains the y component of vector *foo*.

Accessing an MF field containing a vector is done using double array dereferencing. If foo is now an MFVec3f, accessing the y component of the third value is done like this:

```
bar = foo[3][1];
```

Assigning a JavaScript value to a VRML type (such as when sending an eventOut), performs the appropriate type conversion. Assigning a one dimensional array to an SFField with vector contents (SFVec2f, SFVec3f, SFRotation or SFColor) assigns one element to each component of the vector. If too many elements is passed the trailing values are ignored. If too few are passed the vector is padded with 0's. Assigning a numeric value to an SFInt32 truncates the value.

Assigning a simple value to an MFField converts the single value to a multi-value field with one entry. Assigning an array to an SFField places the first array element into the field. Assigning a one dimensional array to an MFField with vector quantities first translates the array into the the vector quantity then assigns this as a single value to the MFField. For instance if foo is a 4 element array and it is assigned to an MFVec2f, the first 2 elements are converted to an SFVec2f, the last 2 elements are discarded, then the SFVec2f is converted to an MFVec2f with one entry.

Assigning a string value to any numeric type (anything but SFString/MFString) attempts to convert the number to a float then does the assignment. If it does not convert a 0 is assigned. Assigning to an SFTime interprets the value as a double.

Assigning to an SFImage interprets the value as a numeric vector with at least 3 values. The first 2 are the x,y dimensions of the image in pixels, the third value is the number of components in the image (1 for monochrome, 3 for rgb, etc.) and the remaining values are pixel colors as described in " *Fields and Events - SFImage*".

D.6 Example

Here's an example of a Script node which determines whether a given color contains a lot of red. The Script node exposes a color field, an eventIn, and an eventOut:

```
Script {
    field    SFColor currentColor 0 0 0
    eventIn  SFColor colorIn
    eventOut SFBool  isRed

    url "javascript: function colorIn(newColor, ts) {
            // This method is called when a colorIn event is received
            currentColor = newColor;
        }

        function eventsProcessed() {
            if (currentColor[0] >= 0.5)
                // if red is at or above 50%
```

```
                isRed = true;
        }"
}
```

For details on when the methods defined in ExampleScript are called - see the "*Concepts - Execution Model*".

Browser class example

- createVrmlFromString method

```
DEF Example Script {
    field   SFNode myself USE Example
    field   MFString url "foo.wrl"
    eventIn MFNode   nodesLoaded
    eventIn SFBool   trigger_event
    url "javascript: function trigger_event(value, ts){
            // do something and then fetch values
            browser.createVRMLFromURL(url, myself, "nodesLoaded");
        }

        function nodesLoaded(value, timestamp){
            // do something
        }"
}
```

- addRoute method

```
DEF Sensor TouchSensor {}
DEF Baa Script {
    field   SFNode myself USE Baa
    field   SFNode fromNode USE Sensor
    eventIn SFBool clicked
    eventIn SFBool trigger_event
    url "javascript: function trigger_event(eventIn_value){
            // do something and then add routing
            browser.addRoute(fromNode, "isActive", myself, "clicked");
        }

        function clicked(value){
        // do something
   }
}
```

The Virtual Reality Modeling Language Specification

Appendix E. Bibliography

Version 2.0, ISO/IEC WD 14772

August 4, 1996

This appendix contains the informative references in the VRML specification. These are references to unofficial standards or documents. All official standards are referenced in "2. *Normative References*".

[DATA]	**"*Data: URL scheme*", IETF work-in-progress, http://www.internic.net/internet-drafts/draft-masinter-url-data-01.txt].**
[GIF]	**"GRAPHICS INTERCHANGE FORMAT (sm)", Version 89a, which appears in many unofficial places on the WWW, [http://www.radzone.org/tutorials/gif89a.txt , http://www.w3.org/pub/WWW/Graphics/GIF/spec-gif87.txt , "CompuServe at: GO CIS:GRAPHSUP, library 16, "Standards and Specs", GIF89M.TXT, 1/12/95"].**
[FOLE]	**"Computer Graphics: Principles and Practice", Foley, van Dam, Feiner and Hughes.**
[OPEN]	**OpenGL 1.1 specification, Silicon Graphics, Inc., [http://www.sgi.com/Technology/openGL/spec.html].**
[URN]	**Universal Resource Name, IETF work-in-progress, [http://services.bunyip.com:8000/research/ietf/urn-ietf/, http://earth.path.net/mitra/papers/vrml-urn.html].**

Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm.org with questions or comments.

The Virtual Reality Modeling Language Specification

Appendix F. Index

Version 2.0, ISO/IEC WD 14772

August 4, 1996

FALSE
File Syntax and Structure
Group
IS
ImageTexture
IndexedFaceSet
IndexedLineSet
Inline
Lights and Lighting
LOD
Material
MFColor
MFFloat
MFInt32
MFNode
MFRotation
MFString
MFTime
MFVec2f
MFVec3f
MovieTexture
NULL
NavigationInfo
Node Concepts
Nodes, Fields, and Events
Normal
NormalInterpolator
OrientationInterpolator
PROTO
PixelTexture
PlaneSensor
PointLight
PointSet
PositionInterpolator
Prototypes
ProximitySensor
ROUTE
ScalarInterpolator
Script
Scripting
SFBool
SFColor
SFFloat
SFImage
SFInt32
SFNode
SFRotation
SFString

Contact rikk@best.com , cmarrin@sgi.com, or gavin@acm.org with questions or comments.

The Virtual Reality Modeling Language Specification

Credits

Version 2.0, WD ISO/IEC 14772

August 4, 1996

There are many people that have contributed to the VRML 2.0 Specification. We have listed the major contributors below.

Authors

Gavin Bell, gavin@acm.org

Rikk Carey, rikk@best .com

Chris Marrin, cmarrin@sgi.com

Contributors

Ed Allard, eda@sgi.com

Curtis Beeson, curtisb@sgi.com

Geoff Brown, gb@sgi.com

Sam T. Denton, denton@maryville.com

Christopher Fouts, fouts@atlanta.sgi.com

Rich Gossweiler, dr_rich@sgi.com

Jan Hardenbergh, jch@jch.com

Jed Hartman, jed@sgi.com

Jim Helman, jimh@sgi.com

Yasuaki Honda, honda@arch.sony.co.jp

Jim Kent, jkent@sgi.com

Chris Laurel, laurel@dimensionx.com

Rodger Lea, rodger@csl.sony.co.jp

Jeremy Leader, jeremy@worlds.net

Kouichi Matsuda, matsuda@arch.sony.co.jp

Mitra, mitra@earth.path.net

David Mott, mott@best.com

Chet Murphy, cmurphy@modelworks.com

Michael Natkin, mjn@sgi.com

Rick Pasetto, rsp@sgi.com

Bernie Roehl, broehl@sunee.uwaterloo.ca

John Rohlf, jrohlf@sgi.com

Ajay Sreekanth, ajay@cs.berkeley.edu

Paul Strauss, pss@sgi.com

Josie Wernecke, josie@sgi.com

Ben Wing, wing@dimensionx.com

Daniel Woods, woods@sgi.com

Reviewers

Yukio Andoh, andoh@dst.nk-exa.co.jp

Gad Barnea, barnea@easynet.fr

Philippe F. Bertrand, philippe@vizbiz.com

Don Brutzman, brutzman@cs.nps.navy.mil

Sam Chen, sambo@sgi.com

Mik Clarke, RAZ89@DIAL.PIPEX.COM

Justin Couch, jtc@hq.adied.oz.au

Ross Finlayson, raf@tomco.net

Clay Graham, clay@sgi.com

John Gwinner, 75162.514@compuserve.com

Jeremy Leader, jeremy@worlds.net

Braden McDaniel, braden@shadow.net

Tom Meyer, tom@tom.com

Stephanus Mueller, steffel@blacksun.de

Rob Myers, rob@sgi.com

Alan Norton, norton@sgi.com

Tony Parisi, tparisi@intervista.com

Mark Pesce, mpesce@netcom.com

Scott S. Ross, ssross@fedex.com

Hugh Steele, hughs@virtuality.com

Dave Story, story@sgi.com

Helga Thorvaldsdottir, helga@sgi.com

Harrison Ulrich, hrulrich@conline.com

Chee Yu, chee@netgravity.com

The entire VRML community, www-vrml@wired.com

Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm,org with questions or comments.

This URL: http://vrml.sgi.com/moving-worlds/spec/credits.html