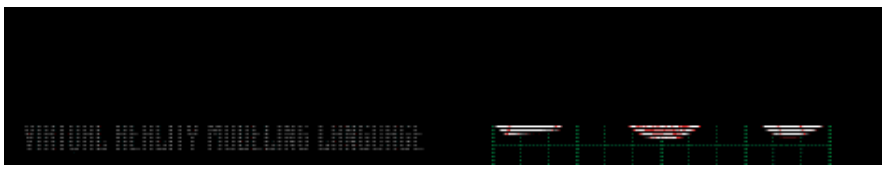


The Virtual Reality Modeling Language Specification

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

The VRML 2.0 Specification

Overview Design Notes Schedule

Official VRML 2.0, Draft 3:

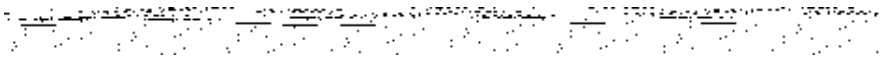
- Compressed (gzip) Postscript (588K)
- Compressed (gzip) HTML (140K)
- Uncompressed HTML (532K)
- Compressed (gzip) tar HTML directory (956K)

The Virtual Reality Modeling Language (VRML) specification, version 2.0, is currently under development. It is scheduled for final specification on August 4, 1996. Note that since the specification is not in its final form and is subject to change, any products or books based on official VRML 2.0 are also subject to change. **Any excerpts or copies of this specification MUST include a readable "Draft #3" on every page.**

The first draft of VRML 2.0 was released in May - files compliant with this unofficial release must contain the following header: **`#VRML Draft #1 V2.0 utf8`**. The second draft was released in June - files compliant with this unofficial release must contain the following header: **`#VRML Draft #2 V2.0 utf8`**. The official VRML header, **`#VRML V2.0 utf8`**, is not valid until the final specification is released in August. The third draft is official on Monday, July 15, 1996. The final specification is scheduled to release on August 4th, 1996.

The specification was orginally developed by Silicon Graphics 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 is a tribute to the successful collaboration of all of the members of the VRML community. Gavin Bell, Chris Marrin, and Rikk Carey have headed the effort at SGI to produce the final specification.

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



What is VRML?

VRML is an acronym for Virtual Reality Modeling Lanaguage. It is a file format for describing 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., reviewed and improved by the VRML email discussion group (www-vrml@wired.com), and later adopted and endorsed by a plethora of companies and individuals. See the San Diego Supercomputing's VRML Repository for lots of information on VRML, or see SGI's VRML site.

What is Moving Worlds?

Moving Worlds is the name of the proposal that was chosen by the VRML community as the working document for VRML 2.0. It was created by Silicon Graphics, 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. It is also intended for readers interested in simply learning more about VRML. Note however that many people (especially non-technical) find the VRML Specification inadequate as a starting point or as a primer for users. However, 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. See San Diego Supercomputer Center's list of browsers for what's available. Note however that since VRML 2.0 is still a **working document**, these browsers are in

a beta phase. At this point, Sony's CyberPassage and Silicon Graphics' Cosmo Player for Windows95 are the only released (beta) browsers supporting **VRML 2.0 Draft #2.**

Related Documents

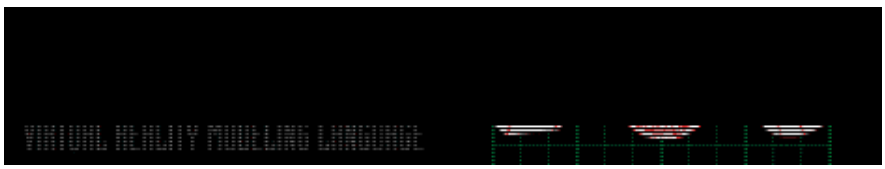
- VRML 1.0 Specification
- VRML 2.0: Request-for-Proposal from the VAG
- VRML 2.0: Process from the VAG
- VRML 2.0: Polling Results

Related Sites

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



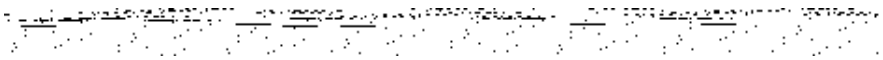
An Overview of the

Virtual Reality Modeling Language

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

- Introduction
- Summary of VRML 2.0 Features
- Changes from VRML 1.0



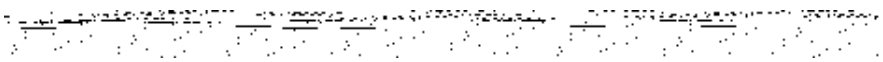
Introduction

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 official 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.

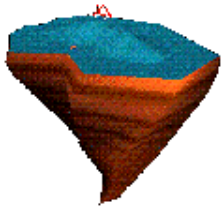
Prototyping

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

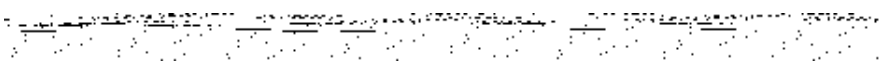
fish, the world builder could just plug in new geometry or behavior.

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 old Separator node.

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 global lighting attributes such as ambient lighting, light attenuation, and fog.

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 cylinder.

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

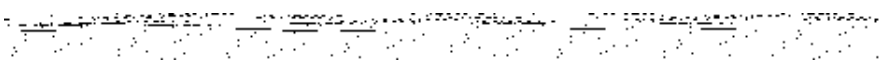
The following VRML 1.0 node types have been removed from VRML 2.0:

- 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

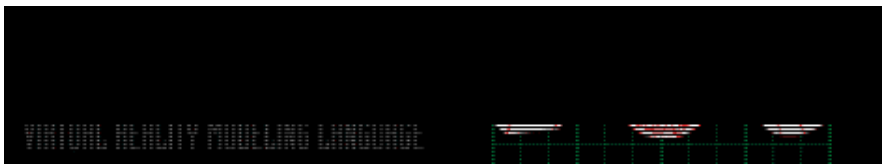
In addition to all of the other changes, VRML 2.0 introduces a couple of new field types:

- 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 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, Official Draft #3, ISO/IEC 14772

July 15, 1996

This document is the third draft of the complete specification **of the Virtual Reality Modeling Language**, (VRML), version 2.0 specification. The *Introduction* section describes the conventions used in the specification, *Key Concepts* describes various fundamentals of VRML 2.0, *Node Reference* provides a precise definition of the syntax and semantics of each node, *Field Reference* defines the datatype primitives used by nodes, *Grammar* presents the BNF. There are two appendices that describe the integration of Java and JavaScript with VRML. 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.

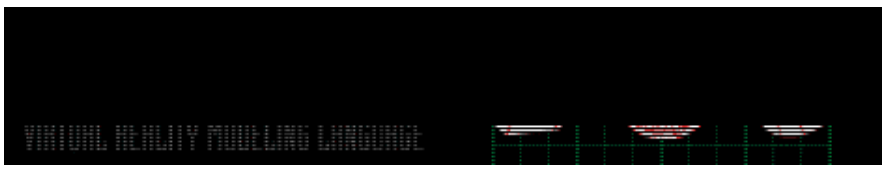
Part 1:

Foreword	1 Scope	A Grammar
Introduction	2 References	B External API
	3 Glossary	C Examples
	4 Concepts	D Java
	5 Nodes	E JavaScript
	6 Fields	F Bibliography
	7 Conformance	G Index

Document change log Credits

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

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



Virtual Reality Modeling Language Specification

Document Change Log

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

7/15/96 (rc):

- **Publish official Draft #3.**

7/12-14/96 (rc):

- Fixed lots of links.
- Lots of minor error correctsions and clarifications.

7/11/96 (rc):

- Released unofficial draft of Draft #3.
- Combined all examples into the spec/part1/examples.html section.
- Changed **Switch**'s *whichChild* to *whichChoice* to be consistent with *choice* field naming.
- Added "+" and "-" to illegal first characters for names.
- Clarified **PlaneSensor** to default to the local XY plane (the text had refs to both XY and XZ).
- Fixed lots of typos and minor errors.
- Added links/ref from nodesRef.html to concepts.html on Grouping nodes (*addChildren* et al fields) and Bounding boxes (*bbox** fields), and removed redundant explanations.
- Changed ***Interpolator**'s *value* exposedField to *keyValue*. This avoid the confusion that the *value_changed* eventOut is tied directly to the *value* exposedField.

7/10/96 (rc):

- Added Gavin Bell's execution model section to concepts.html. This reordered the section and impacted Scripting and Time sections.

7/9/96 (rc):

- Added/updated missing/old sections (java, conformance, etc.).
- Clarified that DEF, USE, PROTO, EXTERNPROTO, TRUE, FALSE, NULL, eventIn, eventOut, field, exposedField, IS, and TO are illegal names (concepts.html).
- Sony: Java.html: Remove base node class which implements Node.

7/8/96 (Sony) (to Java Reference):

- Add base node classes which implement Node.

7/7/96 (rc):

- **Numerous** style and structure changes as per ISO meeting.
- Changed SFImage (and thus **PixelTexture**) to use RGBA (A = alpha), rather than RGBT (T=transparency). This change was motivated by the fact that most image formats and rendering libraries specify in terms of alpha or opacity, not transparency.
- Created new sections as per ISO (empty for now).

- Added section numbers to Concepts section.
- Moved long examples out of Concepts into examples.html.

7/3/96 (rc): **(note: many of these changes were part of cm's pre-nuptial spree, the rest were undo's of cm's changes...rc)**

- **Viewpoint** *description* field was restored to non-exposed as per Draft 2b. Since this field is a hint to the browser, it may not have any effect and thus should not be encouraged to be changed.
- **TouchSensor** *hit** eventOuts where changed to be *hit*_changed* as per naming convention.
- **Shape** *appearance* and *geometry* fields were exposed.
- **PointLight** and **SpotLight** default *radius* changed to 100 meters (from 1m).
- **NavigationInfo**'s *avatarSize* field defaults were changed to a more typical avatar size (from a radius of 1 meter to an avatar with a radius of 0.25m, eye height of 1.6m, and maximum step height of 0.75m). Also, the *bind_changed* event out was changed to *isBound* as per the field naming exceptions.
- *isActive* eventOut was added to **MovieTexture** to be consistent with the **AudioClip** node.
- Added *set_*index* eventIns to **IndexedFaceSet** and **IndexedLineSet**. This enables the geometry to be changed from a script.
- **FontStyle** fields were restored to not exposed to keep the implementation easier and since multiple FontStyles can give most of the effect needed.
- **Fog** was changed as follows: the *size* field was removed, **Fog** is now a bindable node (thus only one is active at a time and no need for *size*), *fogType* was added for more fog effects, and *set_bind* and *isBound* events were added to allow binding. This change was made mostly to make the implementation easier (multiple Fog nodes with limits was extremely difficult to implement). Default *visibilityRange* changed from 1 to 1000. The two values of fogType are: LINEAR and EXPONENTIAL (originally the second type was named "FOG" and thus not very clear). ;)
- **CylinderSensor**, **PlaneSensor**, and **SphereSensor** had *offset* and *autoOffset* exposed fields added - this fixes a major problem with these sensors. See concepts.html#DragSensors for for a summary and see nodesRef.html for specifics per node.
- **Background**'s *bind_changed* changed to *isBound* as per style exception in Concepts.
- Clarified field naming exceptions in Concepts (see /concepts.html#FileSyntaxandStructure).
- All events with the name **Time_changed* that represent actual timestamp had the *_changed* suffix removed, as per the style exception defined in Draft 2. This was done for readability and comprehension.
- All *bbox_size* fields defaults changed from (0,0,0) to (-1,-1,-1). Where (-1,-1,-1) means unspecified by the user (and thus up to the browser to compute), and (0,0,0) defines an infinitely small, (i.e. point), bbox. Added section to Key Concepts on bboxes (.../spec/concepts.html#BoundingBoxes).
- Removed "_" from add_children and remove_children fields of group nodes.
- *duration* eventOut on **AudioClip** was changed to *duration_changed.*
- Decided not to pluralize the MF fields - made all fields singular (w/ exception of children).

7/2/96 (rc):

- Updated to Working Draft #3 titles and date. Most of these are described in the 7/3 log.

7/5/96 (Sony) (to Java Reference):

- The return value of createVrmlFromURL() changes from Node to void.
- The return value of createVrmlFromString() changes from Node to Node[].
- Add 'createVrmlFromURL' and 'addRoute' example(thanks to Justin Couch (jtc@hq.adied.oz.au))

7/3/96 (Sony) (to Java Reference):

- Change from 'interface Node' to 'class Node'.
- Inline script part is deleted.(thanks to Mik Clark(raz89@dial.pipex.com))

7/1/96 (Sony) (to Java Reference):

- Add 'File extension' and 'MIME type' section.
- Add 'Exception' classes example(thanks to Gad Barnea(barnea@easynet.fr)).

6/27/96 (Sony) (to Java Reference):

- The return value's type of **Node**'s **getValue()** method changed from 'ConstField' to '**Field**'.
- Add '**shutdown**()' method to '**Script**' class.
- Add '**getWorldTitle**()' and '**setWorldTitle**()' methods to '**Browser**' class.
- Add the description on how to resolve user-defined classes(thanks to Mik Clarke (RAZ89@DIAL.PIPEX.COM)).

6/14-24/96 (cfm):

- Lots of undocumented changes. **NOTE: I have attempted to document then in later log messages above. (7/2/96...rc).**

6/14/96 (cfm):

- Updated External API to reflect discussions.
- Offer HTML, compressed HTML and compressed Postscript versions of spec.

6/5/96 (cfm):

- Added first draft of External API and externalBindings (for review).
- Added Script voting page

6/4/96 (rc):

- Released Draft #2b.
- Reinstated the *Java Reference* section (added link to front page).
- Moved the *VRMLScript Reference* section to an equal level to the *Java Reference*.
- Removed any statements REQUIRING a scripting language (until further notice).
- Added link to the External API section (TBD).

5/30/96 (rc):

- **Released Draft #2.**

5/29-30/96 (rc):

- Renamed **ElevationGrid** fields: *verticesPerColumn/Row* to *x/yDimension*, renamed *gridSize* to *x/ySpacing* - due to many requests and inquiries.
- Various document fixes and final cleanup, spell-check, and details.
- *scriptType* was removed from the **Script** node - use the data: or vrmlscript: syntax.

5/28/96 (rc):

- Added *bboxCenter, bboxSize, add_children* and *remove_children* eventIns to **Anchor, Billboard,** and **Collision.**
- **Background**'s *sky/groundRange* field renamed to *sky/groundAngle* for clarity.
- Daniel Woods re-wrote the **Interpolators** sections in **concepts.html** and **nodesRef.html (all Interpolators**) - mostly clarifications and improvements.

5/21-27/96 (rc):

- **Background** URL fields changed from *pos/neg/xyz* to *backUrl, bottonUrl, frontUrl, leftUrl, rightUrl*, and *topUrl* to remain consistent with keeping *url* in all URL field names.
- Browser camera is child of the current **Viewpoint** (at all times). Current **NavigationInfo** is automatically parented to current **Viewpoint** too.
- Added *jump* exposedField to **Viewpoint** node - if TRUE, then when binding to **Viewpoint**, browser must move current view to this **Viewpoint**.
- Binding rules for **Background**, **NavigationInfo**, and **Viewpoint** (see concepts) were clarified - multiple *set_bind* of TRUE simply move the node to top of stack (not added twice), while *set_bind* FALSE of an unbound node have no effect.
- For nested hierarchies of **ProxitySensors** or **VisibilitySensors**, all sensors output. If instanced, the user the union of the boxes to test against.
- Various clarifications and consistency improvements to **TouchSensor, CylinderSensor, PlaneSensor**, and **SphereSensor.**
- Merged **DiskSensor** into **CylinderSensor** - one sensor can do both.
- For nested hierarchies of **TouchSensor/CylinderSensor/PlaneSensor/SphereSensor,** the lowest node outputs. If siblings, then each one outputs. If instanced, then check each instance every test.
- For nested hierarchies of **Collisions,** the nearest collision and lowest node outputs. If instanced, then check each instance every test.
- Changed all eventIns and eventOuts to use the recommended naming syntax: set_foo and foo_changed (exceptions for Bool and Time eventOuts - use isFoo for Bool and fooTime for Time).
- Spec does not comment on whether Examine view mode (or any mode for that matter) performs collision detection - this left up to the browsers.
- **ProximitySensors** do send pos/orient outputs when viewer exits.
- **LineSet** and **PointSet** nodes do not perform collisions (since they have no area).

- Clarified sensors edge conditions - browser needs to handle this.
- Significant changes to **TimeSensor** - defaults have changed and evaluation has been clarified and changed.
- Changed *bind* and *isBound* to *set_bind* and *bind_changed* in **Background**, **NavigationInfo**, and **Viewpoint** nodes to be consistent with naming recommendations.
- Clarified **TouchSensor** to output *hit** events when pointer moves regardless if isActive is TRUE -- this enables drag-n-drop actions.
- Moved file syntax defs to top of node sections in Node Reference.
- **NavigationInfo** - recommend that near plane = 1/4 avatar size.
- Re-wrote **VisibilitySensor** section - clarifications only.
- Changed **VisibilitySensor's** bboxC*enter* and bboxS*ize* exposedFields to *center* and *size* in order to clarify that these boxes are not necessarily bounding boxes and to make more consistent with **ProximitySensor**'s box specs.
- Added the missing *isActive* field to **PlaneSensor**.
- Replaced **TimeSensor**'s *discrete* field with *cycleTime* eventOut - much simpler.
- Re-wrote **TimeSensor** and **AudioClip** for clarity.
- Removed **VisibilitySensor**'s *isActive* field and added the field *enabled.*
- Spell check on NodeRef.html.
- Moved sections at the top of *Node Reference* (e.g. Lights and Lighting) to *Key Concepts*.
- A variety of minor style changes and clarifications to *Node Reference* and *Key Concepts*.

5/9/96 (rc):

- Improved **TimeSensor** section
- Miscellaneous clarifications

4/19/96 (rc):

- Clarified ideal spatial **Sound** implementation
- **ProximitySensor** bbox size of (0 0 0) no longer uses the parent's bbox - it disables the sensor.
- Changed all url fields to be exposedFields (**Anchor**, **AudioClip**, **Inline**, ***Texture**, **Script**)
- Removed *numPoints* from **PointSet.**

4/18/96 (rc):

- **Published Draft #1**:

`http://vag.vrml.org/VRML2.0/DRAFT1/spec.main.html`

4/17/96 (rc):

- Changed comma to be a whitespace character
- Added *parameters* field to **Anchor**
- Changed **Anchor** fields to be exposed

4/12/96 (cm):

- Fixed lots of links and caught some instances of old node names
- Fixed error in **Anchor** example caught by mott
- Got rid of duplicate **FontStyle** description caught by jkent
- Got rid of image field in **ImageTexture** node
- Fixed error in **NavigationInfo** of misnamed and duplicated avatarSize field

4/2/96 (rc):

- Changed **ClickSensor** to **TouchSensor** (as per Denton, almost)
- Major update/rewrite **Text** and **FontStyle** nodes and description (thanx to Jan H.).
- Redesigned texture nodes: renamed **Texture2** to **ImageTexture**, created **MovieTexture**, and created **PixelTexture** (no URL, just pixels)
- Changed **Fog**'s maximumVisibility to visibilityRange
- Changed **BoxProximitySensor** to **ProximitySensor** (since there's only one now)
- Removed hit* fields and enabled field from **PlaneSensor**
- Changed all URL fields (e.g. filename) to 'url' - let's face it these are URLs, not files (e.g. data:...").
- Removed **WWW** prefix from all nodes . Too many nodes now have URLs in them - prefer to keep the names simple and clear.
- Made the following changes to simplify: **Coordinate3** -> **Coordinate**, **Texture2Transform** -> **TextureTransform**, **TextureCoordinate2** -> **TextureCoordinate.**
- Changed **Cube** to **Box** (to be more geometrically correct), and merged fields into a single size Vec3f. (Note that Sphere is a sphere since it is defined by a radius, not 3 radii.)
- Minor **Fog** re-write and field name change
- Added bind and isBound fields to **NavInfo**, **Viewpoint**, **Background**, and wrote the section **Bindable Leaf Nodes**.
- Added clickTime to **TimeSensor** and re-defined time as absolute (not relative)

3/24/96 (rc):

- Re-organized and alphabetized nodes in Node Ref
- Added **Billboard**
- Replaced **PointSound** and **DirectedSound** with **Sound** and **AudioClip**
- Changed/updated **GeneralCylinder** to **Extrusion**

3/23/96 (rc):

- Texture2Transform fields exposed
- Disallowed exposed fields in Script nodes
- ColorInterpolate's "eventOut outValue" changed from MF to SFColor to match the Material node
- Re-wrote the Interpolator's section - added more detail
- Lots of minor re-org to improve readability
- Added WWWMovieTexture and ImageTexture
- Changed Texture2 to WWWTexture2 - removed image field

3/5/96(cm): Removed voting booth. Added BNF syntax section to spec. Misc. edits to spec.

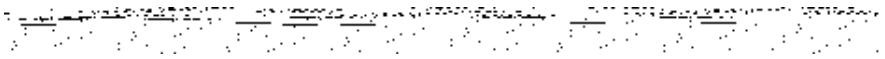
1/24/96(cm): Added questions to the voting booth.

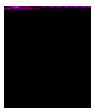
1/24/96(cm): More spec refinements, more logos and a Sample Software section.

1/23/96(cm): Added API and major spec update.

1/16/96(cm): First public version.

1/8/96 (gb): Created.

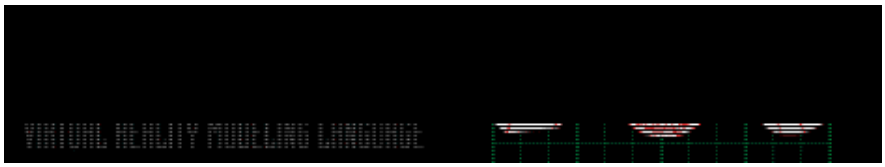




Contact rikk@best.com, cmarrin@sgi.com, or gavin@acm,org with questions or comments.

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

---



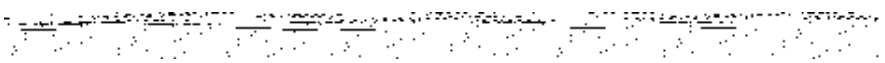
The Virtual Reality Modeling Language Specification

Credits

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

There are many people that have contributed to the VRML 2.0 Specification. We have listed the major contributors below. Please let us know if we left someone out who deserves to be on this list (e.g. you).



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

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

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

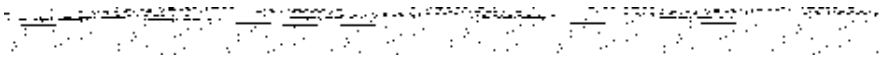
Scott S. Ross, ssross@fedex.com

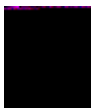
Hugh Steele, hughs@virtuality.com

Helga Thorvaldsdottir, helga@sgi.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

---

The Virtual Reality Modeling Language

Foreword

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 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.

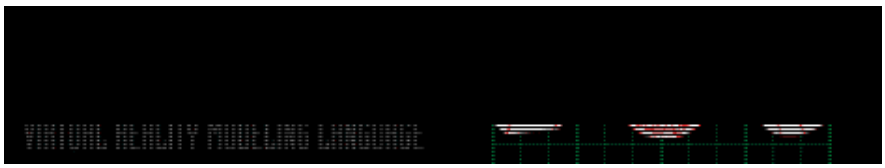
International Standard ISO/IEC 14772 was prepared by Joint Technical Committee ISO/IEC JTC 1, Information Technology Sub-Committee 24, Computer Graphics and Image Processing in collaboration with the VRML Architecture Group (VAG).

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

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/foreword.html

---

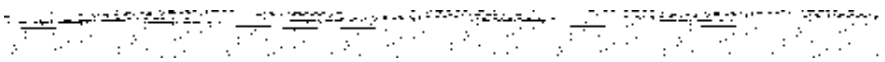


The Virtual Reality Modeling Language

Introduction

Version 2.0, Official Draft #3, ISO #14772

July 15, 1996



Purpose

The Virtual Reality Modelling 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.

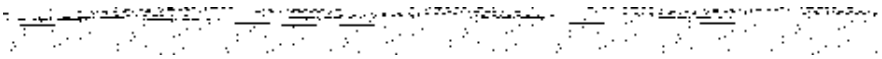
Characteristics of VRML

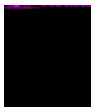
VRML is capable of representing static and animated objects and it can have hyperlinks to other media such as sound, movies, and image. Interpreters (browsers) for VRML are widely available for many different platforms as well as authoring tools for the creation VRML files.

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*.





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/introduction.html.

The Virtual Reality Modeling Language

1. Scope and Field of Application

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 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 modelling 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. Normative References

Version 2.0, Official Draft #3, ISO/IEC 14772

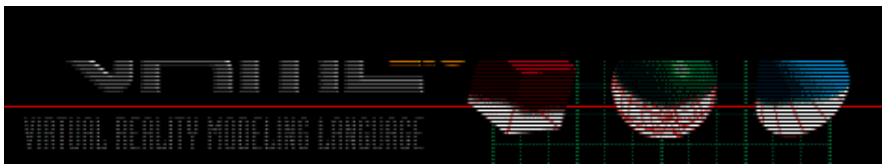
July 15, 1996

TBD - This section will contain all normative references to official standards.

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/references.html

---



The Virtual Reality Modeling Language Specification

3. Glossary

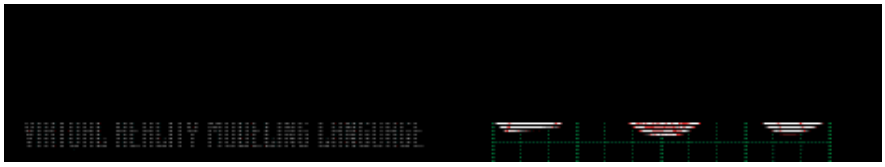
Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

TBD - This section will contain a glossary of terms for the VRML spec.

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/glossary.html



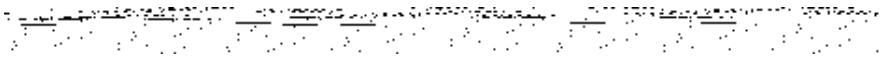
The Virtual Reality Modeling Language Specification

4. Concepts

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

This section describes key concepts related to the definition and use of the VRML 2.0 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 topics on nodes.



4.1 File Syntax and Structure

4.1.1 Syntax Basics

4.1.2 File Syntax vs. Public Interface

4.1.3 URLS and URNs

4.1.4 File Extension and MIME Types

4.2 Nodes, Fields, and Events

4.2.1 Introduction

4.2.2 General Node Characteristics

4.3 The Structure of the Scene Graph

4.3.1 Grouping Nodes and Leaves

4.3.2 Instancing

4.3.3 Coordinate Systems and Transformations

4.3.4 Viewing Model

4.3.5 Bounding Boxes

4.4 Events

4.4.1 Routes

4.4.2 Sensors

4.4.3 Execution Model

4.4.4 Loops

4.4.5 Fan-in

4.5 Time

4.5.1 Introduction

4.5.1 Discrete and continuous changes

4.6 Prototypes

4.6.1 Introduction to Prototypes

4.6.2 Defining Prototypes in External files

4.7 Scripting

4.7.1 Introduction

4.7.2 Script execution

4.7.3 initialize

4.7.4 eventsProcessed

4.7.5 Scripts with direct outputs

4.7.6 Asynchronous scripts

4.7.7 Script Languages

4.7.8 Receiving and Sending Events

4.7.9 Browser Script Interface

4.8 Browser Extensions

4.8.1 Creating Extensions

4.8.2 Reading Extensions

4.9 Node concepts

4.9.1 Bindable leaf nodes

4.9.2 Geometry

4.9.3 Grouping nodes

4.9.4 Interpolators

4.9.5 Lights and Lighting

4.9.6 Sensors

4.1 File Syntax and Structure

4.1.1 Syntax Basics

For easy identification of VRML files, every VRML 2.0 file based on this draft specification 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 the following:

- Any number of prototypes defined using PROTO or EXTERNPROTO.
- Any number of nodes.
- Any number of ROUTE statements.

See the Syntax Reference section for more details.

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**						

For example, each line of the following has various errors (in italics):

```
PROTO [ field field eventIn eventOut field ROUTE ] { ... }
DEF PROTO Transform {
    children [
        DEF USE Shape { geometry DEF EXTERNPROTO ... }
        USE DEF
    ]
}
```

4.1.2 File syntax vs. public interface syntax

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, 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*).
- All eventIns have the prefix "*set_"* - with the exception of the *addChildren* and *removeChildren* eventIns.
- All eventOuts have the suffix "*_changed"* appended - with the excepttion 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 (in Script and PROTO nodes) are not required to follow these rules but doing so would improve the consistency and readability of the file.

4.1.3 URLs and URNs

Issue: This section needs to be tightened up and clarified, esp. wrt URNs.

A URL (Universal Resource Locator) specifies a file located on a particular server and accessed through

a specified protocol. A URN (Universal Resource Name) provides a more persistent way to refer to data than is provided by a URL. The exact definition of a URN is currently under debate. See the discussion at http://www.w3.org/hypertext/WWW/Addressing/Addressing.html for further details.

All URL/URN fields are of type MFString. The strings in such a field indicate multiple places to look for files, in decreasing order of preference. If the browser can't locate the first file or doesn't know how to deal with the URL or URN, it can try the second location, and so on.

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. URN support is specified in a separate document at http://earth.path.net/mitra/papers/vrml-urn.html, which may undergo minor revisions to keep it in line with parallel work happening at the IETF.

Relative URLs are handled as described in IETF RFC 1808, "Relative Uniform Resource Locators."

Data Protocol

Issue: Is this required or even advisable? I think it should be removed from the spec until data: is finalized...rc

Data can be included directly in the VRML file using the *data:* protocol. The *data:* protocol is described at http://www.acl.lanl.gov/HTML_WG/html-wg-96q1.messages/0599.html. It allows inclusion of binary data in base64 encoding. In this way, for instance, JPEG images can be included inline, taking advantage of the image compression offered by JPEG while avoiding multiple fetch requests over the network.

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
          "javabc: ...",                   # custom protocol Java byte
          "java: ...",                     # custom protocol Java src
          "http://bar.com/foo.javascript", # std protocol JavaScript
          "http://bar.com/foo.class",      # std protocol Java byte
          "http://bar.com/foo.java" ]      # std protocol Java src
}
```

Issue: Sony guys needs to verify that this is ok.

4.1.4 File Extension and Mime Type

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

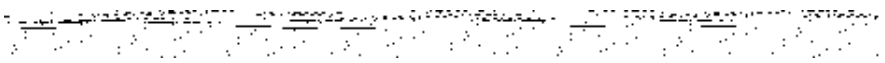
The MIME type for VRML files is defined as follows:

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

where the MIME major type for 3D world descriptions is `x-world`, and the minor type for VRML documents is `x-vrml`.

It is anticipated that the official type will change to "model/vrml". At this time, servers should present files as being of type x-world/x-vrml. Browsers should recognise both x-world/x-vrml and model/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).



4.2 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 and interactive object/world building. These objects are called *nodes*, and contain data which is stored in *fields*.

4.2.2 General Node Characteristics

A node has the following characteristics:

- **A type name -** This is a name like Box, 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 is not important; for example, "Cone { bottomRadius 1 height 6 }" and "Cone { height 6 bottomRadius 1}" are equivalent.
- **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 can 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:

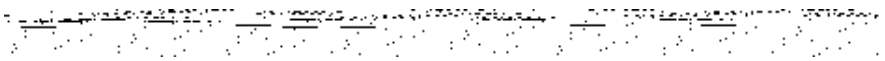
```
eventIn set_foo
field foo
eventOut foo_changed
```

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

The syntax for representing these pieces of information 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 Nodes and Leaves

A scene graph consists of grouping nodes and leaf nodes. Grouping nodes, such as Transform, LOD, and Switch, can have child nodes. These children can be other grouping nodes or leaf nodes, such as shapes, browser information nodes, lights, viewpoints, and sounds. Appearance, appearance properties, geometry, and geometric properties are contained within Shape nodes.

Transformations are stored within Transform nodes. Each Transform node defines a coordinate space for its children. This coordinate space is relative to the parent (Transform) node’s coordinate space--that is, transformation accumulate down the scene graph hierarchy.

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 gives a node a name and creates a the 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 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.

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

VRML 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 inherit 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.)

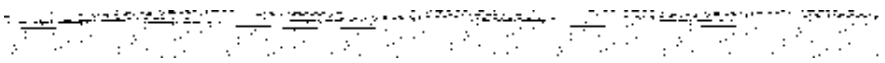
4.3.4 Viewing Model

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 Leaf 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.

4.3.5 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 boundings boxes (-1, -1, -1) implies that the user did not specify the bounding box and the browser must compute it on-the-fly or assume the most conservative case. A bounding box size of (0, 0, 0) is valid and represents a point in space (i.e. infinitely small box).

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 other nodes or to alert monitoring nodes that some state has changed within the source node. Nodes can also have exposedField definitions which bundle an eventIn, a field, and an eventOut. For example, the Transform node has an *translation* exposedField that can receive *translation* events that change the *translation* field, and then send a *translation* output event.

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 can 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: Bool 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 all 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.

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, whcih 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 VisiblitySensor is visible, and a VisibilitySensor that is NOT visible if a TouchSensor is active).

4.4.3 Execution Model

Once a Sensor has generated an **initial event**, the event is propogated 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).

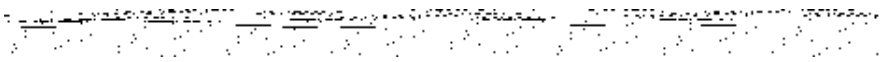
Some sensors generate multiple events simultaneously; in these cases, each event generated initiates a different event cascade.

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

Fan-in occurs when two routes lead 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.



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 12 midnight GMT January 1, 1970.

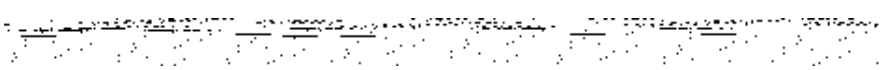
Events that are "in the past" cannot be generated; processing an event with timestamp ’t’ may only result in generating events with timestamps greater than or equal to t.

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 generatedy 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 occuring 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, behaviors, or both.

A prototype definition consists of the following:

- the PROTO keyword
- the name of the new node type
- a list of events the new node type can send and receive
- a list of fields that the new node type makes publicly accessible, with default values
- the definition of the new node type. This definition is one or more scene graphs composed of existing nodes (built-in nodes or previously defined prototypes) and uses the IS keyword to substitute the new field and event names for the built-in names.

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 Scene graph(s)
  (nodes, prototypes, and routes, containing IS statements)
}
```

A prototype does **not** define a node into the scene graph; it merely 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.

The first scene graph, (referred to as the *primary scene graph*), found in the prototype definition containing the IS syntax is used to represent this node. The other scene graphs are not rendered, but may be referenced via routes or scripts and thus cannot be ignored.

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

The *eventIn* and *eventOut* declarations export events from the primary scene graph. Specifying each event's type both in the prototype declaration and in the primary scene graph is intended to prevent errors and to provide consistency with external prototypes.

Events generated or received by nodes in the prototype's implementation are associated with the prototype using the keyword **IS**. For example, the following statement exposes a Transform node's built-in *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 hold the persistent state of VRML objects. Allowing a prototype to export fields allows the initial state of a prototyped object to be specified when an instance of the prototype is created. The fields of the prototype are associated with fields in the implementation using the **IS** keyword. For example:

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

IS statements may appear inside nodes wherever fields may appear. Specifying an IS statement for a node in the primary scene graph which is not part of the prototype's implementation is an error. Inversely, it is also an error for an IS statement to refer to a non-existant declaration. It is an error if the type of the field or event being exposed does not match the type declared in the prototype's interface declaration.

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**

It is valid to specify both the field (or exposedField) default values and the IS association inside a prototype definition. For example, the following prototype maps a Material node's *diffuseColor* (exposedField) to the prototype'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
    }
}
```

A prototype is instantiated as if *typename* 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 { ... }
  ...
}
```

can be instantiated wherever Box can be used, since the first node of the prototype's primary scene graph is a Box node.

A prototype's scene graph defines a DEF/USE name scope separate from the rest of the scene; nodes DEF'ed inside the prototype may not be USE'ed 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 {}
}
```

4.6.2 Defining Prototypes in External Files

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

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

The external prototype is then given the name *prototypename* 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.

To allow the creation of libraries of small, reusable PROTO definitions, browsers shall recognize

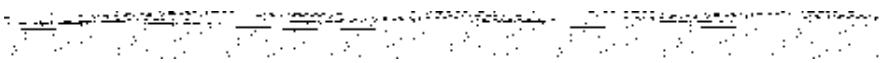
EXTERNPROTO URLs that end with "*#name*" to mean the prototype definition of "name" in the given file. For example, a library of standard materials might be stored in a file called "materials.wrl" that looks like:

```
#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

Issue: This section needs to be tightened up. Specically, the Browser Interface to Script nodes needs to be defined in a language-neutral manner, and it needs to be a comprehensive definition of the required semantics of any scripting implementation. This will occur before the final spec (obviously)...rc

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 that receive events in, process them, and generate other events. A Script node can also keep track of information between invocations, (i.e. managing internal state over time).

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 arbitrary logic into the middle of an event cascade. They 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 a given 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

Some scripting language bindings for VRML may define an initialization method (or constructor or whatever). This method must be called before any events are generated. Any events generated by the initialize method must have timestamps less than any other events that are generated by the Script node.

4.7.4 eventsProcessed

The scripting language binding may also define an eventsProcessed routine that is called after some set of events has been 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 whenver events are received. If it is used in some other way, eventsProcessed can be indeterministic, 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 indeterministic 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

Scripts can be written in any language supported by the browser. The instructions for the script are referenced by the *url* field. This field may contain a URL which points to data on a server. 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 custom language protocol defined for the specific language (in which the language type is inferred).

ISSUE: Gavin suggests that we add a subsection "Time during Script execution" here.....rc

4.7.8 Receiving and Sending Events

ISSUE: This section is needed to define the semantics of get/set that ALL langs must support...rc

4.7.9 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 VRMLfield 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 loadWorld(MFString url);

The **loadWorld()** method loads one of the URLs in the passed string and replaces the current scene root with the VRML file loaded. The browser first attempts to load the first URL in the list; if that fails, it tries the next one, and so on until a valid URL is found or the end of list is reached. If a URL cannot be loaded, some browser-specific mechanism is used to notify the user. Implementations may either block on a **loadWorld()** until the new URL finishes loading, or may return immediately and at some later time (when the load operation has finished) replace the current scene with the new one.

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.

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()** asks the browser to load a VRML scene description from the given URL or URLs. After the scene is parsed *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 A wants to create a Torus geometry node:

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

Browser A 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 stricly 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 NodeConcepts

4.9.1 Bindable Leaf 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. The browser maintains

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 push and pop a given node from its respective stack. A TRUE value sent to *set_bind*, pushes the node to the top of the stack, and FALSE pops it from the stack. The bind_changed event is output when a given node's binding state changes (i.e. whenver *set_bind* is received). 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 state. If an already bound node receives a set_bind TRUE event, then that node is moved to the top of the stack - there are not two entries for this node in the stack. If a node that is not bound receives a set_bind FALSE event, the event has no effect. If there are no bound nodes on the stack, then use the default values for each node.

Issue: Can the bind stack be popped to empty? (rc answer: yes) If so, what is the view? (rc answer:t he view is either at the origin or no-change, regardless of jump value.) Also, what is the view, if the file has zero Viewpoints? (probably undefined, up to browser) Does the isBound send when the node is pushed to top-of-stack and vice versa, OR when it is pushed onto the stack and out of the stack (these are very different behaviors)? (fouts answer: use top-of-stack behavior.)

Bind Stack Behavior

- During read, the first encountered *<binding node>* is bound by pushing it to the top of the *<binding node>* binding stack and an isB*ound* event issued.
- When a *set_bind* TRUE eventIn is received by a *<binding node>* it is pushed to the top of the binding stack and becomes the currently bound *<binding node>*, (and the *isBound* eventOut sends TRUE). However, if the node is already bound, then it is moved to the top of the stack and *isBound* does not send an event.
- When a *set_bind* FALSE eventIn is received by the *<binding node>* on the top of the stack, it is popped from the stack, it sends a *isBound* eventOut FALSE, and the next node in the stack becomes the currently bound *<binding node>*.
- When a *set_bind* FALSE eventIn is received by a *<binding node>* not on the top of the stack, it is removed from the stack, (*isBound* FALSE), but the currently bound *<binding node>* does not change.
- If a *set_bind* FALSE is received by a node not in the stack, the event is ignored and *isBound* events are not sent.
- If all nodes are popped from the stack, then the default nodes values are used.

4.9.2 Geometry

Geometry nodes must be contained by Shape nodes - they are not leaf nodes and thus cannot be children of group nodes. 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

A geometry node can appear only in the *geometry* field of a Shape node. Several geometry nodes also contain Coordinate, Color, Normal, and TextureCoordinate as geometry property nodes. These geometry 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 determined by the parent(s) nodes of the geometry.

ISSUE: This section needs a major clarification of the interaction between Material, Texture, and the various property binding flags (per vertex, per face, overall). This will include the lighting model equations. The subsection that follows is inadequate and must be improved.

Application of material, texture, and colors:

The final rendered look of a piece of geometry depends on the Material and Texture in the associated Appearance node along with any Color node specified with the geometry (such as per-vertex colors for an IndexedFaceSet node). The following describes ideal behavior; implementations may be forced to approximate the ideal behavior:

- Either a full-color (3 or 4 component) texture OR per-vertex/per-face colors should be specified; if both a full-color texture AND colors are specified, the colors will be ignored.
- Intensity-map (1 or 2 component) should ideally modulate the intensity of the object's per-face/per-vertex colors or the diffuse color of the object's material.
- If the *material* field of the Appearance node is not NULL, then any colors or texture specified should ideally take the place of the Material node's *diffuseColor* field.
- If the *material* field of the Appearance node is NULL, then any colors or texture specified will make the geometry function as an emissive surface, unaffected by light sources. If no colors or textures are specified, then the surface should appear completely black (the default emissive color).

Shape Hints Fields:

The ElevationGrid, Extrusion, and IndexedFaceSet nodes all 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 or unknown, this field value is FALSE. 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). 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 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 Grouping nodes

Grouping nodes are used to create hierarchical transformation objects. Grouping nodes have a *children* field that contains a list of nodes which are the descendants of the group. Children nodes are restriced to the following node types:

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

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

The following nodes are grouping nodes:

- Anchor
- Billboard
- Collision
- Group
- Transform

4.9.4 Interpolators

Issue: This section needs to remove the references to time - the *key* field is not time, but the abscissa (sp?) of a linear function.

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 (e.g. scalar, color, normal, etc.). 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 nxm 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 ]
```

Note: Given a sufficiently powerful scripting language, all of these interpolators could be implemented using Script nodes (browsers might choose to implement these as pre-defined prototypes of appropriately defined Script nodes). Keyframed animation is sufficiently common and performance intensive to justify the inclusion of these classes as built-in types.

4.9.5 Lights and Lighting

Issue: This section will have a detailed description of the lighting model in VRML. This will include all the various permuations of material, texture, color, and vertex/face bindings. This needs to include Fog too.

Objects are illuminated by the sum of all of the lights in the world. This includes the contribution of both the direct illumination from lights (PointLight, DirectionalLight, and SpotLight) and the ambient illumination from these lights. Ambient illumination results from the scattering and reflection of light originally emitted directly by the light sources. Therefore, ambient light is associated with the lights in the scene, each having an *ambientIntensity* field. The contribution of a single light to the overall ambient lighting is computed as:

```
if ( light is "on" )
    ambientLight = intensity * ambientIntensity * diffuse color
else
    ambientLight = (0,0,0)
```

This allows the light's overall brightness, both direct and ambient, to be controlled by changing the intensity. Renderers that do not support per-light ambient illumination shall approximate by setting the ambient lighting parameters when the world is loaded.

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 contained by the light's parent group node (including any descendant children of the parent group node).

Temporary: Until the lighting model is clarified above, the following info should help:

- -If the geometry node specifies a non-NULL color field (e.g. IndexedFaceSet), then these colors are used as the diffuseColor (replace the material's diffuse color) for all lighting calculations.
- - If the texture is a 1 or 2 component texture, then the material's diffuse color or the geometry's colors are modulated by the texture intensity (i.e. show through the texture).
- - If the texture is a 3 or 4 component texture, then the material's diffuse color or geometry color are REPLACED by the texture color.
- -If the Appearance's material field is NULL, or Shape's appearance field is NULL, or if the material's diffuseColor, specularColor, and ambientIntensity are (0,0,0), then lighting is turned off (i.e. all light sources are ignored and the texture colors are applied directly to the geometry).

4.9.6 Sensors

There are several different kinds of sensor nodes: ProximitySensor, TimeSensor, VisibilitySensor, and a variety of pointer device sensors. Sensors are leaf nodes in the hierarchy and therefore may be children of grouping nodes.

The ProximitySensor detects when the user navigates into a specified invisible region in the world. The TimeSensor is a stop watch 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. Pointer device sensors detect user pointing events, such as the user clicking 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.

Pointer device sensors are activated when the user points to geometry that is influenced by a specific geometry sensor. Geometry sensors have influence over all geometry that is a descendent of the geometry sensor's parent group. Typically, the geometry sensor is a sibling child to the geometry that it influences. In other cases, the geometry sensor is a sibling to groups which contain geometry (that is influenced by the geometry sensor). For a given user gesture, the *lowest*, enabled geometry sensor in the hierarchy is activated - all other geometry sensors *above* it are ignored. The hierarchy is defined by the geometry leaf node which is activated and the entire hierarchy upward. If there are multiple geometry sensors tied for lowest, then each of these is activated simultaneously and independently. This last feature allows useful combinations of geometry sensors (e.g. TouchSensor and PlaneSensor).

Drag Sensors

VRML has 3 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'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 *offset* is added to the output value (e.g. SphereSensor's *rotation_changed*). 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
}
```

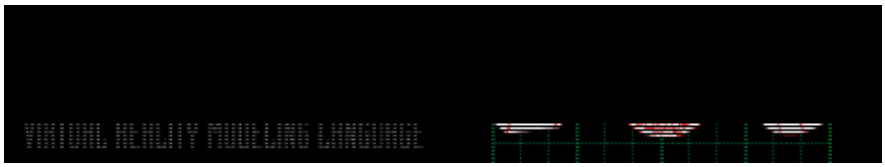
The box will spin when it is grabbed and moved via the pointer.

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.

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/concepts.html

---



The Virtual Reality Modeling Language

5. Node Reference

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 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

DirectionalLight
PointLight
Shape
Sound (AudioClip)
SpotLight
Script
WorldInfo

Sensors

CylinderSensor
PlaneSensor
ProximitySensor
SphereSensor
TimeSensor
TouchSensor
VisibilitySensor

Geometry

Box
Cone
Cylinder
ElevationGrid
Extrusion
Geometric Properties:
- Color
- Coordinate
- Normal
- TextureCoordinate

IndexedFaceSet
IndexedLineSet
PointSet
Sphere Text

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 data to be fetched over the network when any of its children are chosen. If the data pointed to is a VRML world, then that world is loaded and displayed instead of the world of which the Anchor is a part. If another 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, already-open (or newly spawned) general Web browser.

Exactly how a user "chooses" a child of the Anchor is up to the VRML browser; typically, clicking on one of its children 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 the section "Concepts -URLs and URNs" for details on the *url* field.

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 take the viewer to a particular *viewpoint* in a virtual world by specifying a URL ending with "#viewpointName", where "viewpointName" is the name of a viewpoint defined in the world. For example:

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

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

```
Anchor {
  url "#Doorway"
  children [ Sphere { } ]
}
```

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

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

See the "Concepts - Bounding Boxes" section for a description the *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 any of the fields in this node can be NULL. However, if the field is non-NULL, it must contain one specific type of node.

The *material* field, if specified, must contain a Material node. If the material fiels is NULL or unspecified, lighting is off (all lights are ignored during rendering of the object that references this Appearance) - see "Concepts - Lights and Lighting" for details of the 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 NULLor 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
}
```

The AudioClip node represents a sound that is pre-loaded by the browser, can be started at any time, and has a known duration. It can be used as the audio source for any VRML sound node.

The *url* field specifies the URL from which the sound is loaded. It must reference sound data in a supported audio format. Browsers shall support at least the *wavefile* format in uncompressed PCM format. It is recommended that browsers also support the MIDI file type 1 sound format. MIDI files are presumed to use the General MIDI patch set. Audio should be loaded when the Sound node is loaded. See the section on URLs and URNs in "Concepts/URLs and URNs" for details the *url* field. Results are not defined when the URL references unsupported data.

Issue: Need a reference/link to both wavefiles and MIDI.

Browsers may limit the maximum number of sounds that can be played simultaneously and should use the guidelines specified with the Sound node to determine which sounds are actually played.

The *description* field is a textual description of the sound, which may be displayed in addition to or in place of playing the sound.

AudioClip nodes ignore changes to their *startTime* while they are actively outputting values. If a *set_startTime* event is received while the AudioClip is active, then that *startTime* event is ignored (the *startTime* field is not changed, and a *startTime* eventOut is NOT generated). An AudioClip may be re-started while it is active by sending it a *stopTime* "now" event (which will cause the AudioClip to become inactive) and then sending it a *startTime* event (setting it to "now" or any other starting time, in the future or past).

The *loop* field specifies whether or not the sound is constantly repeated. By default, the sound is played only once. If the *loop* field is FALSE, the sound has length *duration_changed* (see below). If the loop field is TRUE, the sound repeats until time *stopTime* or forever if *stopTime* < *startTime*.

The *startTime* field specifies the time at which the sound should start playing. The *stopTime* field may be used to make a sound stop playing some time after it has started.

The *pitch* field specifies a multiplier for the rate at which sampled sound is played and is legal in the range 0 to +infinity range. Changing the *pitch* field affects the pitch of a sound. If *pitch* is set to 2.0, the sound should be played one octave higher than normal which corresponds playing it twice as fast. 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 *duration_changed* eventOut field is sent out whenever there is a new value for the "normal" duration of the clip. Typically this will only occur when the *url* field is changed, indicating that the clip is playing a different sound source. The duration is the length of time in seconds that the sound will play when the pitch is set to 1.0. Changing the *pitch* field should not trigger the *duration_changed* event.

Issue: This section needs a serious review and possible re-write/clarification.

The *isActive* field can be used by other nodes to determine if the clip is currently being played (or at least in contention to be played) by a sound node. Whenever *startTime*, *stopTime*, or `now` changes, the above rules need to be applied to determine if the sound is playing. If it is, then it should be playing the bit of sound at `(now - startTime)` or, if it is looping, `fmod( now - startTime, duration ).`

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 "*Concepts - Bindable Leaf Nodes*" and thus there exists a Background stack, in which the top-most Background on the stack is currently active. To push a Background onto 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*, pops the Background from the stack and unbinds it from the browser viewer. See "*Concepts - Bindable Leaf 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. Both spheres have infinite radius (epsilon apart), and each is painted with concentric circles of color perpendicular to the local Y axis of the sphere. The Background node is subject to the accumulated rotation transformations of its parent transformations. Scaling and translation transformations are ignored. The sky sphere is always slightly behind 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 *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 ground*Color* value 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.

Issue: Insert a diagram here

The *backUrl*, *bottomUrl*, *frontUrl*, *leftUrl*, *rightUrl*, and *topUrl* fields must specify a set of images that

define a background panorama, between the backdrop and the world's geometry. The panorama consists of a six images, each of which is mapped onto the faces of a cube surrounding the world. 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. (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. By default, there are no panorama images. Browsers are required to support the JPEG and PNG image file formats, and in addition, may support any other image formats. Support for the GIF format is also recommended. See the section "Concepts -URLs and URNs" for details on the *url* fields. If a url field refererences an unknown data, results are undefined.

Issue: Need a reference to PNG, JPEG, and GIF.

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 entered.

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 camera. 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

camera even when the camera elevates, pitches and rolls. This special case is distinguished by setting the *axisOfRotation* to (0, 0, 0).

To rotate the Billboard to face the camera, you determine the line between the Billboard's origin and the camera's origin; call this the billboard-to-camera line. The *axisOfRotation* and the billboard-to-camera 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-camera 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) (the y-axis) and the camera flies over the object, then the object will spin as the camera passes directly over the y-axis. It is undefined at the pole. Another example of this ill-defined behavior occurs when the author sets the *axisOfRotation* to (0,0,1) (the z-axis). and sets the camera to look directly down the z-axis of the object.

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

See the "Concepts - Bounding Boxes" section for a description the *bboxCenter* and *bboxSize* fields.

Box

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

This node represents a rectangular box aligned with the coordinate axes. By default, the box is centered at (0,0,0) and measures 2 units in each dimension, from -1 to +1. A box's *width* is its extent along its object-space X axis, its *height* is its extent along the object-space Y axis, and its *depth* is its extent along its object-space Z axis.

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

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**. The Collision grouping node behaves exactly like a Group node with the added functionality of specifying alternative objects to use for collision detection (rather than the rendered geometry), turning off collision detection for an entire group (including all descendants), and sending events signalling that a collision has occurred between the user's avatar and some geometry of the Collision group. If there are no Collision nodes specified in a scene, browsers are required to check for collision with all objects during navigation.

The Collision node's *collide* field turns collision detection on and off. 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 collisions with the nearest collision of any descendant geometry (or proxies) with the exception of IndexedLineSet and PointSet (these geometries have zero area). 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, and so on.

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

See the "*Concepts - Bounding Boxes*" section for a description the *bboxCenter* and *bboxSize* fields.

The collision proxy, defined in the *proxy* field, is a group or leaf node that is used as a subsitute for the Collision's children during collision detection. The proxy is used strictly for collision detection - it is not drawn. All non-geometric nodes found in the proxy are ignored.

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 actual children of the collision node are collided against.

If *proxy* is specified, then any descendant Collision 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 what happens when the user's avatar navigates into a collidable object. 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 - Lights and Lighting*" for details on lighting equations.

[Note: Some browsers may not support this functionality, in which case an average, overall color should be computed and used instead of specifying colors per vertex.]

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 the *Interpolators* section in *Key Concepts* on 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
}
```

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

The cone has two parts: the *side* and the *bottom*. Each part has an associated SFBool field that specifies whether it is visible (TRUE) or invisible (FALSE).

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

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 the *Interpolators* section in *Key Concepts* on 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
}
```

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

The cylinder has three *parts*: the *side*, the *top* (Y = +1) and the *bottom* (Y = -1). Each part has an associated SFBool field that indicates whether the part is visible (TRUE) or invisible (FALSE).

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

CylinderSensor

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

The CylinderSensor maps pointer 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. CylinderSensor uses the descendant geometry of its parent node to determine if a hit occurs.

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 and moved while over any descendant geometry nodes of the its parent group. 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 camera 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 camera 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 value and an *offset_changed* event is generated. See Drag Sensors for more details on *autoOffset* and *offset_changed*.

enter nice diagram here

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 value and an *offset_changed* event is generated. See 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. *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.

If there nested pointer device sensors (CylinderSensor, PlaneSensor, SphereSensor, TouchSensor), the lowest pointer device sensor in the graph is activated and sends outputs - all parent pointer device sensors are ignored. If there multiple, non-nested pointer device sensors, (i.e. siblings) each sensor acts independently, possibly resulting in multiple sensors activated and outputting simultaneously. If a pointer device sensor is instanced (DEF/USE), then the geometry of each parent must be tested for intersection and the sensor is activated if any of its parent's geometry is hit.

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 and Lighting*" for a detailed description of VRML's lighting equations.

A directional light source illuminates only the objects in its enclosing Group. The light illuminates everything within this coordinate system, including the objects that precede it in the scene graph.

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
}
```

This node creates a uniform rectangular grid of varying height in the local XZ plane. 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. 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.

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 Geometry concepts section 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 is used to define 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.

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).

Issue: Need to clarify how normals are generated for sides.

Texture coordinates are automatically generated by extrusions. Textures are mapped like the label on a soup can: 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 cut out of the texture square and applied to the *endCap* and/or *beginCap* planar surfaces. The *beginCap* and *endCap* textures' U and V directions correspond to the X and Z directions in which the *crossSection* coordinates are defined.

See the introductory Geometry section 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 1000
  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* units 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.

Fog nodes are "*Concepts - Bindable Leaf 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 Leaf 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 - Lights and Lighting*" for details on lighting calculations.

FontStyle

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

The FontStyle node, which may only appear in the *fontStyle* field of a Text node, defines the size, font family, and style of the text font, as well as the direction of the text strings and any specific language rendering techniques that must be used for non-English text.

The *size* field specifies the height (in object space units) of glyphs rendered and determines the spacing of adjacent lines of text depended on the direction field. All subsequent strings advance in either x or y by -(size * spacing). (See the Text node for a description of the spacing field.)

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 contains an SFString value that can 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.

The *style* field contains an SFString value that can be an empty string (the default); "BOLD" for boldface type; "ITALIC" for italic type; or "BOLD ITALIC" for bold and italic type.

Direction, Justification and Spacing

The *horizontal*, *leftToRight*, and *topToBottom* fields indicate the direction of the text. The *horizontal* field indicates whether the text is horizontal (specified as TRUE, the default) or vertical (FALSE). The *leftToRight* field indicates whether the text progresses from left to right (specified as TRUE, the default) or from right to left (FALSE). The *topToBottom* field indicates whether the text progresses from top to bottom (specified as TRUE, the default), or from bottom to top (FALSE).

The *justify* field determines where the text is positioned in relation to the origin (0,0,0) of the object coordinate system. The values for the *justify* field are "BEGIN", "MIDDLE", and "END". For a

left-to-right direction , "BEGIN" would specify left-justified text, "END" would specify right-justified text, and "MIDDLE" would specify centered text. See the FontStyle node for details of text placement.

The *spacing* field determines the spacing between multiple text strings.

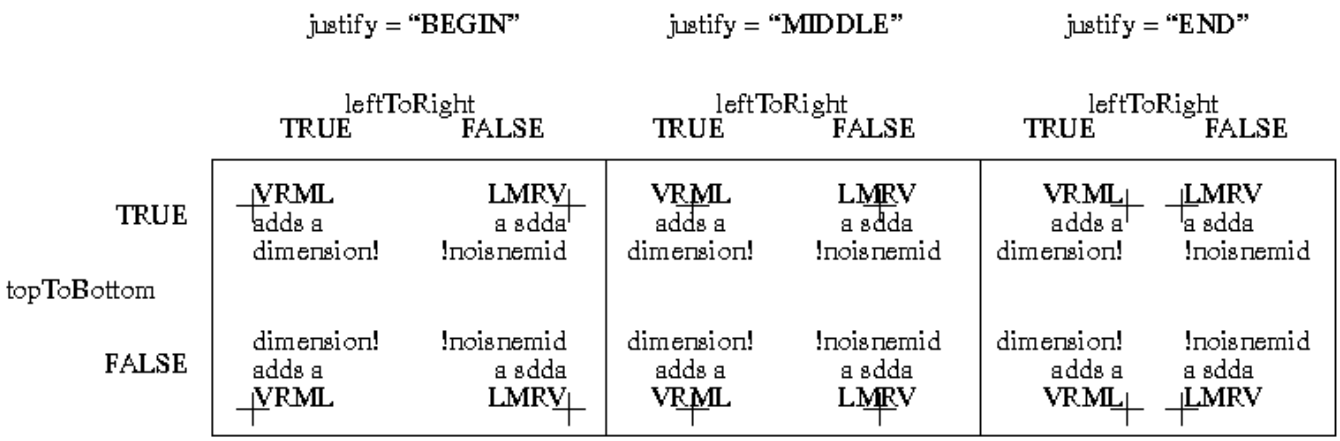
The *size* field of the FontStyle node specifies the height (in object space units) of glyphs rendered and determines the vertical spacing of adjacent lines of text. All subsequent strings advance in either X or Y by -(size * spacing). A value of 0 for spacing causes the string to be in the same position. A value of -1 causes subsequent strings to advance in the opposite direction.

For horizontal text (horizontal = TRUE), the first line of text is positioned with its baseline (bottom of capital letters) at Y = 0. The text is positioned on the positive side of the X origin when leftToRight is TRUE and justify is "BEGIN"; the same positioning is used when leftToRight is FALSE and justify is "END". The text is on the negative side of the X origin when leftToRight is TRUE and justify is "END" (and when leftToRight is FALSE and justify is "BEGIN"). For justify = "MIDDLE" and horizontal = TRUE, each string will be centered at X = 0.

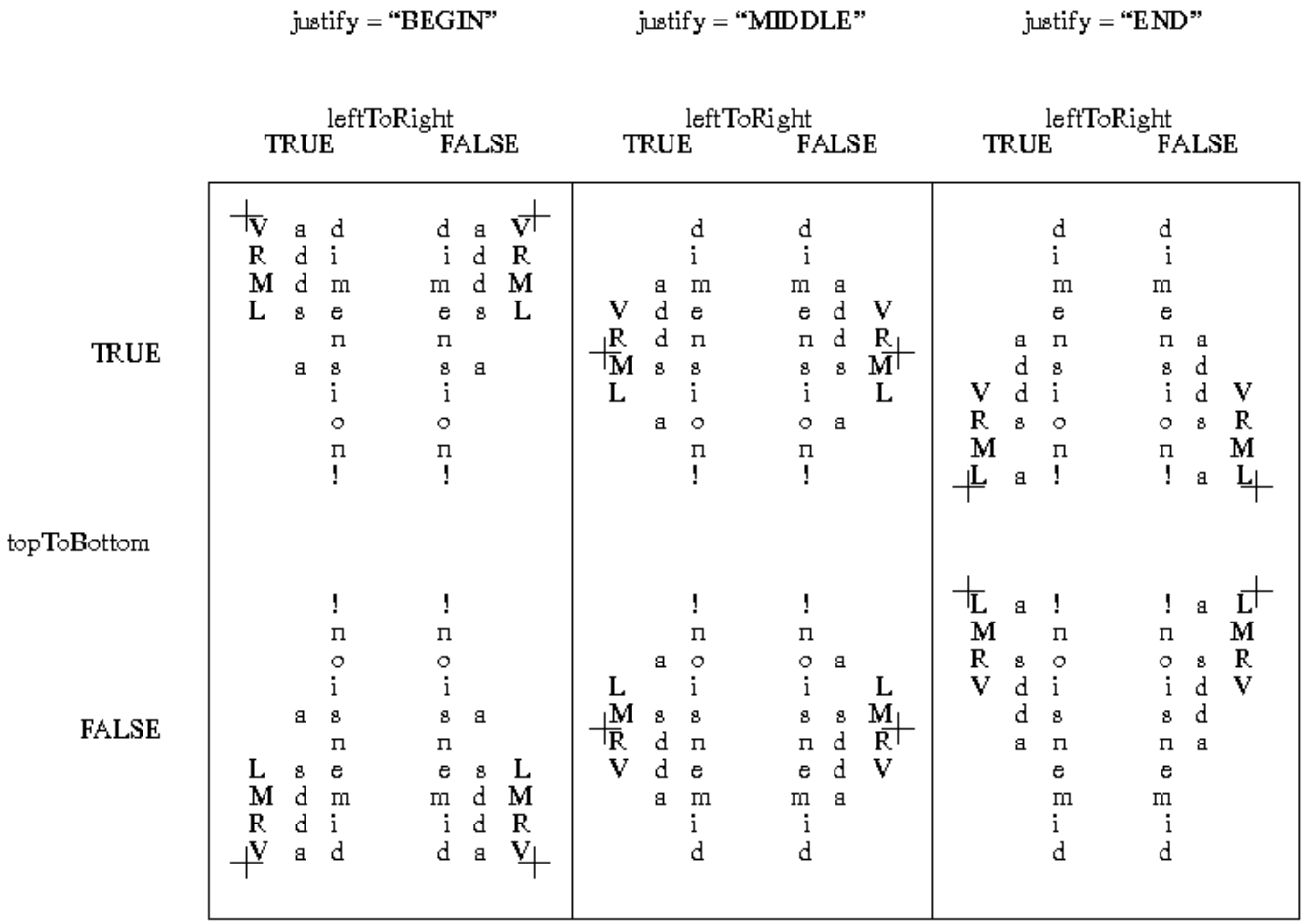
For vertical text (horizontal is FALSE), the first line of text is positioned with the left side of the glyphs along the Y axis. When topToBottom is TRUE and justify is "BEGIN" (or when topToBottom is FALSE and justify is "END"), the text is positioned with the top left corner at the origin. When topToBottom is TRUE and justify is "END" (or when topToBottom is FALSE and justify is "BEGIN"), the bottom left is at the origin. For justify = "MIDDLE" and horizontal = FALSE, the text is centered vertically at X = 0.

In the following tables, each small cross 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 uses the local language binding.

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 a lightweight grouping node that can contain any number of children. It is equivalent to a Transform node, without the transformation fields.

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

See the "*Concepts - Bounding Boxes*" section for a description 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 and parameters for that map.

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 and PNG image file formats, and in addition, may support any other image formats. Support for the GIF format is also recommended. See the section "*Concepts - URLs and URNs*" for details on the *url* field.

Issue: Need a reference to PNG, JPEG, and GIF.

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 - Lights and Lighting*" 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).

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

See "*Concepts - Lights and Lighting*" 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 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         []
}
```

This node represents a 3D shape formed by constructing polylines from vertices listed in the *coord* field. IndexedLineSet uses the indices in its *coordIndex* field to specify the polylines. 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.

For descriptions of the *coord* field, see the Coordinate node.

Lines are not texture-mapped, affected by light sources, or collided with during collision detection.

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

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.

Inline

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

The Inline node is a light-weight grouping node similar to Group that reads its children 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 displayed. An Inline with an empty URL does nothing. The *url* is an arbitrary set of URLs.

An Inline's URLs must refer to a valid VRML file that contains a grouping or child node. See the section "*Concepts - Grouping nodes*" for details on valid nodes. Referring to non-VRML files or VRML files that do not contain a grouping or leaf node is undefined.

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

See the "*Concepts - Bounding Boxes*" section for a description the *bboxCenter* and *bboxSize* fields. If the Inline's *bboxSize* field specifies a non-empty bounding box (a bounding box is non-empty if at least one of its dimensions is greater than zero), then the Inline's object-space bounding box is specified by its *bboxSize* and *bboxCenter* fields. This allows an implementation to quickly determine whether or not the contents of the Inline might be visible. This is an optimization hint only; if the true bounding box of the contents of the Inline is different from the specified bounding box, results will be undefined.

LOD

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

The LOD node, (level of detail), is used to allow browsers to switch between various representations of objects automatically. The *level* field contains nodes that represent the same object or objects at varying levels of detail, from highest to the lowest level of detail.

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 should 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 too many levels 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. Not specifying any values in the *range* field (the default) is a special case that indicates that the browser may decide which child to draw to optimize rendering performance.

Authors should set LOD ranges so that the transitions from one level of detail to the next are barely noticeable. 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. Authors should not use LOD nodes to emulate simple behaviors, because the results will be undefined. For example, using an LOD node to make a door appear to open when the user approaches probably will not work in all browsers. Use a ProximitySensor instead.

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.

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 defines surface material properties for associated geometry nodes.

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 omnidirectional 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.

See "*Concepts - Lights and Lighting*" for a detailed description of the VRML lighting model equations.

Issue: Most of the text in this section will be moved to "*Concepts - Lights and Lighting*"where a complete set of equations and cases for how Material, Texture, Fog, and Geometries interact.

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

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 0 0 then 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-vertext 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 an animated texture map (contained in a movie file) and parameters for controlling the movie and the texture mapping. The *url* field that defines the movie data must support MPEG1-Systems (audio and video) or MPEG1-Video (video-only) movie file formats. See "*Concepts - URLs and URNs*" for details on the *url* field.

See "*Concepts - Lights and Lighting*" 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. When the movie is first loaded, frame 0 is shown in the texture, if *speed* is positive, or the last frame of the movie if *speed* is negative.

MovieTextures do not play until their *startTime* is reached. At the first simulation tick when time **now** is greater than or equal to *startTime*, the MovieTexture will begin to play the movie specified by the *url* field. At this time the movie is considered started, an *isActive* TRUE eventOut is sent, and the frame displayed when the movie starts playing corresponds to the frame in the movie at time:

```
(now - startTime) / speed
```

If this value is negative, then:

```
duration + (now - startTime) / speed
```

Let this value be **frameTime[0]**, and the value of **now** be **t[0]**. Then, the frame displayed while the movie is playing at time **t[i]** (where **i** is a simulation tick) corresponds to the frame in the movie at time:

```
frameTime[i] = (frameTime[i-1] + speed * (t[i] - t[i-1]) )
```

The exposedField *stopTime* controls when the movie stops. *stopTime* is ignored if it is less than

startTime. At the first simulation tick when **time** is greater than or equal to *stopTime*, the frame displayed will correspond to the frame in the movie at *stopTime* - *startTime*, the movie will stop playing, and an *isActive* FALSE event out is sent. The *stopTime* is ignored if *loop* is FALSE and the duration of the movie, divided by the *speed* exposedField (see below), is less than *stopTime* - *startTime*.

The *speed* exposedField indicates how fast the movie should be played. It can be a movie frame rate multiplier, so a *speed* of 2 indicates the movie plays twice as fast. Note that the *duration_changed* output is not affected by the *speed* exposedField. This is because *speed* can be changed 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.

The *loop* exposedField indicates whether or not the movie should start over from the beginning after reaching its end (or, for negative speeds, start over from the end after reaching the beginning). If *loop* is TRUE, the movie will continue playing (looping) until *stopTime* is reached. If *loop* is FALSE, the movie will stop at the end of its duration (last frame for positive speeds, first frame for negative speeds) or at *stopTime*, whichever comes first. In any case, the last frame of the movie that was rendered will remain as the texture, and an *isActive* FALSE event out is sent after the last frame is rendered.

MovieTextures are either referenced by the Appearance node's texture field (as a movie texture) or by the Sound node's source field (as an audio source only). NOTE that a legal implementation of the MovieTexture node may not play audio at a speed other than 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 "*Concepts - Bindable Leaf 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. It is unique in that the current NavigationInfo is automatically 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 reparented to it. Whenever the current NavigationInfo changes, the new NavigationInfo must be reparented 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 a *isBound* FALSE event, and pops to the next entry in the stack which must be reparented to the current Viewpoint. See "*Concepts - Bindable Leaf Nodes*" for more details on the the binding stacks.

The *type* field specifies a navigation paradigm to use. The types that all VRML viewers should support are "WALK", "EXAMINE", "FLY", and "NONE". A walk viewer is used for exploring a virtual world. The viewer should (but is not required to) have some notion of gravity in this mode. A fly viewer is similar to walk except that no notion of gravity should be enforced. There should still be some notion of "up" however. An examine viewer 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 viewer controls - the user navigates using only controls provided in the scene, such as guided tours. Also allowed are browser specific viewer types. These should include a suffix as described in the naming conventions section 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.

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*.

The *avatarSize* field specifies parameters to be used in determining the viewer's dimensions for the purpose of collision detection and terrain following if the viewer type allows these. 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 camera should be maintained. The third should be the height of the tallest object over which the camera can "step". This allows staircases to be build 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 up direction (up vector), since gravity is applied in the opposte direction of the up vector. This up vector should be along the positive Y axis in the local coordinate space of the currently bound Viewpoint (ie, the accumulation of transformations of the parent Transform nodes of the Viewpoint, not including the Viewpoint's orientation field).

The *visibilityLimit* field sets the furthest distance the viewer 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.

The *speed*, *avatarSize* and *visibilityLimit* values are all scaled by the transformation being applied to currently bound Viewpoint. If there is no currenly 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 precomputed lighting (e.g. radiosity solutions) can specify the headlight off here. The headlight should have *intensity* 1, *color* 1 1 1, 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. There are cases where P and Q can be diagonally opposing in which case an infinite number of arcs exists. The interpolation for this case can be along any one of these arcs.

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.

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.

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.

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 - Lights and Lighting*" for details on the VRML lighting equations.

See the "*Field Reference - SFImage field*" 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  enabled             TRUE
  exposedField SFVec2f maxPosition         -1 -1
  exposedField SFVec2f minPosition         0 0
  exposedField SFVec3f offset              0 0 0
  exposedField SFBool  autoOffset          TRUE
  eventOut     SFBool  isActive
  eventOut     SFVec3f trackPoint_changed
  eventOut     SFVec3f translation_changed
}
```

The PlaneSensor maps pointer 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 and moved while over any descendant geometry nodes of the its parent group. 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 camera 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 camera 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 translations in the XY plane of the sensor's local coordinate system. 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.

If there nested pointer device sensors (CylinderSensor, PlaneSensor, SphereSensor, TouchSensor), the lowest pointer device sensor in the graph is activated and sends outputs - all parent pointer device sensors are ignored. If there multiple, non-nested pointer device sensors, (i.e. siblings) each sensor acts independently, possibly resulting in multiple sensors activated and outputting simultaneously. If a pointer device sensor is instanced (DEF/USE), then the geometry of each parent must be tested for intersection and the sensor is activated if any of its parent's geometry is hit.

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 defines a point light source at a fixed 3D location. A point source illuminates equally in all directions; that is, it is omnidirectional.

See "*Concepts - Lights and Lighting*" for a detailed description of VRML's lighting equations.

A PointLight illuminates everything within *radius* of its *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. Renderers that do not support a full attenuation model may approximate as necessary. PointLights are leaf nodes and thus are transformed by the transformation hierarchy of its parents.

PointSet

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

The PointSet node represents a set of points listed in the *coord* field. The coord field must be a Coordinate node (or instance of a Coordinate node). PointSet uses the coordinates in order.

If the color field is not NULL, it must contain a Color node that contains at least the number of points contained in the *coord* node. Colors are always applied to each point in order. Points are not texture-mapped, affected by light sources, or collided with during collision detection.

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.

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

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. Ideally, implementations will 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 specify the position and orientation of the viewer in the ProximitySensor's coordinate system and are generated when the user moves while in inside the region being sensed - 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, or the bound Viewpoint or its coordindate 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 VisibilitySensors 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.

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 general 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 and set global information such as navigation information, 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.

Shape

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

A Shape node has two fields: *appearance* and *geometry*. These fields, in turn, contain other nodes. The *appearance* field contains an Appearance node that has *material*, *texture*, and *textureTransform* fields (see the Appearance node). The *geometry* field contains a geometry node. The specified appearance nodes are applied to the specified geometry node.

See "*Concepts - Lights and Lighting*" for a detailed description of the interaction between Appearance and Geometry nodes.

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
}
```

Note: See TimeSensor for a good idea for how Sound should execute - this section needs to be clarified and improved.

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. 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. If the distance is set to the maximum value, the sound will be ambient over the entire VRML scene.

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 must point to either an AudioClip or a MovieTexture node. Furthermore, the MovieTexture node must refer to a movie format that supports sound (e.g. MPEG1-Systems).

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 *starTime* 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 focii 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 inner ellipsoid defines an 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 value.

Sphere

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

The Sphere node represents a sphere. By default, the sphere is centered at the origin and has a radius of 1.

Spheres generate their own normals. When a texture is applied to a sphere, the texture covers the entire surface, wrapping counterclockwise from the back of the sphere. The texture has a seam at the back on the YZ plane.

SphereSensor

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

The SphereSensor maps pointer 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 and moved while over any descendant geometry nodes of the its parent group. 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 camera 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 camera 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. 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. If *autoOffset* is FALSE,

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. *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.

If there nested pointer device sensors (CylinderSensor, PlaneSensor, SphereSensor, TouchSensor), the lowest pointer device sensor in the graph is activated and sends outputs - all parent pointer device sensors are ignored. If there multiple, non-nested pointer device sensors, (i.e. siblings) each sensor acts independently, possibly resulting in multiple sensors activated and outputting simultaneously. If a pointer device sensor is instanced (DEF/USE), then the geometry of each parent must be tested for intersection and the sensor is activated if any of its parent's geometry is hit.

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 is placed at a fixed location in 3-space and illuminates in a cone along a particular direction.

See "*Concepts - Lights and Lighting*" for a detailed description of VRML's lighting equations.

The cone of light extends a maximum distance of *radius* from its *location*. 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. Renderers that do not support a full attenuation model may approximate as necessary.

The intensity of the illumination may drop off as the ray of light diverges from the light's direction toward the edges of the cone. The angular distribution of light is controlled by the *cutOffAngle*, beyond which the illumination is zero, and the *beamWidth,* the angle at which the beam starts to fall off. Renderers that support a two cone model with linear fall off from full intensity at the inner cone to zero at the cutoff cone should use *beamWidth* for the inner cone angle. Renderers that attenuate using a cosine raised to a power, should use an exponent of *exponent = 0.5*log(0.5)/log(cos(beamWidth))*. When *beamWidth >= PI/2* the illumination is uniform up to the cutoff angle, which is the default.

Switch

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

The Switch grouping node traverses zero or one of its descendants (which are specified in the *choice* field).

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.

Text

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

The Text node represents one or more text strings specified using the UTF-8 encoding as specified as 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 the text string if the natural length of the string is longer than the maximum extent, as measured in the local coordinate space. If the text string is shorter than the maximum extent, it is not changed. The maximum extent is measured horizontally for horizontal text (FontStyle node: *horizontal*=TRUE) and vertically for vertical text (FontStyle node: *horizontal*=FALSE).

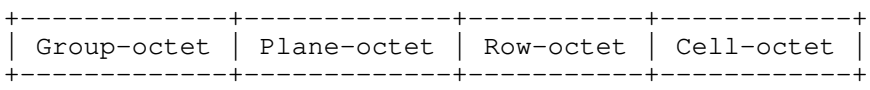
The *length* field contains an MFFloat value that specifies the length of each text string in the local coordinate space. If the string is too short, it is stretched (either by scaling the text or by adding space between the characters). If the string is too long, it is compressed (either by scaling 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 3D text as follows. The texture origin is at the origin of the first string, as determined by the justification. The texture is scaled equally in both S and T dimensions, with the font height representing 1 unit. S increases to the right, 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:



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.

TextureCoordinate

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

This node defines a set of 2D coordinates to be used by vertex-based geometry nodes (e.g. IndexedFaceSet and ElevationGrid) *texCoord* field to map textures to the vertices of some geometry nodes.S

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

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. This node is used only in the *textureTransform* field of the Appearance node and affects the way textures are

applied to the surfaces of the associated Geometry node. The transformation consists of (in order) a nonuniform scale about an arbitrary center point, a rotation about that same point, and a translation. This allows a user to change the size and position of the textures on shapes.

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 eventOuts include *isActive*, which is TRUE if the TimeSensor is running, and FALSE otherwise. The remaining outputs are *fraction_changed*, which is an SFFloat in the interval [0,1] representing time cycle, *time*, an SFTime event specifying the absolute time for a given tick, and *cycleTime*, an SFTime event that sends a time event when a cycle is about to begin (useful for synchronization with other time-based objects).

If the *enabled* exposedField is TRUE, the TimeSensor is enabled and running. When *enabled* is FALSE the TimeSensor does not generate outputs and *isActive* is set to FALSE. However, events on the exposedFields of the TimeSensor, such as *set_startTime,* are processed and *startTime_changed* events are sent regardless of the state of *enabled*.

TimeSensors remain inactive until their *startTime* is reached. At the first simulation tick when time, "now", is greater than or equal to *startTime*, the enabled TimeSensor will begin generating *time* and *fraction* events, which may be routed to other nodes to drive continuous animation or simulated behaviors --(see below for behavior at read time). *Time* events output the absolute time for a given tick of the TimeSensor (time is number of seconds since 12 midnight GMT January 1, 1970). The *cycleInterval* field defines the length of time for execution - this field's values must be greater than 0 (<= 0 produces undefined results). *Fraction_changed* events output a floating point value in the 0.0 to 1.0 range, where 0.0 corresponds to *startTime* and 1.0 corresponds to *startTime*+*cycleInterval:*

```
time = now
fraction = fmod(now - startTime, cycleInterval)
```

Whenever the fraction equals 0.0, *cycleTime* outputs the current time -- this denotes the beginning of an interval and is used for synchronization purposes.

The length of time a TimeSensor generates events is controlled using *cycleInterval, loop,* and *stopTime.* If *loop* is TRUE, the TimeSensor runs until either *stopTime* is reached, or if *stopTime* < *startTime*, then forever. If *loop* is FALSE (default), the TimeSensor generates time events until time

startTime+*cycleInterval* or *stopTime*, depending on which comes first (assuming that stopTime >= startTime). 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 other values of the other fields.

TimeSensors ignore changes to their *cycleInterval*, *enabled*, *loop*, and *startTime* fields while they are actively outputting values. For example, if a *set_startTime* event is received while the TimeSensor is active, then that *set_startTime* event is ignored (the *startTime* field is not changed, and a *startTime_changed* eventOut is not generated). A TimeSensor may be re-started while it is active by sending it a *set_stopTime* "now" event (which will cause the TimeSensor to become inactive) and then sending it a *set_startTime* event (setting it to "now" or any other starting time, in the future or past). If an active TimeSensor receives a *stopTime* event that is less than "now", it behaves as if the *stopTime* requested is "now" and sends the final events (note that *stopTime* is set as specified).

A TimeSensor will generate an *isActive* TRUE event when it begins generating times, and will generate an *isActive* FALSE event when it stops generating times (either because *stopTime* or *startTime*+*cycleInterval* was reached). *isActive* events are only generated when the state of *isActive* changes.

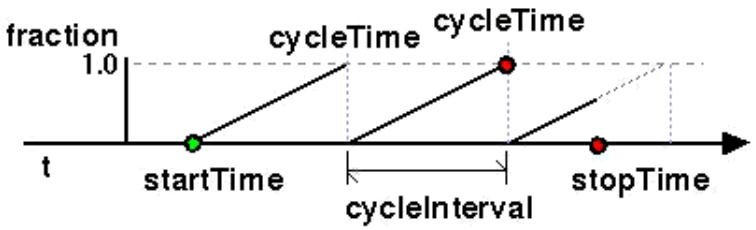
Setting the *loop* field to TRUE makes the TimeSensor start generating events at *startTime* and continue generating events until *stopTime* (if *stopTime* >= *startTime*) or forever (if *stopTime* < *startTime*). This use of the TimeSensor should be used with caution, since it incurs continuous overhead on the simulation. Setting *loop* to FALSE and *cycleInterval* to 0 will result in a single *time* event being generated at *startTime*; this can be used to build an alarm that goes off at some point in the future.

If *startTime* equals *stopTime* then a single *time* event is generated at *startTime*/*stopTime*, a *fraction_changed* event of 0.0 is sent, a *cycleTime_changed* of *startTime*/*stopTime* is sent, and *isActive* remains unchanged at FALSE.

No guarantees are made with respect to how often a TimeSensor will generate time events, but TimeSensors are guaranteed to generate final *time_changed* and *fraction_changed* events. If loop is FALSE, the final time event will be generated at (*startTime*+*cycleInterval*) or *stopTime (*if *stopTime* >= *startTime),* which ever comes first. If *loop* is TRUE, then the final event will be generated at *stopTime* (if *stopTime* >= *startTime)* or never.a TimeSensor with default *startTime, stopTime*, and *loop* values does not generate any eventOut events upon reading.

Note that if a FALSE value of *enabled* is received while the TimeSensor is running, the sensor should evaluate and send all relevant outputs, send a FALSE value for *isActive,* and then disable itself. If a stopTime is received that is less than the current time, now, it is ignored.

The following diagram illustrates the TimeSensor node’s time function:



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 activated or deactivated 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 camera 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 only generates events 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 camera 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.

isOver TRUE/FALSE events are generated as the pointing device "passes over" the TouchSensor's geometry. When the pointing device moves to a position such that its bearing intersects any of the TouchSensor's geometry, an *isOver* TRUE event should be generated. When the pointing device moves to a position such that its bearing no longer intersects the geometry, or some other geometry is obstructing the TouchSensor's geometry, an *isOver* FALSE event should be generated. These events are generated only when the pointing device has moved; 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).

If there nested pointer device sensors (CylinderSensor, PlaneSensor, SphereSensor, TouchSensor), the lowest pointer device sensor in the graph is activated and sends outputs - all parent pointer device sensors are ignored. If there multiple, non-nested pointer device sensors, (i.e. siblings) each sensor acts independently, possibly resulting in multiple sensors activated and outputting simultaneously. If a pointer device sensor is instanced (DEF/USE), then the geometry of each parent must be tested for intersection and the sensor is activated if any of its parent's geometry is hit.

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 Nodes*" section for a description the *children*, *addChildren*, and *removeChildren* fields and eventIns.

See the "*Concepts - Bounding Boxes*" section for a description 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. 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.

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
               ...
}}}}}}}
```

Given a 3-dimensional point **P** and the above transformation node, **P** is transformed into point **P′** in its parent's coordinate system by first scaling, then rotating, and finally translating. In matrix-transformation notation, thinking of T, R, and S as the equivalent transformation matrices,

P′ = T·R·S·−SR·−TC · **P** (**P** is a column vector)

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 0
  field        SFString   description    ""
  eventOut     SFTime     bindTime_changed
  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 Leaf 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 pushed onto the Viewpoint stack and activated. When a Viewpoint is bound, the browser's user view (i.e. camera) is conceptually reparented as a child of the Viewpoint. All subsequent changes to the Viewpoint's coordinate system automatically 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 pops 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 reparented to the next entry in the stack. See "*Concepts - Bindable Leaf Nodes*" for more details on the the binding stacks.

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. Note that the single orientation rotation (which is a rotation about an arbitrary axis) is sufficient to completely specify any combination of view direction and "up" vector. Viewpoints are effected by the transformation hierarchy.

For viewer types (see NavigationInfo) that require a definition of an up vector, the positive Y axis of the transformation space of the currently bound Viewpoint defines the up vector. Note that the *orientation* field of the Viewpoint does not affect the definition of the up vector. This allows the author to separate the view direction from the up vector definition.

The *jump* exposed field specifies whether the browser's user view 'jumps' (or animates) to the position and orientation of the newly bound Viewpoint. If *jump* is TRUE and a *set_bind* TRUE event is received, then the current user's view is saved in the viewpoint stack and the user view is changed to match the values in the *position* and *orientation* fields. If the most recently bound Viewpoint receives a *set_bind* FALSE event with its *jump* field set to TRUE, it is popped from the stack and the previously pushed position and orientation become the current view. If a Viewpoint that is not the top of stack receives a *set_bind* FALSE event, it has effect on the browser's user view. In this way a user can press a button and be teleported to another location by wiring the button to a *set_bind* TRUE event of a Viewpoint with the desired destination and *jump* TRUE. Another button can be pressed to return the user to the original location by wiring it to a *set_bind* FALSE event on that same viewpoint. If the *jump* value is FALSE, then

The *fieldOfView* field specifies a preferred field of view from this viewpoint, in radians. A smaller field of view corresponds to a telephoto lens on a camera; a larger field of view corresponds to a wide-angle lens on a camera. The field of view should be greater than zero and smaller than PI; the default value

corresponds to a 45 degree field of view. *fieldOfView* is a hint to the browser and may be ignored. A browser rendering the scene into a rectangular window will ideally scale things such that a solid angle of *fieldOfView* from the viewpoint in the view direction will be completely visible in the window.

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_changed* eventOuts are sent.

The first Viewpoint encountered in the file is automatically bound (receives *set_bind* TRUE) and is used as the initial location of the user view when the world is entered. 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.

If a Viewpoint is bound (*set_bind*) and is the child of an LOD, Switch, or any grouping node or prototype that disables its children, then the result is undefined.

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 transfomations of its parents.

The *enterTime* event is generated whenever the *isActive* TRUE event is generated, and *exitTime* events are generated whenever *isActive* FALSE event is 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

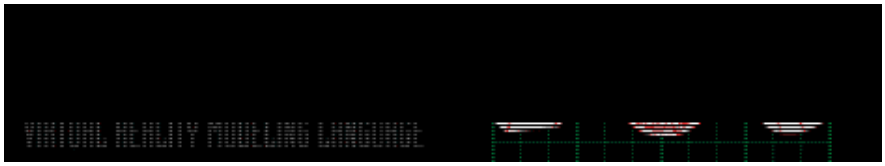
```
WorldInfo {
  field MFString info  []
  field SFString title ""
}
```

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.

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/nodesRef.html

---



The Virtual Reality Modeling Language Specification

6. Field Reference

6.1 Introduction

6.2 SFBool

6.3 SFColor and MF Color

6.4 SFFloat and MFFloat

6.5 SFImage

6.6 SFInt32 and MFInt32

6.7 SFNode and MFNode

6.8 SFRotation and SFRotation

6.9 SFString and MFString

6.10 SFTime

6.11 SFVec2f and MFVec2f

6.12 SFVec3f and MFVec3f

6.1 Introduction

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

Multiple-valued fields are written as a series of values enclosed in square brackets, and separated by whitespace (e.g. commas). If the field 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 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.

6.3 SFColor/MFColor

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

```
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.

6.4 SFFloat/MFFloat

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

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

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

6.5 SFImage

The SFImage field defines a single uncompressed 2-dimensional pixel image. SFImage fields 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 contains 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.

6.6 SFInt32/MFInt32

The SFInt32 field contains zero or one 32-bit integer, and the MFInt32 field contains zero or more 32-bit integers. S/FFInt32 fields 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.

6.7 SFNode/MFNode

The SFNode field contains zero or one nodes, and the MFNode field syntax is just the node that it contain. For example, this is valid syntax for an MFNode field, *fooNode*:

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

The S/MFNode fields may also contain the keyword NULL to indicate that it is empty.

6.8 SFRotation/MFRotation

The SFRotation field contains zero or one arbitrary rotation, and the MFRotation field contains zero or more arbitrary rotations. S/MFRotations are written to file as four floating point values separated by whitespace. The 4 values represent an axis of rotation followed by the amount of right-handed rotation about that axis, in radians. For example, an SFRotation containing a 180 degree rotation about the Y axis is:

```
fooRot 0 1 0  3.14159265
```

6.9 SFString/MFString

The SFString and MFString fields contain strings formatted with the UTF-8 universal character set (ISO/IEC 10646-1:1993, http://www.iso.ch/cate/d18741.html). SFString contains zero or one string, and the MFString contains 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.

6.10 SFTime/MFTime

Field containing a single time value. Each time value is written to file as a double-precision floating point number in ANSI C floating point format. An absolute SFTime is the number of seconds since Jan 1, 1970 GMT.

6.11 SFVec2f/MFVec2f

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

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

is a MFString field, *fooVec2f*, with two valid vectors.

6.12 SFVec3f/MFVec3f

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

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

is a MFString field, *fooVec3f*, with two valid vectors.

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 Minumum Support Requirements

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

7.1 Introduction

7.2 Conformance

7.3 Minimum support requirements

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.

Issue: This is a first draft of the conformance and is open for discussion and change...rc

Issue: Need to support at least 32?, 64?, 128? levels of hierarchy?

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 strings	At least 255 characters per string
All URL fields	At least 16 URL's per field
Anchor	Ignore parameters Ignore description
Appearance	Full support
AudioClip	Ignore description At least 30 seconds duration Wavefile in uncompressed PCM format
Background	At least the first specified pair of ground colours and angles interpreted At least the first specified pair of sky colours and sky angles interpreted
Billboard	Full support except as for all groups
Box	Full support
Collision	Full support except as for all groups
Colour	Full support
ColourInterpolator	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

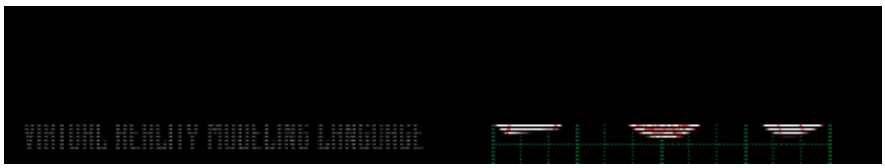
Fog	Full support
FontStyle	If non-Latin characters, family can be ignored.
Group	Full support except as for all groups
ImageTexture	Point sampling At least JPEG and PNG formats
IndexedFaceSet	At least 1024 vertices per face At least 1024 faces Ignore ccw Ignore convex Ignore solid
IndexedLineSet	At least 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	At least radians of beam width At least radians of cut-off Linear fall off from beam width
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

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/conformance.html



The Virtual Reality Modeling Language Specification

A. Grammar Definition

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 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

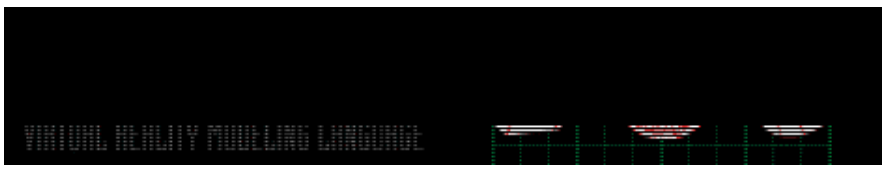
mfint32Value:
sfint32Value
[ ]
[*sfint32Values* **]**

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

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/grammar.html



The Virtual Reality Modeling Language Specification

B. External Programming Interface Reference

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

***** IMPORTANT NOTE *****

This annex of the specification is under discussion. It is **not** officially accepted as part of the specification. If consensus can be reached, it may be official by the August 4th deadline. Otherwise, it will be in the next revision of the spec ...

Also. note that this section needs to be written in a language-neutral manner. The specific language bindings for Java and JavaScript belong in separate appendices.

B.1 Introduction

For communication between a VRML world and "the outside" an external API for VRML is needed. By "the outside" I mean whatever environment the VRML browser is running in. In this proposal the interface internal to the VRML world (the *internal* interface)is separated from the interface on the outside (the *external* interface). The external interface must be bound to the environment, whether this is Java or some other language, or some sort of network aware socket interface. I will describe the internal interface here and give semantics (but not syntax) for the external interface. A separate document will explore some potential external bindings.

B.2 VRML 2.0 API

Nodes in VRML can be named using the "DEF" construct. This naming gives a convenient mechanism for accessing nodes from an external interface. Any node named with the DEF construct can be accessed from outside the VRML world. A node so named is referred to as an *accessible* node. Access to the node's internal data structures from the external API is limited to the eventIns and eventOuts of that

node. The external API can send an eventIn to a accessible node and that node will receive that event just as if that event had come through a ROUTE. When an eventOut is sent from an accessible node that event is sent through the external API for handling.

B.2.1 Language Access

External access to the VRML world will typically be done with an API on some scripting language. An example of this would be Netscape's JavaScript interface known as LiveConnect. This interface allows a plug-in (such as a VRML browser) to expose a set of methods callable from JavaScript or Java on the HTML page. The syntax for such an implementation is outside the scope of this document. The details on the implementation of bindings for a few languages see the VRML External API Bindings document.

For consistency this document contains a C-like language independent syntax for the external bindings. This syntax is to be used as a guide for creating a binding to a particular language. Some languages may be able to use this syntax as is, while others will have to change it to meet language requirements. But all implementations should match the methods and parameterization given, regardless of final syntax.

B.2.2 Member Functions

Conceptually, the external API provides an interface to the eventIns and eventOuts of accessible nodes, and an interface to the Browser API. One of the Browser API calls allows a pointer to a node to be obtained, given a DEF name string. Once this node pointer is gotten the eventIns and eventOuts of that node can be accessed. Three operations are required on the events of a node. The external language must be able to send a typed eventIn to the given node. It must also be able to be activated (have a method called) when an eventOut is generated from a given node. Finally, it must be able to read the current value of an eventOut of a given node.

Once obtained, a node pointer contains member functions which are equivalent to the eventIn and eventOut names of that node. For instance, given this node:

```
SomeNode {
    eventIn  SFTime startTime
    eventOut SFBool isActive
}
```

When a pointer to SomeNode is obtained it has 3 member functions:

```
void   startTime(SFTime value);
void   isActive(void *callback, SFInt32 id);
SFBool get_isActive();
```

The first function sends an eventIn to *startTime*. The second sets an implementation specific callback, called when the *isActive* eventOut is generated by the node. This could be a string of JavaScript statements to be executed, a Java method, or a data structure containing network access information. The *id* is sent to the callback method for unique identification of the event. The last function gets the current value of the *isActive* eventOut. For every eventOut *foo* there is a *get_foo* which gets the current value of the eventOut.

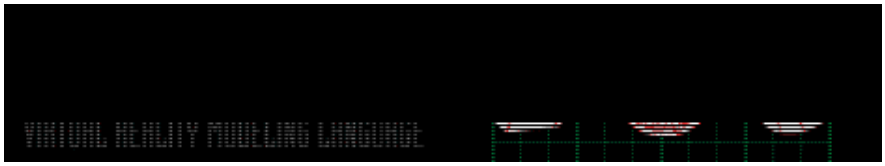
B.2.3 Browser API

The VRML world has an externally accessible API which provides functionality in addition to the event access described above. This gives access to the scene, some browser functions and, most importantly, a way to get node pointers. The functions listed here are all methods on the browser object, which is the external language's interface to the VRML world. This may be an embedded frame in JavaScript, a class instance in Java, or a network accessible process containing a VRML world.

SFNode getNode(SFString name)	Get a pointer to the node with the passed DEF *name*.
SFString getName()	Get a string with the name of the VRML browser.
SFString getVersion()	Get a string containing the version of the VRML browser.
SFFloat getCurrentSpeed()	Get the floating point current rate at which the user is traveling in the scene.
SFFloat getCurrentFrameRate()	Get the floating point current instantaneous frame rate of the scene rendering, in frames per second.
SFString getWorldURL()	Get a string containing the URL of the currently loaded world.
void replaceWorld(MFNode nodes)	Replace the current world with the passed list of *nodes*.
void createVRMLFromURL(MFString url, SFNode node, SFString event)	Parse the data in *url* into a VRML scene. When complete send *event* to *node*. The event is a string with the name of an MFNode eventIn in the passed node. For instance, if *node* were a Transform, *event* would be "add_children". When the url is loaded and scene parsed, the resulting nodes would be added as children of the Transform.
MFNode createVRMLFromString(SFString str)	Parse *string* into a VRML scene and return the list of root nodes for the resulting scene.
void addRoute(SFNode fromNode, SFString fromEventOut, SFNode toNode, SFString toEventIn)	Add the route *fromNode.fromEventOut* TO *toNode.toEventIn*.
void deleteRoute(SFNode fromNode, SFString fromEventOut, SFNode toNode, SFString toEventIn)	Remove the route *fromNode.fromEventOut* TO *toNode.toEventIn*, if one exists.

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/external_API.html.



The Virtual Reality Modeling Language Specification

Java and JavaScript External API Bindings

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

***** IMPORTANT NOTE *****

This annex of the specification is under discussion. It is **not** officially accepted as part of the specification. If consensus can be reached, it may be official by the August 4th deadline. Otherwise, it will be in the next revision of the spec ...

This section needs to be divided into two separate appendices: Java External API and JavaScript External API.

This section describes bindings for the external application programming interface to the VRML browser.

VRML 2.0 has a standard mechanism for interfacing to the outside world. Depending on the external environment a large variety of bindings to the VRML world are possible. This document describes bindings for some popular environments. For a description of the external API general operation see the appendix "External Programming Interface".

Language Bindings

While the actual syntax of the API is language dependent, the mapping of nodes, fields within those nodes and methods on the browser itself are described in the appendix "External Programming Interface Reference".

Java

Java provides its own environment. The VRML world appears as an instance of the *vrml* java class. This instance can be created by the Java applet or, in the case of Java-within-Netscape, it can be accessed through the **document** interface. In either case a pointer to the world allows access to the "External Programming Interface Reference - Browser API" and to the fields of specified nodes.

The Browser API has parameter and return types shown as VRML field types. In Java these are classes as described in the appendix "Java Scripting Reference".

ISSUE: Can the actual node returned from getNode() be a subclass of the base SFNode class? If so we should be able to have methods on the returned node pointer correspond to the eventOuts and eventIns of the node, as described in the External API document.

JavaScript

The JavaScript language allows an author to write small scripts directly on an HTML page. JavaScript can interface to a plug in using a technology known as LiveConnect. The plug-in presents its interface as a Java object, and since JavaScript can interface with Java objects it can interface directly with the plug-in. Therefore the interface is identical to that for Java - the Browser API is used to get a pointer to a node (which is an instance of a Java SFNode class), then the fields of that node can be accessed as described in the External API document.

Here is an example of interfacing to a VRML world from JavaScript. A VRML world is embedded on an HTML page with the EMBED tag. This tag allows a name to be assigned to the VRML world. This name is contained in the **document** object and allows access to the Browser API of the VRML world. The Browse API can be used to get a pointer to a named node which can be used to send and receive events. For instance, take this VRML world:

```
#VRML V2.0 ...

DEF Camera Viewpoint { }
DEF MySphere Transform {
    children [
        Shape {
            appearance Appearance {
                material Material { diffuseColor 0 0 1 }
            }
            geometry Sphere { }
        }
    ]
 }
```

If this world is embedded in an HTML page with the name "vrml" a JavaScript author can translate the Sphere by a relative amount by using the following JavaScript:

```
function myTranslator(x, y, z) {
    node = document.vrml.getNode("MySphere");
    translate = node.get_translation();
    translate[0] += x;
    translate[1] += y;
    translate[2] += z;
```

```
    node.set_translation(translate);
 }
```

To change the authored position of the camera in the above scene use the following JavaScript:

```
function myCameraMover(x, y, z) {
    node = document.vrml.getNode("Camera");
    translate = node.get_position();
    translate[0] += x;
    translate[1] += y;
    translate[2] += z;
    node.set_position(translate);
}
```

Handling an eventOut from JavaScript requires a bit more work. First, a *callback* and *id* must be registered with the eventOut. The callback is a language specific value. In JavaScript this is a function pointer which will get called when the eventOut occurs. It will be called with 3 parameters:

- *value* is the value being sent
- *timestamp* is the time at which the event occurred
- *id* is a user defined id for this event (set when the callback was setup)

> ***ISSUE:*** *In discussions with Paper we discussed having the JavaScript callback be a string of JavaScript to be executed when the event comes in. But there is no way to pass the value or timestamp of the event. So I have shown the callback as a JavaScript function pointer to which 3 parameters are passed. This may not be possible in JavaScript in which case another solution will have to be found.*

For instance, given this world:

```
#VRML V2.0 ...

Shape { ... geometry for a cup ... }
DEF Cup TouchSensor { }
```

and this JavaScript:

```
function handleEventOut(value, timestamp, id) {
    ... handle value event ...
}

function mySetup() {
    node = document.vrml.getNode("Cup");
    node = isActive(handleEventOut, 0);
}
```

When the **mySetup()** function is called the handler is set up. Now when the user presses the mouse button over the cup in the VRML world the function **handleEventOut(true, <time>, 0)** is called. When the user releases the mouse **handleEventOut(false, <time>, 0**) is called.

Integrating Javascript Control with VRML Functionality

Interacting with the scene using the above interface allows simple manipulation of specific nodes. For

more complex interaction JavaScript can send events to a Script node in the VRML scene which performs complex scene control. Here is a simple JavaScript function which starts an interpolated animation in the VRML scene:

```
function myStartAnimation() {
    interface = document.vrml.getNode("Interface");
    interface.start(true);
}
```

The VRML scene looks like this:

```
...
DEF Interface Script {
    eventIn SFBool start
    eventOut SFTime startTime

    url "vrmlscript:
         function start(value, timestamp) {
             startTime = timestamp;
         }"
}

DEF PI PositionInterpolator {
    keys [ ... keys ... ]
    values [ ... position values ... ]
}

DEF TS TimeSensor { cycleInterval 5 } # 5 second animation

DEF T Transform {
    children [ ... geometry to animate ... ]
}

ROUTE Interface.startTime TO TS.startTime
ROUTE TS.fraction TO PI.set_fraction
ROUTE PI.outValue TO T.translation
...
```

The Script can then send multiple messages to start combinations of operations.

With this interface a VRML scene can be completely controlled from external Javascript functions or the external controls can simply stimulate the VRML scene to do more complex functions internally.

Browser API

The Browser API can do many other operations on the VRML world. For example, constructing the first scene above could be done entirely in JavaScript using the Browser API:

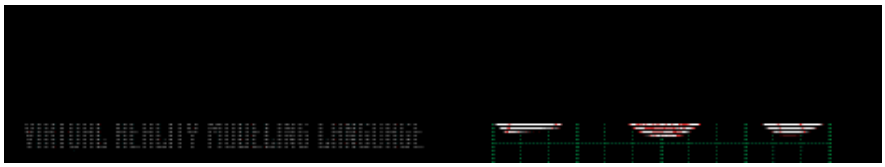
```
function mySceneBuilder {
    with (document.vrml) {
        // create the 2 root nodes
        scene[0] = createVRMLFromString("DEF Camera Viewpoint { }");
        scene[1] = createVRMLFromString("DEF MySphere Transform { }");

        // create the shape and its children
        shape = createVRMLFromString("Shape { }");
        appearance = createVRMLFromString("Appearance { }");
```

```
            shape.set_appearance(appearance);
            geometry = createVRMLFromString("Sphere { }");
            shape.set_geometry(geometry);

            // add a material to the appearance
            material = createVRMLFromURL("Material { }");
            color[0] = 0; color[1] = 0; color[2] = 1;
            material.set_diffuseColor(color);
            appearance.set_material(material);

            // add the shape to the Transform
            scene[1].add_children(shape);

            // make this the vrml scene
            replaceWorld(scene);
        }
    }
```

Now the scene can be controlled as in the early examples.

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/externalBindings.html.



The Virtual Reality Modeling Language Specification

C. Examples

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

*** IMPORTANT NOTE ***

This document is not completely up to date with Draft #3, but is still very useful. It will be updated

---

This section 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 [

    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 { Sphere { radius .2 } }
    }
    Transform {
      translation -2 0 0
      children    USE Joe
    }

  ]
}
```

Prototype example

A simple chair with variable colors for the leg and seat might be prototyped as:

```
PROTO TwoColorChair [ field MFColor legColor  .8 .4 .7
                      field MFColor seatColor .6 .6 .1 ]
{
  Transform {
    children [
      Transform {   # chair seat
        children
          Shape {
            appearance Appearance {
              material Material { diffuseColor IS seatColor }
            }
            geometry Box { ... }
          }
      }

      Transform {   # chair leg
        translation ...
        children
          Shape {
            appearance Appearance {
              material Material { diffuseColor IS legColor }
            }
            geometry Cylinder { ... }
          }
      }
    ] # 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 chair has red legs and a green seat:

```
TwoColorChair {
  legColor 1 0 0
  seatColor 0 1 0
}
```

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 ascii
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 { }
                    }
                ]
            }
        ]
    }
]
}
```

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 SFColor ambientColor      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 {
            ambientColor  IS ambientColor
            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 SFColor ambientColor      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 ]

    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 take 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 looking from the viewpoint named "OverView" when the Box is chosen. If no world is specified, then the current scene is implied; for example:

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

takes 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 [
    Shape { ... }
    DirectionalLight { .... } # lights the preceding 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 { rgb [ 1 0 0, 0 1 0, 0 0 1 ] }
  }
}
Shape {
  geometry PointSet {
    coord USE mypts
    color Color { rgb [ .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. 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 {
  MFFloat keys      [   0.0,    0.5,    1.0 ]
  MFColor values    [ 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 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 made to generate a single event upon reading by routing the *enterTime* of a ProximitySensor that encloses the world to the TimeSensor's *startTime.*
- 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, *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 {
       keys   [ 0,      .33,       .66,        1.0 ]
       values [ 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 TO Animation.set_fraction
ROUTE Animation.outValue TO XForm.rotation
```

2. Play Westminster Chimes once an hour:

```
Group { children [
  DEF Hour TimeSensor {
    discrete      TRUE
    loop          TRUE
    cycleInterval 3600.0         # 60*60 seconds == 1 hour
  }
  DEF Sounder Sound { name "http://...../westminster.mid" }
]}
ROUTE Hour.time TO Sounder.startTime
```

3. Make a grunting noise when the user runs into a wall:

```
DEF Walls Collision { children [
  Transform {
    #... geometry of walls...
  }
  DEF Grunt Sound { name "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 "vrmlscript:
            // constructor: send initial isAtLeft eventOut
            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 TO I.set_fraction
    ROUTE T.isActive TO S.isActive
    ROUTE I.outValue 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 "vrmlscript:
        // 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 TO I.set_fraction
ROUTE I.outValue 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. 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 { },
                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 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

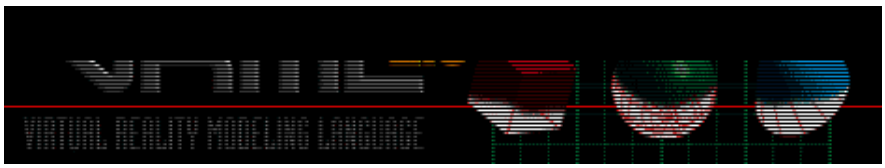
Here's another example of animating the camera. This time it's an elevator to ease access to a multistory building. For this example I'll just show a 2 story building and I'll assume that the elevator is already at the ground floor. To go up you just step inside. A BoxProximitySensor fires and starts 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 TO ElevatorPI.set_fraction
ROUTE ElevatorPI.outValue TO ETransform.set_translation
```

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

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



The Virtual Reality Modeling Language

D. Java Scripting Reference

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

This annex describes the Java classes and methods that allow scripts to interact with associated scenes. It contains links to various Java pages as well as to certain sections of the Moving Worlds VRML 2.0 Specification (including the general description in "Concepts - Scripting").

The Reference includes the following sections:

D.1 Language

D.2 Supported Protocol in the Script node's url field

D.2.1 File Extension

D.2.2 MIME type

D.3 EventIn Handling

D.3.1 Parameter passing and the EventIn Field/Method

D.3.2 EventsProcessed method

D.3.3 Shutdown method

D.4 Accessing Fields

D.4.1 Accessing Fields and EventOuts of the Script

D.4.2 Accessing Fields and EventOuts of Other Nodes

D.4.3 Sending EventOuts

D.5 Exposed Classes and Methods for Nodes and Fields

D.5.1 Field class

D.5.2 Browser class

D.5.3 User-defined classes and packages

D.6 Exceptions

D.7 Example

D.8 Class definitions

D.8.1 Class hierarchy

D.8.2 vrml package

D.9 Example of Exception Class

D.1 Language

Java(TM) is a portable, interpreted, object-oriented programming language developed at Sun Microsystems. It's likely to be the most common language supported by VRML browsers in Script nodes. A full description of Java is far beyond the scope of this appendix; see the Java web site for more information. This appendix describes only the Java bindings of the VRML API (the calls that allow the script in a VRML Script node to interact with the scene in the VRML file).

Implementing this API in VRML browsers and following examples in .wrl source files enables VRML world scenes to be animated by Java code.

D.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 ("http://foo.co.jp/Example.class").

D.2.1 File Extension

The file extension for Java byte code is **.class**.

Issue: Need to address custom protocols for Java (e.g. javabc:... and java:...).

Issue: Is "http://foo.com/Example.java" valid?

D.2.2 MIME type

The MIME type for Java byte code is defined as follows:

```
application/octet-stream
```

D.3 EventIn Handling

Events to the Script node are passed to the corresponding Java method in the script. It is necessary to specify the script in the url field of the Script node.

If a Java byte code or source code file is specified in the url field, the following two conditions must hold:

- it must contain the class definition whose name is exactly same as the body of the file name, and
- it must be a subclass of 'Script' class in vrml package.

Additionally a method must be defined which meets the following the three conditions:

- the method whose name is the same as the eventIn,
- the method is declared as 'public', and
- the method has two arguments(See 'parameter passing and the EventIn Field/Method' section).

If there isn't a corresponding Java method in the script, a browser's behavior is unspecified.

For example, the following Script node has one eventIn field whose name is 'start'.

```
    Script {
           url "Example.class"
           eventIn SFBool start
    }
```

This node points to the script file 'Example.class'. Its source('Example.java') looks like this:

```
    import vrml;
    class Example extends Script {
        ...
        public void start (ConstSFBool eventIn_value, ConstSFTime
timestamp) {
            // ... perform some operation ...
        }
    }
```

In the above example, when the start eventIn is sent the start() method is executed.

D.3.1 Parameter passing and the EventIn Field/Method

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 to the first argument and timestamp of the eventIn is passed to the second argument of the called method.

Suppose that the eventIn type is **SFXXX** and eventIn name is **eventInYYY**, then the method prototype should be

```
    public void eventInYYY(ConstSFXXX eventIn_value,
ConstSFTime timestamp)
```

Arbitrary names can be used for the arguments.

If the prototype of the method is incorrect, the browser's behavior is unspecified. The recommended behavior is that the browser should warn the incompatibility of the prototype and then stop loading.

In the above example this would be an SFBool type. Also, the time each eventIn was received is available as an SFTime value. These are passed as parameters to the Java method:

```
    public void start (ConstSFBool eventIn_value, ConstSFTime timestamp)
```

D.3.2 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 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 nondeterministic, since different implementations may call eventsProcessed at different times.

Events generated from an eventsProcessed routine are given the timestamp of the last event processed.

The prototype of eventsProcessed method is

```
public void eventsProcessed()
    throws Exception;
```

D.3.3 Shutdown method

Authors can define a shutdown method within a class that is called when the corresponding Script node is deleted. Its default behavior is no operation.

D.4 Accessing Fields

The fields, eventIns and eventOuts of a Script node are accessible from its corresponding Java class.

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. Its value can be read or written. This value is persistent across function calls. EventOuts defined in the script node can also be read.

To access fields of Script node can be done by using Script class methods. Script class has two methods to do that: getField() and getEventOut().

- Field **getField**(String *fieldName*)
  is the method to get the reference to the Script node's 'field' field whose name is *fieldName*. The return value can be converted to an appropriate Java Field class.
- Field **getEventOut**(Sting *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.

The Java Field class and its subclasses have several methods to get and set values: getValue(), setValue() or set1Value().

- **getValue**()
  converts VRML-type value into Java-type value and returns it.

- **setValue**(*value*)
  converts Java-type *value* into VRML-type value and sets it to the VRML field.
- **set1Value**(int *index*, *value*)
  It converts from Java-type *value* to VRML-type value and set it to the *index*-th element(only available for multiple value field classes).

When you call setValue() or set1Value() methods 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() or set1Value() methods 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 as specified by the associated Route in the VRML scene.

Example:

```
    Script {
        url "Example.class"
        eventIn   SFBool start
        eventOut  SFBool on
        field SFBool state TRUE
    }

    import vrml;
    class Example extends Script {
        private SFBool state = (SFBool) getField("state");
        private SFBool on = (SFBool) getEventOut("on");

        public void start (ConstSFBool eventIn_value, ConstSFTime timestamp) {
            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);
            }
        }
    }
```

D.4.2 Accessing Fields and EventOuts of Other Nodes

If a script program has an access to a previously DEF'ed VRML node, any field or eventOut of that node is accessible by using the getValue() 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 DEF'ed node. The following example shows how this is done.

```
    DEF SomeNode Transform { }
    Script {
         field SFNode node USE SomeNode
         eventIn SFVec3f pos
         url "Example.class"
    }
```

```
import vrml;
class Example extends Script {
    private SFNode node = (SFNode) getField("node");
    private SFVec3 trans;

    public void pos (ConstSFVec3 vec, ConstSFTime timestamp){
           // gets the reference to the 'translation' field of Transfrom node
            trans = (SFVec3)(node.getValue()).getValue("translation");
            trans.setValue(vec.getValue());
    }
}
```

D.4.3 Sending EventOuts

Sending eventouts is done by setting value to the reference to the 'eventOut' by setValue() or set1Value() method.

D.5 Exposed Classes and Methods for Nodes and Fields

Java classes for VRML are defined in the package *vrml*.

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 is followed by a comment indicating what type of exception will be thrown.

D.5.1 Field Class

All VRML data types have an equivalent classes in Java. The 'Field' class is the root class of all field classes.

```
class Field {
}
```

Field classes have two class types: read-only(constant)classes and writeable classes for each field types.

- **Read-only class**
  This class supports only **getValue()** method.

  ConstSFBool, ConstSFColor, ConstMFColor, ConstSFFloat, ConstMFFloat, ConstSFImage,

ConstSFInt32, ConstMFInt32, ConstSFNode, ConstMFNode, ConstSFRotation, ConstMFRotation, ConstSFString, ConstMFString, ConstSFVec2f, ConstMFVec2f, ConstSFVec3f, ConstMFVec3f, ConstSFTime

- **Writeable 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()** method.
  SFBool, SFColor, MFColor, SFFloat, MFFloat, SFImage, SFInt32, MFInt32, SFNode, MFNode, SFRotation, MFRotation, SFString, MFString, SFVec2f, MFVec2f, SFVec3f, MFVec3f, SFTime

See 'vrml package' for each classes' methods definition.

D.5.2 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 methods, see the "Browser Interface" section of the "Scripting" section of the spec.

Return value	Method name
String	**getName**()
String	**getVersion**()
float	**getCurrentSpeed()**
float	**getCurrentFrameRate**()
String	**getWorldURL**()
void	**loadWorld**(String [] url)
void	**replaceWorld**(Node[] nodes)
Node	**createVrmlFromString**(String vrmlSyntax)
Node	**createVrmlFromURL**(String[] url, Node node, String event)
String	**getNavigationType**()
void	**setNavigationType**(String type)
float	**getNavigationSpeed**()
void	**setNavigationSpeed**(float speed)
float	**getNavigationScale**()
void	**setNavigationScale**(float scale)
boolean	**getHeadlight**()
void	**setHeadlight**(boolean onOff)
String	**getWorldTitle**()
void	**setWorldTitle**(String title)
void	**addRoute**(Node fromNode, String fromEventOut, Node toNode, String toEventIn)
void	**deleteRoute**(Node fromNode, String fromEventOut, Node toNode, String toEventIn)

See ’vrml package’ for each methods definition.

Conversion table from the types used in browser class to Java type.

VRML type	Java type
SFString	String
SFFloat	float
MSString	String[]
MFNode	Node[]

D.5.3 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 for with the relative path from the URL the world was loaded from.

D.6 Exceptions

Java methods may generate the following exceptions:

- **InvalidFieldException**
  is generated at the time getField() is executed and the field name is invalid.
- **InvalidEventInException**
  is generated at the time postEventIn() is executed and the eventIn is invalid.
- **InvalidEventOutException**
  is generated at the time getEventOut() is executed and the eventOut is invalid.
- **InvalidVRMLException**
  is generated at the time createVrmlFromString() is executed and the vrml string is invalid.
- **InvalidRouteException**
  is generated at the time addRoute() or deleteRoute() is executed and one more the arguments is invalid.
- **InvalidNavigationTypeException**
  is generated at the time setNavigationType() is executed and the navigation type is invalid.
- **ArrayIndexOutOfBoundsException**
  is generated at the time setValue() or set1Value() 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’ section.

D.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;

class ExampleScript extends Script {

  // Declare field(s)
  private SFColor currentColor = (SFColor) getField("currentColor");

  // Declare eventOut field(s)
  private SFBool isRed = (SFBool) getEventOut("isRed");

  public void colorIn(ConstSFColor newColor, ConstSFTime ts) {
    // This method is called when a colorIn event is received
    currentColor.setValue(newColor.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 the "Execution Model" section of the "Concepts" document.

Browser class example

- createVrmlFromString method

```
DEF Example Script {
  url "Example.class"
  field   MFString url "foo.wrl"
  eventIn MFNode   nodesLoaded
  eventIn SFBool   trigger_event
}

import vrml;
```

```
class Example extends Script {
    private MFString url = (MFString)getField("url");

    public void trigger_event(ConstSFBool value, ConstSFTime ts){
        // do something and then fetch values
        createVRMLFromURL(url.getValue(), this, "nodesLoaded");
    }

    public void nodesLoaded(ConstMFNode value, ConstSFTime timestamp){
        // do something
    }
}
```

- addRoute method

```
DEF Sensor TouchSensor {}
DEF Baa Script {
  url "Example.class"
  field   SFNode fromNode USE Sensor
  eventIn SFBool clicked
  eventIn SFBool trigger_event
}

import vrml;

class Example extends Script {
    private SFNode fromNode = (SFNode) getField("fromNode");

    public void trigger_event(ConstSFBool eventIn_value, ConstSFTime timestamp)
        // do something and then add routing
        addRoute(fromNode.getValue(), "isActive", this, "clicked");
    }
    public void clicked(ConstSFBool value, ConstSFTime ts){
        // do something
   }
}
```

D.8 Class definitions

D.8.1 Class hierarchy

The vrml package class hierarchy looks like this:

```
vrml package
     |
     +- Field -+- ConstSFBool
     |         +- ConstSFColor
     |         +- ConstMFColor
     |         +- ConstSFFloat
     |         +- ConstMFFloat
     |         +- ConstSFImage
     |         +- ConstSFInt32
     |         +- ConstMFInt32
     |         +- ConstSFNode
     |         +- ConstMFNode
```

```
|          +- ConstSFRotation
|          +- ConstMFRotation
|          +- ConstSFString
|          +- ConstMFString
|          +- ConstSFVec2f
|          +- ConstMFVec2f
|          +- ConstSFVec3f
|          +- ConstMFVec3f
|          +- ConstSFTime
|          |
|          +- SFBool
|          +- SFColor
|          +- MFColor
|          +- SFFloat
|          +- MFFloat
|          +- SFImage
|          +- SFInt32
|          +- MFInt32
|          +- SFNode
|          +- MFNode
|          +- SFRotation
|          +- MFRotation
|          +- SFString
|          +- MFString
|          +- SFVec2f
|          +- MFVec2f
|          +- SFVec3f
|          +- MFVec3f
|          +- SFTime
|
+- Browser
+- Node(interface) -+- Script
```

D.8.2 vrml package

```
package vrml;

public class Field {
}

//
// Read-only (constant) classes, one for each field type:
//

public class ConstSFBool extends Field {
  public boolean getValue();
}

public class ConstSFColor extends Field {
  public float[] getValue();
}

public class ConstMFColor extends Field {
  public float[][] getValue();
}

public class ConstSFFloat extends Field {
  public float getValue();
}
```

```
public class ConstMFFloat extends Field {
  public float[] getValue();
}

public class ConstSFImage extends Field {
  public byte[] getValue(int[] dims);
}

public class ConstSFInt32 extends Field {
  public int getValue();
}

public class ConstMFInt32 extends Field {
  public int[] getValue();
}

public class ConstSFNode extends Field {
  /* ***************************************
   * Return value must implement Node interface.
   * The concrete class is implementation dependent
   * and up to browser implementation.
   *************************************** */
  public Node getValue();
}

public class ConstMFNode extends Field {
  public Node[] getValue();
}

public class ConstSFRotation extends Field {
  public float[] getValue();
}

public class ConstMFRotation extends Field {
  public float[][] getValue();
}

public class ConstSFString extends Field {
  public String getValue();
}

public class ConstMFString extends Field {
  public String[] getValue();
}

public class ConstSFVec2f extends Field {
  public float[] getValue();
}

public class ConstMFVec2f extends Field {
  public float[][] getValue();
}

public class ConstSFVec3f extends Field {
  public float[] getValue();
}

public class ConstMFVec3f extends Field {
  public float[][] getValue();
}
```

```
public class ConstSFTime extends Field {
  public double getValue();
}

//
// And now the writeable versions of the above classes:
//

public class SFBool extends Field {
  public boolean getValue();
  public void setValue(boolean value);
}

public class SFColor extends Field {
  public float[] getValue();
  public void setValue(float[] value)
    throws ArrayIndexOutOfBoundsException;
}

public class MFColor extends Field {
  public float[][] getValue();
  public void setValue(float[][] value)
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFColor value);
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, float[] value);
    throws ArrayIndexOutOfBoundsException;
}

public class SFFloat extends Field {
  public float getValue();
  public void setValue(float value);
}

public class MFFloat extends Field {
  public float[] getValue();
  public void setValue(float[] value);
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFFloat value);
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, float value);
    throws ArrayIndexOutOfBoundsException;
}

public class SFImage extends Field {
  public byte[] getValue(int[] dims);
  public void setValue(byte[] data, int[] dims)
    throws ArrayIndexOutOfBoundsException;
}

// In Java, the int class is a 32-bit integer
public class SFInt32 extends Field {
  public int getValue();
  public void setValue(int value);
}

public class MFInt32 extends Field {
  public int[] getValue();
  public void setValue(int[] value);
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFInt32 value);
```

```
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, int value);
    throws ArrayIndexOutOfBoundsException;
}

public class SFNode extends Field {
  public Node getValue();
  public void setValue(Node node);
}

public class MFNode extends Field {
  public Node[] getValue();
  public void setValue(Node[] node);
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFNode node);
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, Node node);
    throws ArrayIndexOutOfBoundsException;
}

public class SFRotation extends Field {
  public float[] getValue();
  public void setValue(float[] value)
    throws ArrayIndexOutOfBoundsException;
}

public class MFRotation extends Field {
  public float[][] getValue();
  public void setValue(float[][] value)
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFRotation value);
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, float[] value);
    throws ArrayIndexOutOfBoundsException;
}

// In Java, the String class is a Unicode string
public class SFString extends Field {
  public String getValue();
  public void setValue(String value);
}

public class MFString extends Field {
  public String[] getValue();
  public void setValue(String[] value);
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFString value);
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, String value);
    throws ArrayIndexOutOfBoundsException;
}

public class SFTime extends Field {
  public double getValue();
  public void setValue(double value);
}

public class SFVec2f extends Field {
  public float[] getValue();
  public void setValue(float[] value)
    throws ArrayIndexOutOfBoundsException;
```

```
}

public class MFVec2f extends Field {
  public float[][] getValue();
  public void setValue(float[][] value)
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFVec2f value);
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, float[] value);
    throws ArrayIndexOutOfBoundsException;
}

public class SFVec3f extends Field {
  public float[] getValue();
  public void setValue(float[] value)
    throws ArrayIndexOutOfBoundsException;
}

public class MFVec3f extends Field {
  public float[][] getValue();
  public void setValue(float[][] value)
    throws ArrayIndexOutOfBoundsException;
  public void setValue(ConstMFVec3f value);
    throws ArrayIndexOutOfBoundsException;
  public void set1Value(int index, float[] value);
    throws ArrayIndexOutOfBoundsException;
}

//
// Interfaces
// (http://java.sun.com/1.0alpha3/doc/javaspec/javaspec_6.html)
 (abstract classes that your classes can inherit from
// but that you can't instantiate) relating to events and nodes:
//

interface EventIn {
  public String getName();
  public SFTime getTimeStamp();
  public ConstField getValue();
}

//
// This is the general Node interface
//
public interface Node {
  public Field getValue(String fieldName)
    throws InvalidFieldException;
  public void postEventIn(String eventName, Field eventValue)
    throws InvalidEventInException;
}

//
// 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; thus, the exception codes for those exceptions are to be
// determined by the script author.
//

public class Script implements Node {
  public Field getValue(String fieldName)
```

```
    throws InvalidFieldException;
  public void postEventIn(String eventName, Field eventValue)
    throws InvalidEventInException;
  public void processEvents(EventsIn [] events)
    throws Exception; // Script:code is up to script author
  public void eventsProcessed()
    throws Exception; // Script:code is up to script author
  protected Field getEventOut(String eventName)
    throws InvalidEventOutException;
  protected Field getField(String fieldName)
    throws InvalidFieldException;
  public void shutdown(); // This method is called when this Script
node is deleted.
}

public class Browser {
  public static String getName();
  public static String getVersion();

  public static float getCurrentSpeed();

  public static float getCurrentFrameRate();

  public static String getWorldURL();
  public static void loadWorld(String [] url);

  public static void replaceWorld(Node[] nodes);

  public static Node[] createVrmlFromString(String vrmlSyntax);
    throws InvalidVRMLException;

  public static void createVrmlFromURL(String[] url, Node node,
String event);
    throws InvalidVRMLException;

  public static String getNavigationType();
  public static void setNavigationType(String type)
    throws InvalidNavigationTypeException;

  public static float getNavigationSpeed();
  public static void setNavigationSpeed(float speed);

  public static float getNavigationScale();
  public static void setNavigationScale(float scale);

  public static boolean getHeadlight();
  public static void setHeadlight(boolean onOff);

  public static String getWorldTitle();
  public static void setWorldTitle(String [] title);

  public static void addRoute(Node fromNode, String fromEventOut,
    Node toNode, String toEventIn)
    throws InvalidRouteException;
  public static void deleteRoute(Node fromNode, String fromEventOut,
    Node toNode, String toEventIn)
    throws InvalidRouteException;
}
```

D.9 Example of exception class

```
public class InvalidEventInException extends Exception
{
    /**
     * 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 Inv lideEventInException(String s) {
        super(s);
    }
}

public class InvalidEventOutException extends Exception
{
    public InvalidEventOutException() {
        super();
    }

    public InvalidEventOutException(String s) {
        super(s);
    }
}

public class InvalidFieldException extends Exception
{
    public InvalidFieldException() {
        super();
    }

    public InvalidFieldException(String s) {
        super(s);
    }
}

public class InvalidNavigationTypeException extends Exception
{
    public InvalidNavgationTypeException() {
        super();
    }

    public InvalidNavgationTypeException(String s) {
        super(s);
    }
}

public class InvalidRouteException extends Exception
{
    public InvalidRouteException() {
        super();
    }
```

```
    public InvalidRouteException(String s) {
        super(s);
    }
}

public class InvalidVRMLException extends Exception
{
    public InvalidVRMLException() {
        super();
    }

    public InvalidVRMLException(String s) {
        super(s);
    }
}
```

Contact matsuda@arch.sony.co.jp, 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 Specification

E. JavaScript Scripting Reference

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

*** IMPORTANT NOTE ***

This section needs to be tightened up. The name VRMLScript has been changed to JavaScript. However, the current text of this section does not describe the full JavaScript implementation or refer to any formal JavaScript documents.

VRML 2.0 does NOT require any scripting language. However, if a browser chooses to implement JavaScript, it shall adhere to the specifications in this annex. This section describes integrating JavaScript with VRML 2.0. It provides functions called when events come into the Script, access to fields within the script, logic to operate on the fields and the ability to send events out from the script. The Reference includes the following sections:

E.1 Script Syntax and Structure

- E.1.1 BNF of script syntax
- E.1.2 Supported Protocol in the Script Node

E.2 EventIn Handling

- Parameter passing
- EventsProcessed function

E.3 Accessing Fields

- Data Types
- Accessing Fields and EventOuts of the Script
- Accessing Fields and EventOuts of Other Nodes
- Sending EventOuts

E.4 Statements

- Conditional Statements
- Looping Statements
- Expression Statements
- Return Statement

E.5 Expressions

- Assignment Expressions
- Logical Expressions
- Arithmetic Expressions

E.6 Built-In Objects

- Browser Object
- Math Object
- Date Object

E.1 Script Syntax and Structure

The script syntax is based on JavaScript. Currently no BNF for JavaScript is available so the BNF for the language is presented here. Currently differences include:

- Semicolon is required at the end of expressions (in JavaScript it is optional). This was done to allow parsing by YACC.
- The optional *var* keyword is not allowed.
- The object referencing syntax is not allowed (neither the '[', ']' notation nor the '.' notation)
- The *break* and *continue* syntax is not supported.

E.1.1 BNF of script syntax

script :
    functions
    NULL

functions:
    functions function
    function

function:
    function *beginFunction* **(** *args* **)** *statementBlock*

beginFunction:
    identifier

args:
    args **,** *identifier*
    identifier
    NULL

stmntblk:
    { *statements* **}**
    { }
    statement

statements :
    statements statement
    statement

statement :
    ifStatement
    forStatement
    exprStatement **;**
    returnStatement **;**

returnStatement :
    return *expr*
    return

if :
    if (*expr*) *statementBlock*
    if (*expr*) *statementBlock* **else** *statementBlock*

forStatement :
    for (*optionalExpr* **;** *optionalExpr* **;** *optionalExpr*) *statementBlock*

optionalExpr:
    expr
    NULL

expr : (*expr*)
    - *expr*
    ! *expr*
    variable **=** *expr*
    expr **==** *expr*
    expr **!=** *expr*
    expr **<** *expr*
    expr **<=** *expr*
    expr **>=** *expr*
    expr **>** *expr*
    expr **+** *expr*
    expr **-** *expr*
    expr ***** *expr*
    expr **/** *expr*
    expr **%** *expr*
    string
    number
    variable

variable :
    identifier

string:
    ' *utf8* **'**

number:
    0*{***0-7***}+*
    ... ANSI C floating point number ...
    0X*{* **0-9** *}+*
    0x*{* **0-9** *}+*

identifier:

*utf8Character { utf8 }**

utf8Character:
... any legal UTF8 character except 0-9 ...

utf8:
utf8Character
0-9

E.1.2 Supported Protocol in the Script Node

The url field of the Script node contains a URL referencing JavaScript code. The javascript: protocol allows the script to be placed inline as follows:

```
Script {
    url "javascript:
            function foo() { ... }"
}
```

The *url* field can also contain a URL to a file containing the JavaScript:

```
Script {
    url [ "http://foo.com/myScript.javascript",
          "javascript:
              function foo() { ... }",
}
```

E.2 EventIn Handling

Events to the Script node are passed to the corresponding JavaScript function in the script:

```
Script {
       eventIn SFBool start
           url "... function start() { ... perform some operation ... }"
}
```

In the above example, when the start eventIn is sent the start() function is executed.

E.2.1 Parameter passing

Each eventIn is passed a corresponding data value. In the above example this would be an SFBool type. Also, the time each eventIn was received is available as an SFTime value. These are passed as parameters to the JavaScript function:

```
url "javascript:function start(value, timestamp) { ... }"
```

The parameters can have any name. The function can have no parameters, just the value or the value and timestamp. If the function has more than two parameters the extra parameters are not filled in. To JavaScript the value is numeric with 0 being false and 1 being true. The timestamp is a floating point

value containing the number of seconds since midnight, Jan. 1, 1970.

EventsProcessed function

Some implementations of the Script node may choose to defer processing of incoming events until a later time. This could be done as an optimization to skip executing scripts that do not affect visible parts of the scene, or because rendering has taken enough time to allow several events to be delivered before execution can be done. In this case the events are processed sequentially, in timestamp order. After the last eventIn is processed, the eventsProcessed function is called. This allows lengthy operations to be performed or status checks to be made once rather than in each eventIn function. Any eventOut generated during the execution of this function have the timestamp of the last eventIn processed.

E.3 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.

E.3.1 Data Types

All VRML data types have an equivalent object in JavaScript. All MFFields can be dereferenced into their corresponding SFField using the JavaScript indexing mechanism. If a is an MFVec3f and you perform the operation "b = a[3]" then b contains an SFVec3f which is the 3rd element of a. The scalar quantities (SFInt32, SFBool, SFFloat, ...) become numeric values, SFString becomes a JavaScript String object, and the vector quantities (SFRotation, SFVec3f, ...) allow access to their individual scalar components using the JavaScript indexing mechanism. In the above example, after the operation "c = b[1]", c would contain element 1 (the Y component) of b.

E.3.2 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 persistant across function calls. EventOuts defined in the script node can also be read. The value is the last value sent. assigning to an eventOut sends that event at the end of event execution. This implies that assigning to the eventOut multiple time during one execution of the function still only sends one event and that event is the last value assigned.

E.3.3 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
    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.

E.3.4 Sending EventOuts

<TBD>

E.4 Statements

JavaScript statements are block scoped the same as other C-like languages. A statement can appear alone in the body of an ***if*** or ***for*** statement. A body with multiple statements, or *compound statement*, must be placed between '{' and '}' characters. This constitutes a new block and all variables defined in this block go out of scope at the end of the block. Statements of a compound statement much be separated by the ';' character.

Example:

```
if (a < b)
    c = d;       // simple statement

else {           // compound statement
    e = f;       // e is local to this block
    c = h + 1;
}                // e is no longer defined here
```

E.4.1 Conditional Statements

The ***if*** statement evaluates an expression, and selects one of two statements for execution. A simple ***if*** statement executes the statement following the condition if the result of the result of the expression evaluation is not 0. The ***if...else*** statement additionally executes the statement following the else clause if the result of the expression evaluation is 0.

Example

```
if (a < 0)  // simple if statement
    <statement>

if (b > 5)  // if...else statement
    <statement>
else
    <statement>
```

E.4.2 Looping Statements

The ***for*** statement contains 3 expressions which control the looping behavior, followed by a statement to which the loop is applied. It executes its first expression once before loop execution. It then evaluates its second expression before each loop and, if the expression evaluates to 0, exits the loop. It then executes the statement, followed by evaluation of the third expression. The loop then repeats, until looping is terminated, either by the second expression evaluating to 0. In typical use, the first expression initialized a loop counter, the second evaluates it, and the third increments it.

Example:

```
for (i = 0; i < 10; ++i)
    <statement>
```

E.4.3 Expression Statements

Any valid expression in JavaScript can be a statement. The 2 most common expressions are the function call and the assignment expression (see below).

E.4.4 Return Statement

The return statement does an immediate return from the function regardless of its nesting level in the block structure. If given its expression is evaluated and the result returned to the calling function.

Example:

```
if (a == 0) {
    d = 1;
    return 5 + d;
}
```

E.5 Expressions

Expressions are the basic instructions of each function. Each expression requires one or 2 values. Resultant values can be used in further expressions building up compound expressions. Precedence rules are used to order evaluation. The default rules can be overridden with the use of the '(' and ')' characters to bracket higher precedence operations. Default rules are:

<TBD>

E.5.1 Assignment Expressions

An expression of the form *expression* = *expression* assigns the result of the right-hand expression to the expression on the left-hand side. The left-hand expression must result in a variable into which a value may be stored. This includes simple identifiers, subscripting operators, and hte return value of a function call.

Examples:

```
a = 5;          // simple assignment
a[3] = 4;       // subscripted assignment
foo()[2] = 3;   // function returning an array
```

E.5.2 Logical Expressions

Logical expressions include *logical and* ('***&&***'), *logical or* ('***||***'), *logical not* ('***!***'), and the comparison operators ('***<***', '***<=***', '***==***', '***!=***', '***>=***', '***>***'). Logical not is unary, the rest are binary. Each evaluates to either 0 (***false***) or 1 (***true***). The constants ***true*** and ***false*** can also be used.

Examples:

```
a < 5
b > 0 && c > 1
!((a > 4) || (b < 6))
```

E.5.3 Arithmetic Expressions

Arithmetic expressions include *and* ('&'), *or* ('|'), *exclusive or* ('^'), *not* ('~'), *negation* ('-'), and the operators ('***+***', '***-***', '*******', '***/***', '***%***'). Not and negation are unary, the rest are binary.

Examples:

```
5 + b
(c + 5) * 7
(-b / 4) % 6
(c & 0xFF) | 256
```

E.6 Built-In Objects

Some, but not all of the built-in objects from JavaScript are supported. In particular, none of the Window objects are supported, but the String, Date and Math objects are. Additionally, JavaScript has a Browser object which contains several VRML specific methods.

E.6.1 Browser Object

The Browser object gives access to several aspects of the VRML browser. The function of each method is as described in the "Concepts - Scripting" section. Since JavaScript directly supports all VRML field types the parameters passed and the values returned are as described there. The methods on the Browser object are:

getName()	Get a string with the name of the VRML browser.
getVersion()	Get a string containing the version of the VRML browser.
getCurrentSpeed()	Get the floating point current rate at which the user is traveling in the scene.
getCurrentFrameRate()	Get the floating point current instantaneous frame rate of the scene rendering, in frames per second.
getWorldURL()	Get a string containing the URL of the currently loaded world.
loadWorld(url)	Load the passed URL as the new world. This may not return.
replaceWorld(nodes)	Replace the current world with the passed list of nodes.
createVRMLFromURL(url, node, event)	Parse the passed URL into a VRML scene. When complete send the passed event to the passed node. The event is a string with the name of an MFNode eventIn in the passed node.
createVRMLFromString(str)	Parse the passed string into a VRML scene and return a the list of root nodes from the resulting scene.
addRoute(fromNode, fromEventOut, toNode, toEventIn)	Add a route from the passed eventOut to the passed eventIn.
deleteRoute(fromNode, fromEventOut, toNode, toEventIn)	Remove the route between the passed eventOut and passed eventIn, if one exists.

E.6.2 Math Object

The math object is taken from JavaScript. It consists of the keyword ***math*** dereferenced with the functions supported by the package. Supported functions are:

<TBD>

Examples:

```
a = math.sin(0.78);
dist = math.sqrt(a*a + b*b);
```

E.6.3 Date Object

The date object is taken from JavaScript. It consists of the keyword date dereferenced with the functions

supported by the package. Supported functions are:

<TBD>

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/javascript.html.

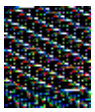
The Virtual Reality Modeling Language Specification

F. Bibliography

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

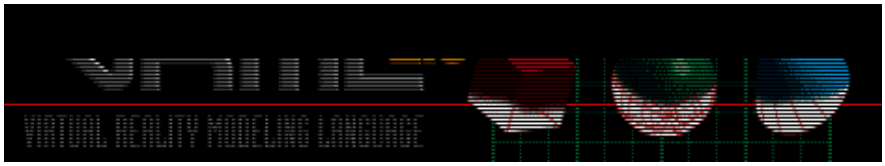
TBD: This section contains the *informative* reference list. These are references to unofficial standards or documents. All official standards are referenced in "2. Normative References section".



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/bibliography.html

---



The Virtual Reality Modeling Language Specification

Index

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

<TBD - add concepts, etc. -- make 3 column table>

Anchor
Appearance
AudioClip
Background
Billboard
Box
Collision
Color
ColorInterpolator
Cone
Coordinate
CoordinateInterpolator
Cylinder
CylinderSensor
DirectionalLight
ElevationGrid
Fog
FontStyle
Extrusion
Group
ImageTexture
IndexedFaceSet
IndexedLineSet
Inline
LOD
Material

MFColor
MFFloat
MFInt32
MFNode
MFRotation
MFString
MFTime
MFVec2f
MFVec3f
MovieTexture
NavigationInfo
Normal
NormalInterpolator
OrientationInterpolator
PixelTexture
PlaneSensor
PointLight
PointSet
PositionInterpolator
ProximitySensor
ScalarInterpolator
Script
SFBool
SFColor
SFFloat
SFImage
SFInt32
SFNode
SFRotation
SFString
SFTime
SFVec2f
SFVec3f
Shape
Sound
Sphere
SphereSensor
SpotLight
Switch
Text
TextureTransform
TextureCoordinate
TimeSensor
TouchSensor
Transform
Viewpoint
VisibilitySensor
WorldInfo

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/part1.index.html

---

Important Note: This document has not been updated to the Draft #3 spec, but is still generally useful and accurate.....rc

The Virtual Reality Modeling Language

Design Notes

Version 2.0, Official Draft #3, ISO/IEC 14772

July 15, 1996

This document describes the "why" of the Moving Worlds VRML 2.0 design -- why design decisions were made, why things were changed from VRML 1.0. It is written for armchair VRML designers and for the people who will be implementing VRML 2.0.

It contains the following sections:

- Simplifying the scene structure
- Node Design
- Prototypes
- Extensibility
- Why Routes?
- Implementing and Optimizing routes
- Scene Graph? WHAT Scene Graph?
- Transforms, Events, NodeReference
- Script API
- Materials
- Simplified Bindings

Simplifying the scene structure

There has been a lot of feedback from people implementing VRML 1.0 that the very general scene structure and property inheritance model of VRML 1.0 makes its implementation unnecessarily complex. Many rendering libraries (such as RealityLab, RenderMorphics, IRIS Performer) have a simpler notion of *rendering state* than VRML 1.0. The mismatch between these rendering libraries and VRML causes performance problems and implementation complexity, and these problems become much worse in VRML 2.0 as we add the ability to change the world over time.

To ensure that VRML 2.0 implementations are low-memory and high performance, the Moving Worlds VRML 2.0 proposal makes two major changes to the basic structure of the node hierarchy:

1. Shape properties (material, texture, shapeHints) are moved to become an integral part of the shape.
2. Transformation and Separator nodes are combined, so that a Transform defines a coordinate system relative to its parent.

To make this change, two new nodes are introduced (the Shape and Appearance nodes), several are removed (Translate, Rotate, Scale, Separator, and MatrixTransform), and a few nodes are changed (Transform, IndexedFaceSet); this change has the added benefit of making VRML simpler.

Node Design

The decisions on how to partition functionality into separate objects were motivated mainly by considerations of what should or should not be individually **sharable**. Sharing (DEF/USE in VRML 1.0, also known as 'cloning' or 'multiple instancing') is very important, since it allows many VRML scenes to be much smaller on disk (which means much shorter download times) and much smaller in memory.

One extreme would be to allow absolutely ANYTHING in the VRML file to be shared, even individual numbers of a multiple-valued field. Allowing sharing on that fine a level becomes an implementation problem if the values are allowed to change-- and the whole point of behaviors is to allow values in the scene to change. Essentially, some kind of structure must be kept for anything that can be shared that may also later be changed.

We considered allowing any field to be shared, but we believe that even that is too burdensome to implementations, since there may not be a one-to-one mapping between fields in the VRML file and the implementation's in-memory data structures.

VRML 1.0 allows nodes to be shared (via DEF/USE), and allowing sharing of any node seems reasonable, especially since events (the mechanism for changing the scene graph) are routed to nodes and because as much compatibility with VRML 1.0 as possible is one of the goals of the Moving Worlds proposal.

Shape

A new node type is introduced-- the Shape node. It exists only to contain geometry and appearance information, so that geometry+appearance may be easily shared. It contains only two fields; the **geometry** field must contain a geometry node (IndexedFaceSet, Cube, etc) and the **appearance** field may contain one or more appearance properties (Material, Texture2, etc):

```
Shape {
    field SFNode appearance
    field SFNode geometry
}
```

The three-way decomposition of shapes (Shape/Geometry/Appearance) was chosen to allow sharing of entire shapes, just a shape's geometry, or just the properties. For example, the pieces of a wooden chair and a marble table could be re-used to create a wooden table (shares the texture of the wooden chair and the geometry of the marble table) and/or to create multiple wooden chairs.

It is an error to specify the same property more than once in the appearance array, and doing so will result in undefined results.

Geometry

The existing VRML 1.0 geometry types are modified as necessary to include the geometric information needed to specify them. For example, a **vertexData** field is added to the IndexedFaceSet node to contain Coordinate3, TextureCoordinate2 and Normal nodes that define the positions, texture coordinates and normals of the IndexedFaceSet's geometry. In addition, the fields of the ShapeHints node are added to IndexedFaceSet.

These changes make it much easier to implement authoring tools that read and edit VRML files, since a Shape has a very well-defined structure with all of the information necessary to edit the shape contained inside of it. They also make VRML "cleaner"-- for example, in VRML 1.0 the only shape that pays attention to the ShapeHints node is the IndexedFaceSet. Therefore, it makes a lot of sense to put the ShapeHints information INSIDE the IndexedFaceSet.

Groups

Shapes and other "Leaf" classes (such as Viewpoints, Lights, etc) are collected into a scene hierarchy with group nodes such as Transform and LOD. Group nodes may contain only other group nodes or leaves as children; adding an appearance property or geometry directly to a group node is an error.

VRML 1.0 had a complicated model of transformations; transformations were allowed as children of group nodes and were accumulated across the children. This causes many implementation problems even in VRML 1.0 with LOD nodes that have transformations as children; the addition of behaviors would only make those problems worse.

Allowing at most one coordinate transformation per group node results in much faster and simpler implementations. Deciding which group nodes should have the transformation information built-in is fairly arbitrary; obvious choices would be either "all" or "one". Because we believe that transformations for some of the group nodes (such as LOD) will rarely be useful and maintaing fields with default values for all groups will be an implementation burden, we have chosen "one" and have added the fields of the old VRML 1.0 Transform nodes to the Transform node:

```
Transform {
    field SFVec3f    translation          0 0 0
    field SFRotation rotation             0 0 1  0
    field SFVec3f    scaleFactor          1 1 1
    field SFRotation scaleOrientation     0 0 1  0
    field SFVec3f    center               0 0 0
    field SFVec2f    textureTranslation   0 0
    field SFFloat    textureRotation      0
    field SFVec2f    textureScaleFactor   1 1
    field SFVec2f    textureCenter        0 0
}
```

These allow arbitrary translation, rotation and scaling of either coordinates or texture coordinates.

Side note: we are proposing that the functionality of the MatrixTransform node NOT be supported, since most implementations cannot correctly handle arbitrary 4x4 transformation matrices. We are willing to provide code that decomposes 4x4 matrices into the above form, which will take care of most current uses of MatrixTransform. The minority of the VRML community that truly need arbitrary 4x4 matrices can define a MatrixTransform extension with the appropriate field.

Classes

The nodes that can appear in a world are grouped into the following categories:

Groups:
Transform, LOD, Switch, Anchor, Group, Collision
Group nodes may ONLY have other groups or leaves as children.

Leaves:
Shape, Lighting, Viewpoints, Info-type nodes (WorldInfo, etc)
Leaf nodes are things that exist in one or more coordinate systems (defined by the groups that they are part of)

Geometry:
IndexedFaceSet, IndexedLineSet, PointSet, Sphere, Box, etc
Geometry nodes are contained inside Shape nodes. They in turn contain geometric properties

Geometric properties:
Coordinate, Normal, TextureCoordinate
Are contained inside geometry.

Appearance properties:
Material, *Texture
Are contained inside Appearance nodes, which are contained within Shapes, and define the shape's appearence.

Geometric Sensors:
TouchSensor, PlaneSensor, et al
Are contained inside Transforms, and generate events with respect to the Transform's coordinate system and geometry.

Inline:
Inline cuts across all of the above categories (assuming that it is useful to externally reference any of the above).

Nodes
All of the above, plus TimeSensors and Script nodes, which are not part of the world's

transformational hierarchy.
Nodes contain data (stored in fields), and may be prototyped and shared.

Why Appearance in a separate node?

Bundling properties into an Appearance node simplifies sharing, decreases file and run-time bloat and mimics modelling paradigms where one creates a palette of appearances ("materials") and then instances them when building geometry. Without Appearances, there is no easy way of creating and identifying a "shiny wood surface" that can be shared by the kitchen chair, the hardwood floor in the den, and the Fender Strat hanging on the wall.

Another major concern of VRML in general and the Appearance node in particular is expected performance of run-time implementations of VRML. It is important for run-time data structures to closely correspond to VRML; otherwise browsers are likely to maintain 2 distinct scene graphs, wasting memory as well as time and effort in keeping the 2 graphs synchronized.

The Appearance node offers 2 distinct advantages for implementations:

1. Memory is saved in the Shape node since it has only a single pointer to an Appearance node rather than many pointers to individual property nodes. In my experience there are orders-of-magnitude more Shape nodes than Appearance nodes so any memory bloat in Shape is a problem. As VRML expands to include more property nodes, Shape bloat becomes even more of an issue.
2. The Appearance node facilitates state sorting and other optimizations that are applicable to both hardware and software implementations. For example, an implementation can quickly determine if 2 shapes have the same appearance by checking for pointer equality rather than comparing every property reference of a shape. As another exmple, the Appearance node offers a good place to maintain a cache which in the case of a software implementation may be a pre-wired path of rendering modules.

Prototypes

There are several different ways of thinking about prototypes:

- An extensibility mechanism that allows a new node to be defined in terms of other, pre-defined nodes. Prototypes can replace all of the existing VRML 1.0 extension mechanisms (isA and fields[]).
- A protection mechanism that allows an author to limit what can be done to an object. EXTERNPROTO allows authors to specify how their worlds/objects/behaviors can be used within other worlds/objects/behaviors.
- An object-definition mechanism that allows somebody to define objects with specific (standardized) operations allowed. Prototypes allow application-specific **policies** to be imposed on the general scene structure.
- A convenience mechanism that allows geometry and/or behavior to be packaged in a convenient way.
- An optimization mechanism that allows browsers to reason about which object can or cannot be

changed.

- A bandwith-saving mechanism that allows the definition of a world **structure** to be defined once and re-used multiple times.

The prototype declaration

A prototype's interface is declared using one of the following syntaxes:

```
PROTO name [ field    fieldType name defaultValue
             eventIn  fieldType name
             eventOut fieldType name
           ] { implementation }
EXTERNPROTO name [ field    fieldType name
                   eventIn  fieldType name
                   eventOut fieldType name
                 ] URL(s)
```

(there may be any number of field/eventIn/eventOut declarations in any order).

A prototype just declares a new kind of node; it does not create a new instance of a node and insert it into the scene graph, that must be done by instantiating a prototype instance.

First, why do we need to declare a prototype's interface at all? We could just say that any fields, eventIns or eventOuts of the nodes inside the prototypes implementation exposed using the **IS** construct (see below) are the prototype's interface. As long as the browser knows the prototype's interface it can parse any prototype instances that follow it.

The declarations are necessary for EXTERNPROTO because a browser may not be able to get at the prototype's implementation. Also requiring them for PROTO makes the VRML file both more readable (it is much easier to see the PROTO declaration rather than looking through reams of VRML code for nodes with IS) and makes the syntax more consistent.

Default values must be given for a prototype's fields so that they always have well-defined values (it is possible to instantitate a prototype without giving values for all of its fields, just like any other VRML node). Default values must not be specified for an EXTERNPROTO because the default values for the fields will be defined inside the URL that the EXTERNPROTO refers to.

EXTERNPROTO refers to one or more URLs, with the first URL being the preferred implementation of the prototype and any other URLs defining less-desireable implementations. Browsers will have to be able to deal with the possibility that an EXTERNPROTO's implementation cannot be found because none of the URL's are available (or the URL array is empty!); browsers may also decide to "delay-load" a prototype's implementation until it is actually needed (like they do for the VRML 1.0 WWWInline node).

Browsers can properly deal with EXTERNPROTO instances without implementations. Events will never be generated from such instances, of course, so that isn't a problem. The browser can decide to either throw away any events that are routed to such an instance or to queue them up until the implementation does become available. If it decides to queue them up, the results when they're finally processed by the prototype's implementation could be indeterminate IF the prototype generates output events in response to the input events. A really really smart browser could deal with this case by

performing event rollback and roll-forward, re-creating the state of the world (actually, only the part of the world that can possibly be influenced by the events generated from the prototype need to be rolled forward/back) when the events were queued and "re-playing" input events from there.

The fields of a prototype are internal to it, and a browser needs to know their current and default values only to properly create a prototype instance. Therefore, if the browser cannot create prototype instances (because the prototype implementation is not available) the default values of fields aren't needed. So, EXTERNPROTO provides all the information a browser needs.

The prototype implementation

The prototype's implementation is surrounded by curly braces to separate it from the rest of the world. A prototype's implementation creates a new **name scope** -- any names defined inside a prototype implementation are available only inside that prototype implementation. In this way a prototype's implementation can be thought of as if it is a completely separate file. Which, of course, is exactly what EXTERNPROTO does.

There's an interesting issue concerning whether or not things defined **outside** the prototype's implementation can be USEd inside of it. We think that defining prototypes such that they are completely self-contained (except for the information passed in via eventIn or field declarations) is wisest.

The **node type** of a prototype is the type of the first node of its implementation. So, for example, if a prototype's implementation is:

```
{ IndexedFaceSet { ... } }
```

Then the prototype can only be used in the scene wherever an IndexedFaceSet can be used (which is in the geometry field of a Shape node). The extra curly braces allow Scripts, TimeSensors and ROUTES to be part of the prototype's implementation, even though they're "off to the side" of the prototype's scene graph.

The **IS** syntax for specifying what is exposed inside a prototype's implementation was suggested by Conal Elliott of Microsoft. It was chosen because:

- it removed some ambiguities that could arise about what a prototype's field's default values were in a previous syntax
- it doesn't require nodes to be given names just to expose them in the prototype
- it allows fan-in (one eventIn or field going to multiple nodes in the prototype implementation)
- it matches traditional programming languages better

Instantiating a prototype

Once a PROTO or EXTERNPROTO has been declared, a prototype can be instantiated and treated just like any built-in node. In fact, built-in nodes can just be treated as if there are a set of pre-defined PROTO definitions available at start-up in all VRML browsers.

Each prototype instance is independent from all others-- changes to one instance do not affect any other instance. Conceptually, each prototype instance is equivalent to a completely new copy of the prototype implementation.

However, even though prototype instances are conceptually completely separate, they can be implemented so that information is automatically shared between prototype instances. For example, consider this PROTO:

```
PROTO Foo [ eventIn SFVec3f changeTranslation ] {
    Transform {
        translation IS changeTranslation
        Shape {
           ... geometry+properties stuff...
        }
    }
}
```

Because the translation of the Transform is the only thing that can possibly be changed, either from a ROUTE or from a Script node, only the Transform needs to be copied. The same Shape node may be shared by all prototype instances.

Script nodes that contain SFNode/MFNode fields (or may receive SFNode/MFNode events) can be treated in a similar way; for example:

```
PROTO Foo [ eventIn SFFloat doSomething ] {
   DEF Root Transform {
      ... stuff ...
   }
   DEF MyScript Script {
      eventIn doIt IS doSomething
      field SFNode whatToAffect USE Root
        ... other script stuff...
   }
}
```

In this case, a brand-new copy of everything inside Foo will have to be created for every prototype instance because MyScript may modify the Root Transform or any of it children using the script API. Of course, if some of the Transform's children are prototype instances the browser might be able to optimize **them**.

Issue: If we can get users to use something like this prototype definition, browsers might have more opportunities for optimization:

```
# A Transform that cannot be changed:
#
PROTO ConstantTransform [
       field MFNode children
       field SFVec3f translation 0 0 0 ... etc for other fields...
   ] {
       Transform { children IS children
                   translation IS translation  ... etc ...
       }
}
```

We can imagine variations on the above-- Transforms with transformations that can be changed, but children that can't, transformations that can't but children that can, etc.

Extensibility

By extending the syntax of a URL in an EXTERNPROTO, all of the current and proposed extensibility mechanisms for VRML can be handled (credit for these ideas go to Mitra).

The idea is to use the URL syntax to refer to an internal or built-in implementation of a node. For example, imagine your system has a Torus geometry node built-in. The idea is to use EXTERNPROTO to declare that fact, like this:

```
EXTERNPROTO Torus [ field SFFloat bigRadius
                    field SFFloat smallRadius ]
  "internal:Torus"
```

URL's of the form "internal:name" tell the browser to look for a "native" implementation (perhaps searching for the implementation on disk, etc).

Just as in any other EXTERNPROTO, if the implementation cannot be found the browser can safely parse and ignore any prototype instances.

The 'alternateRep' notion is handled by specifying multiple URLs for the EXTERNPROTO:

```
EXTERNPROTO Torus [ field SFFloat bigRadius
                    field SFFloat smallRadius ]
  [ "internal:Torus", "http://machine/directory/protofile" ]
```

So, if a "native" implementation of the Torus can't be found, an implementation is downloaded from the given machine/directory/protofile-- the implementation would probably be an IndexedFaceSet node with a Script attached that computes the geometry of the torus based on bigRadius and smallRadius.

The 'isA' notion of VRML 1.0 is also handled using this mechanism. The ExtendedMaterial example from the VRML 1.0 spec:

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

becomes:

```
PROTO ExtendedMaterial [
   field MFFloat indexOfRefraction 0
   field MFColor ambientColor [ 0 0 0 ]
   field MFColor diffuseColor [ .8 .8 .8 ]
     ... etc, rest of fields... ]
{
    Material {
       ambientColor IS ambientColor
       diffuseColor IS diffuseColor
```

```
        ... etc ...
    }
}

ExtendedMaterial {
    indexOfRefraction .34
    diffuseColor .8 .54 1
}
```

This nicely cleans up the rules about whether or not the fields of a new node must be defined only the first time the node appears inside a file or every time the node appears in the file (the PROTO or EXTERNPROTO must appear one before the first node instance). And it makes VRML simpler.

Why Routes?

Several different architectures for applying changes to the scene graph were considered before settling on the ROUTE syntax. This section documents the arguments for and against the alternative architectures.

All-API architecture

One alternative is to try to keep all behaviors out of VRML, and do everything inside the scripting API.

In this model, a VRML file looks very much like a VRML 1.0 file, containing only static geometry. In this case, instead of loading a .wrl VRML file into your browser, you would load some kind of .script file that then referenced a .wrl file and then proceeded to modify the objects in the .wrl file over time. This is similar to conventional programming; the program (script) loads the data file (VRML .wrl file) and then proceeds to make changes to it over time.

One advantage of this approach is that it makes the VRML file format simpler. A disadvantage is that the scripting language may need to be more complex.

The biggest disadvantage, however, is that it is difficult to achieve good optimizibility, scalability and composability-- three of our most important goals.

In VRML 1.0, scalability and composability are accomplished using the WWWInline node. In an all-API architecture, some mechanism similar to WWWInline would have to be introduced into the scripting language to allow similar scalability and composability. That is certainly possible, but putting this functionality into the scripting language severely affects the kinds of optimizations that browsers are able to perform today.

For example, the browser can pay attention to the direction that a user is heading and pre-load parts of the world that are in that direction **if the browser knows where the WWWInline nodes are**. If the WWWInline concept is moved to the scripting language the browser probably will NOT know where they are.

Similarly, a browser can perform **automatic behavior culling** if it knows which parts of the scene may

be affected by a script. For example, imagine a lava lamp sitting on a desk. There is no reason to simulate the motion of the blobs in the lamp if nobody is looking at it-- the lava lamp has a completely self-contained behavior. In an API-only architecture, it would be impossible for the browser to determine that the behavior was self-contained; however, with routes, the browser can easily determine that there are no routes into or out of the lava lamp, and that it can therefore be safely behavior culled. (side note: we do propose flags on Scripts for cases in which it is important that they NOT be automatically culled).

Another disadvantage to this approach is that it allows only re-use of geometry. Because the behaviors must directly load the geometry, it is impossible to "clone" a behavior and apply it to two different pieces of geometry, or to compose together behavior+geometry that can then be re-used several times in the same scene.

The disconnect between the VRML file and the script file will make revision control painful. When the VRML file is changed, the script may or may not have to be changed-- in general, it will be very difficult for a VRML authoring system to maintain worlds with behaviors. If the VRML authoring system cannot parse the scripting language to find out what it referrs to in the VRML file, then it will be impossible for the authoring system to ensure that behaviors will continue to work as the VRML file is edited.

All-VRML architecture

Another alternative is to extend VRML so that it becomes a complete programming language, allowing any behavior to be expressed in VRML.

The main disadvantage to this approach is that it requires inventing Yet Another Scripting Language, and makes implementation of a VRML browser much more complicated. If the language chosen is very different from popular languages, there will be very few people capable of programming it and very little infrastructure (classes, books, etc) to help make it successful.

Writing a VRML authoring system more sophisticated than a simple text editor becomes very difficult if a VRML file may contain the equivalent of an arbitrary program. Creating ANY VRML content becomes equivalent to programming, which will limit the number of people able to create interesting VRML worlds.

The main advantage to an all-VRML architecture is the opportunity for automatic optimizations done by the browser, since the browser knows everything about the world.

Routes and Script nodes architecture

The alternative we chose was to treat behaviors as "black boxes" (Script nodes) with well-defined interfaces (routes and fields).

Treating behaviors as black boxes allows any scripting language to be used (Java, VisualBasic, ML, whatever) without changing the fundamental architecture of VRML. Implementing a browser becomes much easier because only the interface between the scene and the scripting language needs to be implemented, not the entire scripting language.

Expressing the interface to behaviors in the VRML file allows an authoring system to intelligently deal with the behaviors, and allows most world creation tasks to be done with a graphical interface. A programming editor only need appear when a sophisticated user decides to create or modify a behavior (opening up the black box, essentially). The authoring system can safely manipulate the scene hierarchy (add geometry, delete geometry, rename objects, etc) without inadvertently breaking connections to behaviors.

The existing VRML composability and scalability features are retained, and because the possible effects of a behavior on the world are known to the browser, most of the optimizations that can be done in an all-VRML architecture can still be done.

Implementing and Optimizing routes

This section gives some "thumb-nail" design for how a browser might decide to implement routes. It points out some properties of the routes design that are not obvious at first glance and that can make an implementation of routes simple and efficient.

There doesn't need to be any data copying at all as an event "travels" along a route. In fact, the event doesn't need to "travel" at all-- the ROUTE is really just a re-naming from the eventIn to the eventOut that allows the composability, authorability, extensibility and scalability that are major goals of the Moving Worlds design.

The data for an event can be stored at the source of the event-- with the "eventOut". The "eventIn" doesn't need to store any data, because it is impossible to change an "eventIn"-- it can just point to the data stored at the "eventOut". That means that moving an event along a ROUTE can be as cheap as writing a pointer. In fact, in the VERY common case in which there is no "fan-in" (there aren't multiple eventOut's routed into a single eventIn) NO data copying at all need take place-- the eventIn can just point to eventOut since that eventOut will always be the source of its events.

Exposed fields-- fields that have corresponding eventOut's-- can share their value between the eventOut and the field itself, so very little extra overhead is imposed on "exposed" fields. Highly optimized implementations of nodes with exposed fields could store the data structures needed to support routes separately from the nodes themselves and use a dictionary mapping node pointers to routing structures, adding NO memory overhead for nodes that do not have routes coming into or out of them (which is the common case).

Because the routing structures are known to the browser, many behavior-culling optimizations are possible. A two-pass notification+evaluation implementation will automatically cull out any irrelevant behaviors without any effort on the part of the world creator. The algorithm works by delaying the execution of behaviors until their results are necessary, as follows:

Imagine a TimeSensor that sends alpha events to a Script that in turn sends setDiffuseColor events to an object, to change the object's color over time. Allocate one bit along each of these routes; a "dirty bit" that determines whether or not changes are happening along that route. The algorithm works as follows:

1. Time changes. All routes from the TimeSensor are marked dirty, all the way through the routing network (from the TimeSensor, to the Script, to the object who's material we're changing). This "notification" process can stop as soon as a route that has already been marked "dirty" is reached. Most browsers will probably let notification continue up through the children[] MFNode fields of groups; if the notification eventually reaches the root of the scene the the browser will know that the scene must be redrawn.
2. When (or before) the browser redraws the scene, any object that will be drawn that has a route to it with its dirty bit set must be re-evaluated by evaluating whatever is connected to it up-stream. When a route is evaluated, its dirty bit is cleared.

This two-pass "push notification / pull events" algorithm has several nice properties:

- If an object cannot be perceived, its behavior does not need to be run. In the example of the changing material, if the object is never drawn, its material will not be needed. The TimeSensor may change, but the route from the TimeSensor to the Script will be dirty, so notification will stop right away. Note that TimeSensor is carefully defined such that only the last "time changed" event is guaranteed to be available, so you don't even need to queue up all of the "time changed" events. In the case of a ClickSensor or something else where every event might be important, either events will have to be queued up OR scripts will have to be run as soon as there is more than one event waiting.
- If the behavior is affecting something like a transformation, then it will automatically always get run IF that transformation is needed to compute the perceptibility of the object. If a maximum possible bounding box can be specified by the world creator, then even behaviors that affect transformations need not get run.
- If the behavior is affecting something that cannot be seen, the scene doesn't need to be redrawn. And it won't be, because the notification process will stop at the first "dirty" route and will never reach the root of the scene graph.
- If the view changes so that something now CAN be seen, the scene will be redraw (because the view changed) and portions of the scene that were marked "dirty" that are now visible will be "clean".
- If you have to shadow things that are changing in your rendering library, the dirty bits will tell you exactly what changed so you can update your rendering library.

Scene Graph? WHAT Scene Graph?

Moving Worlds has been carefully designed so that a browser will only need to keep the parts of the VRML scene graph that might be changed. There is a tradeoff between world creators who want to have control over the VRML scene graph structure and browser implementors who also want complete control over the VRML scene graph structure; Moving Worlds is designed to compromise between these two, allowing world creators to impose a particular structure on selected parts of the world while allowing browsers to optimize away the rest.

One example of this is the routing mechanism. Consider the following route:

```
Shape {
    appearance Appearance {
```

```
        material DEF M Material { ... }
    geometry Cube { }
}
ROUTE  MyAnimation.color -> M.setDiffuseColor
```

A browser implementor might decide not to maintain the Material as a separate object, but instead to route all setDiffuseColor events directly to the relevant shape(s). If the Material was used in several shapes then several routes might need to be established where there was one before, but as long as the visual results are the same the browser implementor is free to do that.

There is a potential problem if some Script node has a pointer to or can get a pointer to the Material node. In that case, there _will_ need to be at least a stand-in object for the Material (that forwards events on to the appropriate shapes) IF the Script might directly send events to what it thinks is the Material node. However, Script nodes that do this MUST set the "directOutputs" flag to let the browser know that it might do this. And the browser will know if any Script with that flag set can get access to the Material node, because the only way Scripts can get access to Nodes is via a field, an eventIn, or by looking at the fields of a node to which it already has access.

World creators can help browsers by limiting what Script nodes have access to. For example, a browser will have to maintain just about the entire scene structure of this scene graph:

```
DEF ROOT Transform {
    children [
          Shape { ... geometry Sphere{ } },
          Transform {
             ... stuff ...
          }
    ]
}
Script {
    directOutputs TRUE
    field SFNode whatToChange USE ROOT
    ...
}
```

Because the Script has access to the root of the scene, it can get the children of that root node, send them events directly, add children, remove children, etc.

However, this entire scene can be optimized below the Transform, because the browser KNOWS it cannot change:

```
PROTO ConstTransform [ field MFNode children ] {
    Transform { children IS children }
}
DEF ROOT ConstTransform {
    children [
          Shape { ... geometry Sphere{ } },
          Transform {
             ... stuff ...
          }
    ]
}
Script {
    unknownOutputs TRUE
    field SFNode whatToChange USE ROOT
```

```
    ...
}
```

Because of the prototype interface, the browser KNOWS that the Script cannot affect anything inside the ConstTransform-- the ConstTransform has NO exposed fields or eventIn's. If the ConstTransform doesn't contain any sources of changes (Sensors or Scripts), then the entire subgraph can be optimized away-- perhaps stored ONLY as a display list for a rendering library, or perhaps collapsed into a "big bag of triangles" (also assuming that there are no LOD's, of course).

The other nice thing about all this is that a PROTO or EXTERNPROTO (or WWWInline, which is pretty much equivalent to a completely opaque prototype) can be optimized independently of everything else, and the less control an author gives over how something might be changed, the more opportunities for optimizations.

Transforms, Events, NodeReference

The children of a Transform (or other group node) are kind of strange-- they aren't specified like fields in the VRML 1.0 syntax.

Issue: They could be-- they are functionally equivalent to an MFNode field. For example, this:

```
# Old syntax?
Transform {
    Transform { ... }
    Transform { ... }
}
```

is equivalent to the slightly wordier:

```
# New syntax?
Transform {
  children [
    Transform { ... } ,
    Transform { ... }
  ]
}
```

... where "children" is an MFNode field. The issue is whether or not we should keep the VRML 1.0 syntax as a convenient short-hand that means the same as the wordier syntax. The advantages are that it would make the VRML file syntax easier to parse and would eliminate some ambiguities that can arise if fields and nodes are allowed to have the same type names. The disadvantages are that it would make VRML files slightly bigger, is less convenient to type in, and is a change from VRML 1.0 syntax.

In any case, to allow grouping nodes to be used as prototypes and to allow them to be seen in the script API, their children must "really" be an MFNode field. So a Transform might be specified as:

```
PROTO Transform [
    field SFVec3f translation 0 0 0
    eventIn SFVec3f setTranslation
    eventOut SFVec3f translationChanged
```

```
        ... etc for the other transformation fields...
    field MFNode children [ ]
    eventIn MFNode setChildren
    eventOut MFNode childrenChanged
] ...
```

Specifying events corresponding to the children field implies that the children of a Transform can change-- that the structure of the scene can be changed by behaviors.

Setting all of the children of a Transform at once (using setChildren) is inconvenient; although not strictly necessary, the following might be very useful:

```
    eventIn MFNode addChildren
    eventIn MFNode removeChildren
```

Sending an addChildren event to the Transform would add all of the children in the message to the Transform's children. Sending a removeChildren event would remove all of the children in the message (little tiny issue: maybe SFNode addChild/removeChild events would be better?).

The Transform node's semantics were carefully chosen such that the order of its children is irrelevant. That allows a lot of potential for implementations to re-order the children either before or during rendering for optimization purposes (for example, draw all texture-mapped children before all non-texture mapped children, or sort the children by which region of space they're in, etc). The addChildren/removeChildren events maintain this property-- anything using them doesn't need to concern itself with the order of the children.

A previous version of Moving Worlds had a node called "NodeReference" that was necessary to allow nodes to be inserted as children into the scene. Exposing the children of groups as MFNode fields eliminates the need for something like NodeReference.

Script node: Minimal API

This section describes the API from the point of view of somebody using VRML to create behaviors. At least the following functionality will be necessary:

init/destroy/processEvents
: The browser must call the user's init routine before calling processEvents or destroy.
The processEvents routine may be called any time between init and destroy, and will usually process all waiting events and generate events and/or modify the Script node's fields.
The browser must call the destroy routine to allow the script an opportunity to do cleanup. After destroy is called, processEvents must not be called until after another init is done.

get/set fields
: The fields of a script node must be accessible from the API. That implies that the VRML field types (SFFloat, MFFloat, etc) must somehow be exposed in the API.

send/receive events
: The processEvents routine must have access to a list of events received from things routed to it. Each event will have:

- Name
- Type (any of the field types)
- Value (same as field value) and API to get/set the event's contents (both get and set for events that will be output, only get for events that come in)
- Timestamp

Synchronization API
: To support scripting languages such as Java which allow the creation of asynchronous processes (threads), some mechanism for synchronizing with the browser when changing the Script's fields and generating events is necessary. At the very least, a mechanism to "bracket" or "bundle up" a set of changes is necessary.

Script node: Node API

Once a Script node has access to an SFNode or an MFNode value (either from one of the Script's fields, or from an eventIn that sends the script a node), we must decide what operations a script can perform on them. A straw-man proposal:

get/set "exposed" fields
: For any field"foo" of the node that has both a "setFoo" eventIn and a "fooChanged" eventOut, allow that field to be directly set and get. There should be API to get the list of exposed fields, of a given node.

get list of eventIn/eventOut
: Given a node, there should be a way of determining what events it can send and receive.

establish routes
: There should be some way of establishing a route from within a Script, assuming the script has somehow gotten access to the nodes on both ends of the route.

"compileVRML"
: An API call that allows VRML file format contained in a string to be "compiled" into a Node from inside a script, allowing a Script node to receive file format from over the network, for example.

communication with the browser
: The API must provide methods by which a Script node can communicate with the browser to request operations such as loading a new URL as the world (to allow WWWAnchor-like functionality controlled by a Script), to get the current "simulation" time (which may be different from the current wall-clock time), etc.

Convenience method: search by name/type
: Search for nodes by name or by type "under" a given node. Assuming that the children of group nodes is exposed in the API as an MFNode field called "children", this is really a short-hand convenience way of performing something like:

```
Node search(Node startingNode, ...criteria...) {
   for all fields of startingNode {
      if field type is SFNode {
         Node kid = contents of field
         if kid matches criteria, return kid
         else {
            Node Found = search(kid, criteria)
            if (Found != NULL) return Found
         }
      }
      else if field type is MFNode, for all values i {
         Node kid = value[i]
```

```
            if kid matches criteria, return kid
            else {
               Found = search(kid, criteria)
               if (Found != NULL) return Found
            }
          }
        }
        return NULL
    }
```

Throughout this discussion I'm assuming that access to prototyped nodes is restricted by the prototype's interface. That will allow implementations to know what can and what can't change, which will enable many optimizations.

Materials

The VRML 1.0 material specification is more general than currently supported by most 3D rendering libraries and hardware. It is also fairly difficult to explain and understand; a simpler material model will make VRML 2.0 both easier to understand and easier to implement.

First, the notion of per-vertex or per-face materials/colors should be moved from the Material node down into the geometric shapes that support such a notion (such as IndexedFaceSet). Doing this will make colors more consistent with the other per-vertex properties (normals and texture coordinates) and will make it easier for browsers to ensure that the correct number of colors has been specified for a given geometry, etc.

The new syntax for a geometry such as IndexedFaceSet will be:

```
IndexedFaceSet {
  exposedField  SFNode  coord             NULL
  exposedField  SFNode  color             NULL
  exposedField  SFNode  normal            NULL
  exposedField  SFNode  texCoord          NULL
  ...
}
```

A new node, similar to the Normal/TextureCoordinate2 nodes, is needed for the color field. It is often useful to define a single set of colors to function as a "color map" that is used by several different geometries, so the colors are specified in a separate node that can be shared. That node will be:

```
Color {
    exposedField MFColor rgb [ ]       # List of rgb colors
}
```

The material parameters in the material node would all be single-valued, and we suggest that the ambientColor term be removed:

```
Material {
  exposedField SFColor diffuseColor  0.8 0.8 0.8
  exposedField SFColor specularColor 0 0 0
  exposedField SFColor emissiveColor 0 0 0
```

```
  exposedField SFFloat shininess     0.2
  exposedField SFFloat transparency  0
}
```

If multiple colors are given with the geometry, then the they either replace the diffuse component of the Material node (if the material field of the Appearance node is not NULL) or act as an "emissive-only" source (if the material field of the Appearance node is NULL).

Issue: *The colors in a VRML SFImage field are RGBA-- RGB plus transparency. Perhaps we should allow SFColor/MFColor fields to be specified with 1, 2, 3 or 4 components to be consistent with SFImage. That would get rid of the transparency field of Material, allow transparency per-face or per-vertex, and would allow compact specification of greyscale, greyscale-alpha, RGB, and RGBA colors. However, that might cause problems for the behavior API and would make parsing more complicated.*

Simplified Bindings

Another complicated area of VRML 1.0 are all of the possible bindings for normals and materials-- DEFAULT, OVERALL, PER_PART, PER_PART_INDEXED, PER_FACE, PER_FACE_INDEXED, PER_VERTEX, and PER_VERTEX_INDEXED. Not all bindings apply to all geometries, and some combinations of bindings and indices do not make sense.

A much simpler specification is possible that gives equivalent functionality:

```
IndexedFaceSet {
  ...
  field         MFInt32 coordIndex        [ ]
  field         MFInt32 colorIndex        [ ]
  field         SFBool  colorPerVertex    TRUE
  field         MFInt32 normalIndex       [ ]
  field         SFBool  normalPerVertex   TRUE
  field         MFInt32 texCoordIndex     [ ]
  ...
}
```

The existing materialBinding/normalBinding specifications are replaced by simple booleans that specify whether colors or normals should be applied per-vertex or per-face. If indices are specified, then they are used. If they are not specified, then either the vertex indices are used (if per-vertex normals/colors), OR the normals/colors are used in order (if per-face).

In more detail:

- If normals/colors are NOT specified, the browser should:
  -- normals: generate automatically.
  -- colors: just use Material specification from Appearance (like VRML 1.0 OVERALL)
- If normals/colors are specified (non-NULL normal/color field):
  - If normal/colorPerVertex is FALSE:
    - If Index is not empty, then use indices to choose normals/colors per face (equivalent to VRML 1.0 PER_FACE_INDEXED)

        - Else use the normals/colors in order
          (equivalent to VRML 1.0 PER_FACE)
    - If normal/colorPerVertex is TRUE:
        - If Index is not empty, then use indices to choose normals/colors per vertex
          (equivalent to VRML 1.0 PER_VERTEX_INDEXED)
        - Else choose normals/colors using the coordIndex indices
          (also equivalent to VRML 1.0 PER_VERTEX_INDEXED)

Texture coordinates do not have a PerVertex flag, because texture coordinates are always specified per vertex. The rules for texture coordinates are the same as for per-vertex colors/normals: if texCoordIndex is empty, the vertex indices in coordIndex are used.

IndexedLineSet would add color and colorPerVertex fields, with similar rules to IndexedFaceSet. PointSet would need only a color field (OVERALL color if empty, otherwise color-per-point). The shapes that allow PER_PART colors in VRML 1.0 (Cylinder, Cone) would also only need a color field (PER_PART colors if specified, OVERALL otherwise).

Comparison with VRML 1.0: if all of the possibilities are written out, the only binding missing is the VRML 1.0 PER_VERTEX binding, which ignores the Index fields and just takes colors/normals in order for each vertex of each face. For example, in VRML 1.0 if the coordIndex array contained [10, 12, 14, -1, 11, 13, 10, -1] (two triangles with one shared vertex), then the PER_VERTEX binding is equivalent to a PER_VERTEX_INDEXED binding with indices [0, 1, 2, -1, 3, 4, 5, -1] -- that is, each positive entry in the coordIndex array causes another color/normal to be taken from their respective arrays. VRML 1.0 files with PER_VERTEX bindings that are converted to VRML 2.0 will be somewhat larger, since explicit indices will have to be generated.

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

This URL:http://webspace.sgi.com/moving-worlds/Design.html