The Virtual Reality Modeling Language

International Standard ISO/IEC 14772-1:1997

Copyright Information

TITLE: ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (VRML97)
URL: **http://www.vrml.org/Specifications/VRML97**
AUTHORS: Rikk Carey, Gavin Bell, Chris Marrin

INTELLECTUAL PROPERTY NOTICE

Other copyrights and trademarks

Acknowledgements

The VRML Consortium gratefully acknowledges the authors, Rikk Carey, Gavin Bell, and Chris Marrin, whose valuable efforts produced the VRML standard.

We would like to give special thanks to Steve Carson, chair of the ISO/IEC JTC 1/SC 24, *Computer Graphics and Image Processing* subcommittee, and to Dick Puk, liaison between the VRML Consortium and SC 24, for guiding the standards process as well as their significant contributions to the document itself. Also, thanks to all the members of ISO who participated in the review and editing of ISO/IEC 14772.

Special thanks to Kouichi Matsuda and the Sony VRML team for their work on the Java annex and to Jan Hardenbergh for his work on the ECMAScript annex. Thanks to Curtis Beason, Chris Fouts, John Gebhardt, Rich Gossweiler, Paul Isaacs, and Daniel Woods for writing key sections. Thanks to Justin Couch and the Script Working Group for drafting several improvements to the scripting sections. Thanks to all the others who drafted text for the standard, too numerous to name them all.

Thanks to Mark Pesce, Tony Parisi, Mitra, Brian Behlendorf, and Dave Raggett for their early pioneering work and continued efforts on VRML.

Thanks to the hundreds of participants who contributed ideas, reviews, and feedback on the VRML standard.

Thanks to Kevin Hughes for the VRML logo artwork.

And, last but not least, thanks to the members of the VRML community for their support, passion, and hard work that has made VRML into an International Standard.

VRML97

ISO/IEC 14772-1:1997

The Virtual Reality Modeling Language

International Standard ISO/IEC 14772-1:1997

This document is part 1 of ISO/IEC 14772-1:1997, the Virtual Reality Modeling Language (VRML), also referred to as "VRML97". The full title of this part of the International Standard is: *Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 1: Functional specification and UTF-8 encoding.*

Background		Clauses		Annexes	
Foreword	ii	1 Scope	1	A Grammar	141
Introduction	iii	2 Normative references	3	B Java platform	148
		3 Definitions	6	C ECMAScript	185
		4 Concepts	19	D Examples	209
		5 Field and event reference	62	E Bibliography	233
		6 Node reference	67	F Extensions	235
		7 Conformance	132		

The ***Foreword*** provides background on the standards process for VRML. The ***Introduction*** describes the purpose, design criteria, and characteristics of VRML. The following clauses define part 1 of ISO/IEC 14772:

a. ***Scope*** defines the problem area that VRML addresses.

b. ***Normative references*** lists the normative standards referenced in this part of ISO/IEC 14772.

c. ***Definitions*** contains the glossary of terminology used in this part of ISO/IEC 14772.

d. ***Concepts*** describes various fundamentals of VRML.

e. ***Field and event reference*** specifies the datatypes used by nodes.

f. ***Node reference*** defines the syntax and semantics of VRML nodes.

g. ***Conformance and minimum support requirements*** describes the conformance requirements for VRML implementations.

There are several annexes included in the specification:

A. ***Grammar definition*** presents the grammar for the VRML file format.

B. ***Java platform scripting reference*** describes how VRML scripting integrates with the Java platform.

C. ***ECMAScript scripting reference*** describes how VRML scripting integrates with ECMAScript.

D. ***Examples*** includes a variety of VRML example files.

E. ***Bibliography*** lists the informative, non-standard topics referenced in this part of ISO/IEC 14772.

F. ***Recommendations for non-normative extensions*** lists informative recommendations for extensions to VRML.

Questions or comments should be sent to rikk@wasabisoft.com.

Foreword

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form a 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. See http://www.iso.ch for information on ISO and http://www.iec.ch for information on IEC.

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. See http://www.iso.ch/meme/JTC1.html for information on JTC 1.

International Standard ISO/IEC 14772 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 24, *Computer graphics and image processing*, in collaboration with The VRML Consortium, Inc. (http://www.vrml.org) and the VRML moderated email list (www-vrml@vrml.org).

ISO/IEC 14772 consists of the following part, under the general title *Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language:*

Part 1: Functional specification and UTF-8 encoding.

Further parts will follow.

Annexes A to C form an integral part of this part of ISO/IEC 14772. Annexes D to F are for information only.

Introduction

Purpose

The Virtual Reality Modeling Language (VRML) is a file format for describing interactive 3D objects and worlds. VRML is designed to be used on the Internet, intranets, and local client systems. VRML is also intended to be a universal interchange format for integrated 3D graphics and multimedia. VRML may be used in a variety of application areas such as engineering and scientific visualization, multimedia presentations, entertainment and educational titles, web pages, and shared virtual worlds.

Design Criteria

VRML has been designed to fulfill the following requirements:

Authorability

> *Enable the development of computer programs capable of creating, editing, and maintaining VRML files, as well as automatic translation programs for converting other commonly used 3D file formats into VRML files.*

Composability

> *Provide the ability to use and combine dynamic 3D objects within a VRML world and thus allow re-usability.*

Extensibility

> *Provide the ability to add new object types not explicitly defined in VRML.*

Be capable of implementation

> *Capable of implementation on a wide range of systems.*

Performance

> *Emphasize scalable, interactive performance on a wide variety of computing platforms.*

Scalability

> *Enable arbitrarily large dynamic 3D worlds.*

Characteristics of VRML

VRML is capable of representing static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies, and images. VRML browsers, as well as authoring tools for the creation of VRML files, are widely available for many different platforms.

VRML supports an extensibility model that allows new dynamic 3D objects to be defined allowing application communities to develop interoperable extensions to the base standard. There are mappings between VRML objects and commonly used 3D application programmer interface (API) features.

ISO/IEC 14772-1:1997

Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language -- Part 1: Functional specification and UTF-8 encoding

1 Scope

ISO/IEC 14772, the Virtual Reality Modeling Language (VRML), defines a file format that integrates 3D graphics and multimedia. Conceptually, each VRML file is a 3D time-based space that contains graphic and aural objects that can be dynamically modified through a variety of mechanisms. This part of ISO/IEC 14772 defines a primary set of objects and mechanisms that encourage composition, encapsulation, and extension.

The semantics of VRML describe an abstract functional behaviour of time-based, interactive 3D, multimedia information. ISO/IEC 14772 does not define physical devices or any other implementation-dependent concepts (e.g., screen resolution and input devices). ISO/IEC 14772 is intended for a wide variety of devices and applications, and provides wide latitude in interpretation and implementation of the functionality. For example, ISO/IEC 14772 does not assume the existence of a mouse or 2D display device.

Each VRML file:

a. implicitly establishes a world coordinate space for all objects defined in the file, as well as all objects included by the file;

b. explicitly defines and composes a set of 3D and multimedia objects;

c. can specify hyperlinks to other files and applications;

d. can define object behaviours.

An important characteristic of VRML files is the ability to compose files together through inclusion and to relate files together through hyperlinking. For example, consider the file *earth.wrl* which specifies a world that contains a sphere representing the earth. This file may also contain references to a variety of other VRML files representing cities on the earth (e.g., file *paris.wrl)*. The enclosing file, *earth.wrl*, defines the coordinate system that all the cities reside in. Each city file defines the world coordinate system that the city resides in but that becomes a local coordinate system when contained by the earth file.

Hierarchical file inclusion enables the creation of arbitrarily large, dynamic worlds. Therefore, VRML ensures that each file is completely described by the objects contained within it.

Another essential characteristic of VRML is that it is intended to be used in a distributed environment such as the World Wide Web. There are various objects and mechanisms built into the language that support multiple distributed files, including:

g. in-lining of other VRML files;

h. hyperlinking to other files;

i. using established Internet and ISO standards for other file formats;

j. defining a compact syntax.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 14772. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 14772 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

Annex E, Bibliography, contains a list of informative documents and technology.

Identifier	Reference
1766	IETF RFC 1766, Tags for the Identification of Languages, Internet standards track protocol. http://ds.internic.net/rfc/rfc1766.txt
CGM	ISO/IEC 8632:1992 (all parts) Information technology -- Computer graphics -- Metafile for the storage and transfer of picture description information. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=8632
ESCR	ISO/IEC DIS 16262 Information technology -- ECMAScript: A general purpose, cross-platform programming language. http://www.ecma.ch http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=16262
HTML	HTML 3.2 Reference Specification. http://www.w3.org/TR/REC-html32.html
I639	ISO 639:1988 Code for the representation of names of languages. http://www.iso.ch/isob/switch-engine-cate.pl?KEYWORDS=10918&searchtype=refnumber, http://www.chemie.fu-berlin.de/diverse/doc/ISO_639.html
I3166	ISO 3166:1997 (all parts) Codes for the representation of names of countries and their subdivisions. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=3166
I8859	ISO/IEC 8859-1:1987 Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=8859

ISOC	ISO/IEC 9899:1990 Programming languages -- C. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=9899
ISOG	ISO/IEC 10641:1993 Information technology -- Computer graphics and image processing -- Conformance testing of implementations of graphics standards. http://www.iso.ch/isob/switch-engine-cate.pl?KEYWORDS=10641&searchtype=refnumber
JAVA	"The Java Language Specification" by James Gosling, Bill Joy and Guy Steele, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63451-1. http://java.sun.com/docs/books/jls/index.html "The Java Virtual Machine Specification" by Tim Lindhold and Frank Yellin, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63452-X. http://java.sun.com/docs/books/vmspec/index.html
JPEG	"JPEG File Interchange Format," JFIF, Version 1.02, 1992. http://www.w3.org/pub/WWW/Graphics/JPEG/jfif.txt ISO/IEC 10918-1:1994 Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines. http://www.iso.ch/isob/switch-engine-cate.pl?KEYWORDS=10918&searchtype=refnumber
MIDI	Complete MIDI 1.0 Detailed Specification, MIDI Manufacturers Association, P.O. Box 3173, La Habra, CA 90632 USA 1996. http://www.midi.org
MPEG	ISO/IEC 11172-1:1993 Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s -- Part 1: Systems. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=11172
PNG	PNG (Portable Network Graphics), Specification Version 1.0, W3C Recommendation, 1 October 1996. http://www.w3.org/pub/WWW/TR/REC-png-multi.html
RURL	IETF RFC 1808 Relative Uniform Resource Locator, Internet standards track protocol. http://ds.internic.net/rfc/rfc1808.txt
URL	IETF RFC 1738 Uniform Resource Locator, Internet standards track protocol. http://ds.internic.net/rfc/rfc1738.txt

UTF8	ISO/IEC 10646-1:1993 Information technology -- Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane, Internet standards track protocol. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=10646, http://ds.internic.net/rfc/rfc2044.txt

3 Definitions

For the purposes of this part of ISO/IEC 14722, the following definitions apply.

3.1 activate

To cause a *sensor node* to generate an "isActive" *event*. The various types of sensor nodes are "activated" by *user* interactions, the passage of *time*, or other events. Only active sensor nodes affect the *user's* experience. A Script *node* is activated when it receives an event. A pointing device such as a *mouse* is activated when one of its buttons is depressed by a user. See 4.12.2, Script execution, for details.

3.2 ancestor

A *node* which is an antecedent of another node in the *transformation hierarchy*.

3.3 author

A person or agent that creates *VRML files*. Authors typically use *generators* to assist them.

3.4 authoring tool

See *generator*.

3.5 avatar

The abstract representation of the *user* in a VRML *world*. The physical dimensions of the avatar are used for collision detection and terrain following. See 6.29, NavigationInfo, for details.

3.6 bearing

A straight line passing through the *pointer* location in the direction of the pointer. If multiple sensors' geometry intersect this line, only the sensor nearest the viewer will be eligible to generate *events* regardless of material and texture properties (e.g., transparency).

3.7 bindable node

A *node* that may have many *instances* in a *scene graph*, but only one instance may be active at any instant of *time*. A node of type Background, Fog, NavigationInfo, or Viewpoint. See 4.6.10, Bindable children nodes, for details.

3.8 browser

A computer program that interprets *VRML files*, presents their content to a *user* on a *display device*, and allows the user to interact with *worlds* defined by VRML files by means of a user interface.

3.9 browser extension

Nodes defined using the prototyping mechanism that are understood only by certain *browsers*. See 4.9.3, Browser extensions, for details.

3.10 built-in node

A *node* of a *type* explicitly defined in this part of ISO/IEC 14772.

3.11 callback

A function defined in a *scripting language* to which *events* are passed. See 4.12.8, EventIn handling, for details.

3.12 candidate

One of potentially several choices. The *user* or the *browser* will select none or one of the choices when all candidates are identified. See 4.6.10, Bindable children nodes, and 6.2, Anchor, for details.

3.13 child

An instance of a *children node*.

3.14 children node

One of a set of *node type*s, instances of which can be collected in a group to share specific properties dependent on the type of the *grouping node*. See 4.6.5, Grouping and children nodes, for a list of allowable children nodes.

3.15 client system

A computer system, attached to a *network*, that relies on another computer (the server) for essential processing functions. Many client systems also function as stand-alone computers.

3.16 collision proxy

A *node* used as a substitute for all of a Collision node's children during collision detection. See 6.8, Collision, for details.

3.17 colour model

Characterization of a colour space in terms of explicit parameters. ISO/IEC 14772 allows colours to be defined only with the RGB colour model. However, colour interpolation is performed in the HSV colour space.

3.18 culling

The process of identifying *objects* or parts of objects which do not need to be processed further by the *browser* in order to produce the desired view of a *world*.

3.19 descendant

A *node* which descends from another node in the *transformation hierarchy*. A *children node*.

3.20 display device

A graphics device on which VRML *worlds* may be rendered.

3.21 drag sensor

A *pointing device sensor* that causes *events* to be generated in response to sensor-dependent pointer motions. For example, the SphereSensor generates spherical rotation events. A *node* of type CylinderSensor, PlaneSensor, or SphereSensor. See 4.6.7, Sensor nodes, and 4.6.7.4, Drag sensors, for details.

3.22 environmental sensor

A sensor *node* that generates *events* based on the location of the viewpoint in the *world* or in relation to *objects* in the world. The TimeSensor node generates events at regular intervals in *time*. A node of type Collision, ProximitySensor, TimeSensor, or VisibilitySensor. See 4.6.7.2, Environmental sensors, for details.

3.23 event

A *message* sent from one *node* to another as defined by a *route*. *Events* signal external stimuli, changes to *field* values, and interactions between nodes. An event consists of a *timestamp* and a field value.

3.24 event cascade

A sequence of *events* initiated by a script or sensor event and propagated from *node* to node along one or more *routes*. All events in an event cascade are considered to have occurred simultaneously. See 4.10.3, Execution model, for details.

3.25 eventIn

A logical receptor attached to a *node* which receives *events*.

3.26 eventOut

A logical output terminal attached to a *node* from which *events* are sent. The eventOut also stores the event most recently sent.

3.27 execution model

The rules governing how *events* are processed by *browsers* and scripts.

3.28 exposed field

A *field* that is capable of receiving *events* via an *eventIn* to change its value(s), and generating events via an *eventOut* when its value(s) change.

3.29 external prototype

A *prototype* defined in an external file and referenced by a *URL*.

3.30 field

A property or attribute of a *node*. Each *node type* has a fixed set of fields. Fields may contain various kinds of data and one or many values. Each field has a default value.

3.31 field name

The identifier of a *field*. Field names are unique within the scope of the *node*.

3.32 file

A collection of related data. A file may be stored on physical media or may exist as a data stream or as data within a computer program.

3.33 frame

A single rendering of a *world* on a *display device* or a single time-step in a simulation.

3.34 generator

A computer program which creates *VRML files*. A generator may be used by a person or operate automatically. Synonymous with *authoring tool*.

3.35 geometric property node

A *node* defining the properties of a specific geometry node. A node of type Color, Coordinate, Normal, or TextureCoordinate. See 4.6.3.2, Geometric property nodes, for details.

3.36 geometric sensor node

A *node* that generates *events* based on *user* actions, such as a *mouse* click or navigating close to a particular *object*. A node of type CylinderSensor, PlaneSensor, ProximitySensor, SphereSensor, TouchSensor, VisibilitySensor, or Collision. See 4.6.7.1, Introduction to sensors, for details.

3.37 geometry node

A *node* containing mathematical descriptions of three-dimensional (3D) points, lines, surfaces, text strings and solids. A node of type Box, Cone, Cylinder, ElevationGrid, Extrusion, IndexedFaceSet, IndexedLineSet, PointSet, Sphere, or Text. See 4.6.3, Shapes and geometry, for details.

3.38 grab

To receive *events* from activated pointing devices (e.g., *mouse* or *wand*). A *pointing device sensor* becomes the exclusive recipient of pointing device events when one or more pointing devices are activated simultaneously.

3.39 gravity

In the context of ISO/IEC 14772, gravity may be simulated by constraining the motion of the viewpoint to the lowest possible path (smallest Y-coordinate in the local coordinate system of the viewpoint) consistent with following the surface of encountered *objects*. See 6.29, NavigationInfo, for details.

3.40 grouping node

One of a set of *node types* which include a list of nodes, referred to as its *children nodes*. These children nodes are collected together to share specific properties dependent on the type of the grouping node. Each grouping node defines a coordinate space for its children relative to its own coordinate space. The children may themselves be instances of grouping nodes, thus forming a *transformation hierarchy*. See 4.6.5, Grouping and children nodes, for details.

3.41 HSV

Hue, Saturation, and Value colour model. See E.[FOLE].

3.42 HTML

HyperText Markup Language. See 2.[HTML].

3.43 hyperlink

A reference to a *URL* that is associated with an Anchor *node*. See 6.2, Anchor, for details.

3.44 ideal VRML implementation

An implementation of VRML that presents all *objects* and simulates movement without approximation. Not realizable in practice.

3.45 IEC

International Electrotechnical Commission. See `http://www.iec.ch`.

3.46 IETF

Internet Engineering Task Force. The organization which develops *Internet* standards. See `http://www.ietf.org/overview.html`.

3.47 image

A two-dimensional (2D) rectangular array of pixel values. Pixel values may have from one to four components. See 5.5, SFImage, for details.

3.48 in-lining

The mechanism by which one *VRML file* is hierarchically included in another.

3.49 Internet

The world-wide named *network* of computers which communicate with each other using a common set of communication protocols known as TCP/IP. See *IETF*. The *World Wide Web* is implemented on the Internet.

3.50 instance

A reference to a previously defined and named *node*. Nodes are named by means of the DEF syntax and reference by USE syntax (see 4.6.2, DEF/USE semantics). Instances of nodes may be used in any context in which the defining node may be used.

3.51 interpolator node

A *node* that defines a piece-wise linear interpolation. A node of type ColorInterpolator, CoordinateInterpolator, NormalInterpolator, OrientationInterpolator, PositionInterpolator, or ScalarInterpolator. See 4.6.8, Interpolator nodes, for details.

3.52 intranet

A private *network* that uses the same protocols and standards as the *Internet*.

3.53 ISO

International Organization for Standardization. See `http://www.iso.ch/infoe/intro.html`.

3.54 JPEG

Joint Photographic Experts Group. See 2.[JPEG].

3.55 JTC 1

ISO/IEC Joint Technical Committee 1. See `http://www.iso.ch/meme/JTC1.html`.

3.56 level of detail

The amount of detail or complexity which is displayed at any particular *time* for any particular *object*. The level of detail for an object is controllable as a function of the distance of the object from the viewer. See 6.26, LOD, for details. (Abbreviated LOD)

3.57 line terminator

A linefeed character (0x0A) or a carriage return character (0x0D).

3.58 loop

A sequence of *events* which would result in a specific *eventOut* sending more than one event with the same *timestamp*.

3.59 message

A data string sent between *nodes* upon the occurrence of an *event*. See 4.10, Event processing, for details.

3.60 MIDI

Musical Instrument Digital Interface. A standard for digital music representation. See 2.[MIDI].

3.61 MIME

Multipurpose Internet Mail Extension. Used to specify filetyping rules for *Internet* applications, including *browsers*. See 4.5.1, File extension and MIME types, for details. See also E.[MIME].

3.62 mouse

A pointing device that moves in two dimensions and that enables a *user* to move a cursor on a *display device* in order to point at displayed *objects*. One or more push buttons on the mouse allow the user to indicate to the computer program that some action is to be taken.

3.63 MPEG

Moving Picture Experts Group. See `http://drogo.cselt.stet.it/mpeg/`.

3.64 multimedia

An integrated presentation, typically on a computer, of content of various types, such as computer graphics, audio, and video.

3.65 network

Set of interconnected computers.

3.66 node

The fundamental component of a *scene graph* in ISO/IEC 14772. Nodes are abstractions of various real-world objects and concepts. Examples include spheres, lights, and material descriptions. Nodes contain *fields* and *events*. *Messages* may be sent between nodes along *routes*.

3.67 node type

A characteristic of each *node* that describes, in general, its particular semantics. For example, Box, Group, Sound, and SpotLight are node types. See 4.6, Node semantics, and 6, Node reference, for details.

3.68 now

The present *time* as perceived by the *user*.

3.69 object

A collection of data and procedures, packaged according to the rules and syntax defined in ISO/IEC 14772. "Object" is usually synonymous with *node*.

3.70 object space

The coordinate system in which an *object* is defined.

3.71 panorama

A background texture that is placed behind all geometry in the scene and in front of the ground and sky. See 6.5, Background, for details.

3.72 parent

A *node* which is an instance of a *grouping node*.

3.73 PNG

Portable Network Graphics. A specification for representing two-dimensional images in *files*. See 2.[PNG].

3.74 pointer

A location and direction in the *virtual world* defined by the *pointing device* which the *user* is currently using to interact with the virtual world.

3.75 pointing device

A hardware device connected to the *user's* computer by which the user directly controls the location and direction of the *pointer*. Pointing devices may be either two-dimensional or three-dimensional and may have one or more control buttons. See 4.6.7.5, Activating and manipulating sensors, for details.

3.76 pointing device sensor

A sensor *node* that generates *events* based on *user* actions, such as *pointing device* motions or button activations. A node of type Anchor, CylinderSensor, PlaneSensor, SphereSensor, or TouchSensor. See 4.6.7.3, Pointing device sensors, for details.

3.77 polyline

A sequence of straight line segments where the end point of the first segment is coincident with the start point of the second segment, the endpoint of the second segment is coincident with the start point of the third segment, and so on. A piecewise linear curve.

3.78 profile

A named collection of criteria for functionality and conformance that defines an implementable subset of a standard.

3.79 prototype

The definition of a new *node type* in terms of the *nodes* defined in this part of ISO/IEC 14772. See 4.8, Prototype semantics, for details.

3.80 prototyping

The mechanism for extending the set of *node types* from within a *VRML file*.

3.81 public interface

The formal definition of a *node type* in this part of ISO/IEC 14772.

3.82 RGB

The colour model used within ISO/IEC 14772 for the specification of colours. Each colour is represented as a combination of the three primary colours red, green, and blue. See E.[FOLE].

3.83 route

The connection between a *node* generating an *event* and a node receiving the event. See 4.3.9, Route statement syntax, and 4.10.2, Route semantics, for details.

3.84 route graph

The set of connections between *eventOuts* and *eventIns* formed by ROUTE statements or addRoute method invocations.

3.85 run-time name scope

The extent to which a name defined within a VRML file applies and is visible. Several different run-time name scopes are recognized and are defined in 4.4.6, Run-time name scope.

3.86 RURL

Relative Uniform Resource Locator. See 2.[RURL].

3.87 scene graph

An ordered collection of *grouping nodes* and other nodes. Grouping nodes, (such as LOD, Switch, and Transform nodes) may have *children nodes*. See 4.2.3, Scene graph, and 4.4.2, Scene graph hierarchy, for details.

3.88 script

A set of procedural functions normally executed as part of an *event cascade* (see 6.40, Script). A script function may also be executed asynchronously (see 4.12.6, Asynchronous scripts).

3.89 scripting

The process of creating or referring to a script.

3.90 scripting language

A system of syntactical and semantic constructs used to define and automate procedures and processes on a computer. Typically, scripting languages are interpreted and executed sequentially on a statement-by-statement basis whereas programming languages are generally compiled prior to execution.

3.91 sensor node

A *node* that enables the *user* to interact with the *world* in the scene graph hierarchy. Sensor nodes respond to user interaction with geometric *objects* in the world, the movement of the user through the world, or the passage of *time*. See 4.6.7, Sensor nodes, for details.

3.92 separator character

A *UTF-8* character used to separate syntactical entities in a *VRML file*. Specifically, commas, spaces, tabs, linefeeds, and carriage-returns are separator characters wherever they appear outside of string *fields*. See 4.3.1, Clear text (UTF-8) encoding, for details.

3.93 sibling

A *node* which shares a *parent* with other nodes.

3.94 simulation tick

The smallest time unit capable of being identified in a digital simulation of analog time. *Time* in the context of ISO/IEC 14772 is conceptually analog but is realized by an implementation as a digital simulation of abstract analog time. See 4.11, Time, for details.

3.95 special group node

A *grouping node* that exhibits special behaviour. Examples of such special behaviour include selecting one of many *children nodes* to be rendered based on a dynamically changing parameter value and dynamically loading children nodes from an external file. A node of type Inline, LOD (level of detail), or Switch. See 4.6.5, Grouping and children nodes, for details.

3.96 texture

An *image* used in a *texture map* to create visual appearance effects when applied to *geometry nodes*.

3.97 texture coordinates

The set of two-dimensional coordinates used by some vertex-based *geometry nodes* (*e.g.*, IndexedFaceSet and ElevationGrid) and specified in the TextureCoordinate node to map textures to the vertices of those nodes. Texture coordinates range from 0 to 1 across each axis of the texture image. See 4.6.11, Texture maps, and 6.48, TextureCoordinate, for details.

3.98 texture map

A *texture* plus the general parameters necessary for mapping the texture to geometry.

3.99 time

A monotonically increasing value generated by a node. Time (0.0) starts at 00:00:00 GMT January 1, 1970. See 4.11, Time, for details.

3.100 timestamp

The part of a *message* that describes the *time* the *event* occurred and that caused the message to be sent. See 4.11, Time, for details.

3.101 transformation hierarchy

The subset of the *scene graph* consisting of *nodes* that have well-defined coordinate systems. The transformation hierarchy excludes nodes that are not *descendants* of the scene graph root nodes and nodes in SFNode or MFNode fields of Script nodes.

3.102 transparency chunk

A section of a PNG file containing transparency information (derived from 2.[PNG]).

3.103 traverse

To process the *nodes* in a *scene graph* in the correct order.

3.104 UCS

Universal multiple-octet coded Character Set. See 2.[UTF8].

3.105 URL

Uniform Resource Locator. See 2.[URL].

3.106 URN

Universal Resource Name. See E.[URN].

3.107 UTF-8

The character set used to encode *VRML files*. The 8-bit UCS Transformation Format. See 2.[UTF8].

3.108 user

A person or agent who uses and interacts with *VRML files* by means of a *browser*.

3.109 viewer

A location, direction, and viewing angle in a *virtual world* that determines the portion of the virtual world presented by the *browser* to the *user*.

3.110 virtual world

See *world*.

3.111 VRML browser

See *browser*.

3.112 VRML document server

A computer program that locates and transmits *VRML files* and supporting files in response to requests from *browsers*.

3.113 VRML file

A set of VRML nodes and statements as defined in this part of ISO/IEC 14772. This set of VRML nodes and statements may be in the form of a file, a data stream, or an in-line sequence of VRML information as defined by a particular VRML encoding.

3.114 wand

A pointing device that moves in three dimensions and that enables a *user* to indicate a position in the three-dimensional coordinate system of a world in order to point at displayed *objects*. One or more push buttons on the wand allow the user to indicate to the computer program that some action is to be taken.

3.115 white space

One or more consecutive occurrences of a *separator character*. See 4.3.1, Clear text (UTF-8) encoding, for details.

3.116 world

A collection of one or more *VRML files* and other multimedia content that, when interpreted by a *VRML browser*, presents an interactive experience to the *user* consistent with the *author's* intent.

3.117 world coordinate space

The coordinate system in which each VRML *world* is defined. The world coordinate space is an orthogonal right-handed Cartesian coordinate system. The units of length are metres.

3.118 World Wide Web

The collection of documents, data, and content typically encoded in HTML pages and accessible via the *Internet* using the HTTP protocol.

3.119 XY plane

The plane perpendicular to the Z-axis that passes through the point Z = 0.0.

3.120 YZ plane

The plane perpendicular to the X-axis that passes through the point X = 0.0.

3.121 ZX plane

The plane perpendicular to the Y-axis that passes through the point Y = 0.0.

4 Concepts

4.1 Introduction and table of contents

4.1.1 Introduction

This clause describes key concepts in ISO/IEC 14772. 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 scripts into a *VRML file*, and various general topics on nodes.

4.1.2 Table of contents

See Table 4.1 for the table of contents for this clause.

Table 4.1 -- Table of contents, Concepts

4.1.3 Conventions used

The following conventions are used throughout this part of ISO/IEC 14772:

Italics are used for event and field names, and are also used when new terms are introduced and equation variables are referenced.

A `fixed-space` font is used for URL addresses and source code examples. ISO/IEC 14772 UTF-8 encoding examples appear in **`bold, fixed-space`** font.

Node type names are appropriately capitalized (e.g., "The Billboard node is a grouping node..."). However, the concept of the node is often referred to in lower case in order to refer to the semantics of the node, not the node itself (e.g., "To rotate the billboard...").

The form "0xhh" expresses a byte as a hexadecimal number representing the bit configuration for that byte.

Throughout this part of ISO/IEC 14772, references are denoted using the "x.[ABCD]" notation, where "x" denotes which clause or annex the reference is described in and "[ABCD]" is an abbreviation of the reference title. For example, 2.[ABCD] refers to a reference described in clause 2 and E.[ABCD] refers to a reference described in annex E.

4.2 Overview

4.2.1 The structure of a VRML file

A *VRML file* consists of the following major functional components: the header, the *scene graph*, the prototypes, and *event routing*. The contents of this file are processed for presentation and interaction by a program known as a *browser*.

4.2.2 Header

For easy identification of VRML files, every VRML file shall begin with:

```
#VRML V2.0 <encoding type> [optional comment] <line terminator>
```

The header is a single line of UTF-8 text identifying the file as a VRML file and identifying the encoding type of the file. It may also contain additional semantic information. There shall be exactly one space separating "**`#VRML`**" from "**`V2.0`**" and "**`V2.0`**" from "`<encoding type>`". Also, the "`<encoding type>`" shall be followed by a linefeed (0x0a) or carriage-return (0x0d) character, or by one or more space (0x20) or tab (0x09) characters followed by any other characters, which are treated as a comment, and terminated by a linefeed or carriage-return character.

The `<encoding type>` is either "**`utf8`**" or any other authorized values defined in other parts of ISO/IEC 14772. The identifier "**`utf8"`** indicates a clear text encoding that allows for international characters to be displayed in ISO/IEC 14772 using the UTF-8 encoding defined in ISO/IEC 10646-1 (otherwise known as Unicode); see 2.[UTF8]. The usage of UTF-8 is detailed in 6.47, Text, node. The header for a UTF-8 encoded VRML file is

```
#VRML V2.0 utf8 [optional comment] <line terminator>
```

Any characters after the `<encoding type>` on the first line may be ignored by a browser. The header line ends at the occurrence of a `<line terminator>`. A `<line terminator>` is a linefeed character (0x0a) or a carriage-return character (0x0d) .

4.2.3 Scene graph

The scene graph contains nodes which describe objects and their properties. It contains hierarchically grouped geometry to provide an audio-visual representation of objects, as well as nodes that participate in the event generation and routing mechanism.

4.2.4 Prototypes

Prototypes allow the set of VRML node types to be extended by the user. Prototype definitions can be included in the file in which they are used or defined externally. Prototypes may be defined in terms of other VRML nodes or may be defined using a browser-specific extension mechanism. While ISO/IEC 14772 has a standard format for identifying such extensions, their implementation is browser-dependent.

4.2.5 Event routing

Some VRML nodes generate events in response to environmental changes or user interaction. Event routing gives authors a mechanism, separate from the scene graph hierarchy, through which these events can be propagated to effect changes in other nodes. Once generated, events are sent to their routed destinations in time order and processed by the receiving node. This processing can change the state of the node, generate additional events, or change the structure of the scene graph.

Script nodes allow arbitrary, author-defined event processing. An event received by a Script node causes the execution of a function within a script which has the ability to send events through the normal event routing mechanism, or bypass this mechanism and send events directly to any node to which the Script node has a reference. Scripts can also dynamically add or delete routes and thereby changing the event-routing topology.

The ideal event model processes all events instantaneously in the order that they are generated. A timestamp serves two purposes. First, it is a conceptual device used to describe the chronological flow of the event mechanism. It ensures that deterministic results can be achieved by real-world implementations that address processing delays and asynchronous interaction with external devices. Second, timestamps are also made available to Script nodes to allow events to be processed based on the order of user actions or the elapsed time between events.

4.2.6 Generating VRML files

A *generator* is a human or computerized creator of VRML files. It is the responsibility of the generator to ensure the correctness of the VRML file and the availability of supporting assets (e.g., images, audio clips, other VRML files) referenced therein.

4.2.7 Presentation and interaction

The interpretation, execution, and presentation of VRML files will typically be undertaken by a mechanism known as a browser, which displays the shapes and sounds in the scene graph. This presentation is known as a *virtual world* and is navigated in the browser by a human or mechanical entity, known as a *user*. The world is displayed as if experienced from a particular location; that position and orientation in the world is known as the *viewer*. The browser provides navigation paradigms (such as walking or flying) that enable the user to move the viewer through the virtual world.

In addition to navigation, the browser provides a mechanism allowing the user to interact with the world through sensor nodes in the scene graph hierarchy. Sensors respond to user interaction with geometric objects in the world, the movement of the user through the world, or the passage of time.

The visual presentation of geometric objects in a VRML world follows a conceptual model designed to resemble the physical characteristics of light. The VRML lighting model describes how appearance properties and lights in the world are combined to produce displayed colours (see 4.14, Lighting Model, for details).

Figure 4.1 illustrates a conceptual model of a VRML browser. The browser is portrayed as a presentation application that accepts user input in the forms of file selection (explicit and implicit) and user interface gestures (e.g., manipulation and navigation using an input device). The three main components of the browser are: Parser, Scene Graph, and Audio/Visual Presentation. The Parser component reads the VRML file and creates the Scene Graph. The Scene Graph component consists of the Transformation Hierarchy (the nodes) and the Route Graph. The Scene Graph also includes the Execution Engine that processes events, reads and edits the Route Graph, and makes changes to the Transform Hierarchy (nodes). User input generally affects sensors and navigation, and thus is wired to the Route Graph component (sensors) and the Audio/Visual Presentation component (navigation). The Audio/Visual Presentation component performs the graphics and audio rendering of the Transform Hierarchy that feeds back to the user.

4.2.8 Profiles

ISO/IEC 14772 supports the concept of profiles. A profile is a named collection of functionality and requirements which shall be supported in order for an implementation to conform to that profile. Only one profile is defined in this part of ISO/IEC 14772. The functionality and minimum support requirements described in ISO/IEC 14772-1 form the *Base* profile. Additional profiles may be defined in other parts of ISO/IEC 14772. Such profiles shall incorporate the entirety of the Base profile.

4.3 UTF-8 file syntax

4.3.1 Clear text (UTF-8) encoding

This section describes the syntax of UTF-8-encoded, human-readable VRML files. A more formal description of the syntax may be found in annex A, Grammar definition. The semantics of VRML in terms of the UTF-8 encoding are presented in this part of ISO/IEC 14772. Other encodings may be defined in other parts of ISO/IEC 14772. Such encodings shall describe how to map the UTF-8 descriptions to and from the corresponding encoding elements.

For the UTF-8 encoding, the # character begins a comment. The first line of the file, the header, also starts with a "#" character. Otherwise, all characters following a "#", until the next line terminator, are ignored. The only exception is within double-quoted SFString and MFString fields where the "#" character is defined to be part of the string.

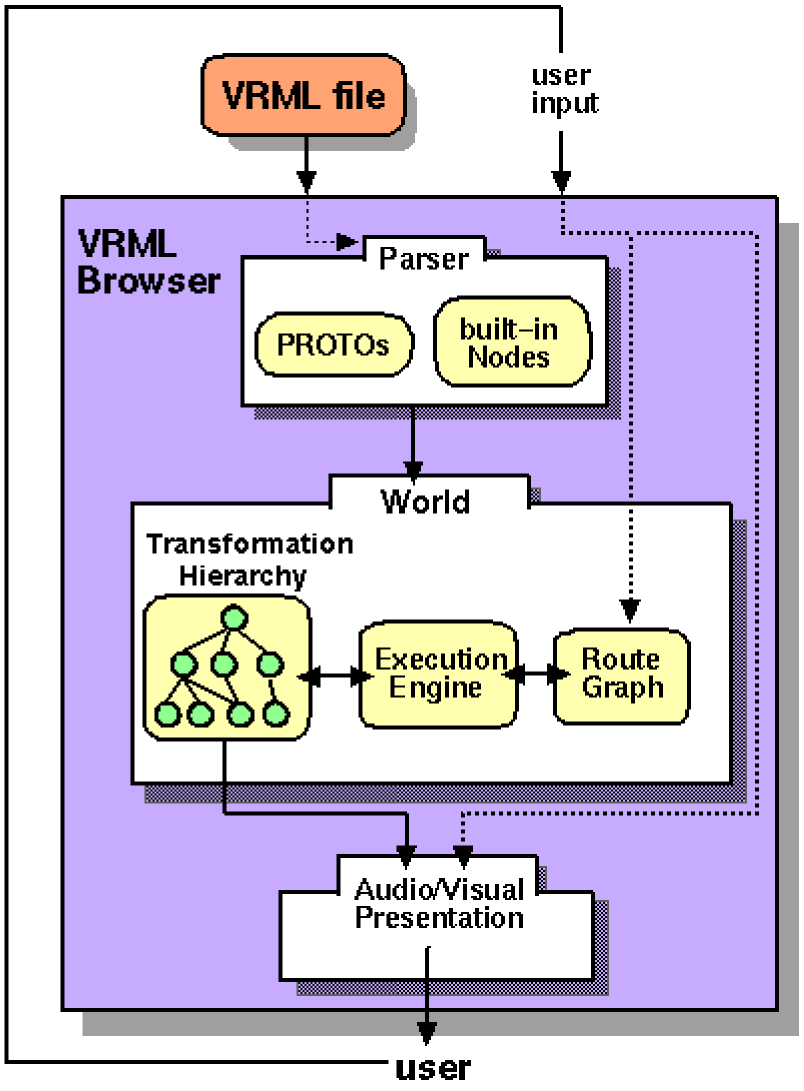


Figure 4.1 -- Conceptual model of a VRML browser

Commas, spaces, tabs, linefeeds, and carriage-returns are separator characters wherever they appear outside of string fields. Separator characters and comments are collectively termed *whitespace*.

A VRML document server may strip comments and extra separators including the comment portion of the header line from a VRML file before transmitting it. WorldInfo nodes should be used for persistent information such as copyrights or author information.

Field, event, PROTO, EXTERNPROTO, and node names shall not contain control characters (0x0-0x1f, 0x7f), space (0x20), double or single quotes (0x22: ", 0x27: '), sharp (0x23: #), comma (0x2c: ,), period (0x2e: .), brackets (0x5b, 0x5d: []), backslash (0x5c: \) or braces (0x7b, 0x7d: {}). Further, their first character shall not be a digit (0x30-0x39), plus (0x2b: +), or minus (0x2d: -) character. Otherwise, names may contain any ISO 10646 character 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 field, event, PROTO, EXTERNPROTO, or node names:

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

4.3.2 Statements

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

a. Any number of PROTO or EXTERNPROTO statements (see 4.8, Prototype semantics);

b. Any number of root node statements (see 4.4.1, Root nodes);

c. Any number of USE statements (see 4.6.2, DEF/USE semantics);

d. Any number of ROUTE statements (see 4.10.2, Route semantics).

4.3.3 Node statement syntax

A node statement consists of an optional name for the node followed by the node's type and then the body of the node. A node is given a name using the keyword DEF followed by the name of the node. The node's body is enclosed in matching braces ("**{ }**"). Whitespace shall separate the DEF, name of the node, and node type, but is not required before or after the curly braces that enclose the node's body. See A.3, Nodes, for details on node grammar rules.

```
[DEF <name>] <nodeType> { <body> }
```

A node's body consists of any number of field statements, IS statements, ROUTE statements, PROTO statements or EXTERNPROTO statements, in any order.

See 4.6.2, DEF/USE, sematnics for more details on node naming. See 4.3.4, Field statement syntax, for a description of field statement syntax and 4.7, Field, eventIn, and eventOut semantics, for a description of field statement semantics. See 4.6, Node semantics, for a description of node statement semantics.

4.3.4 Field statement syntax

A field statement consists of the name of the field followed by the field's value(s). The following illustrates the syntax for a single-valued field:

```
<fieldName> <fieldValue>
```

The following illustrates the syntax for a multiple-valued field:

```
<fieldName> [ <fieldValues> ]
```

See A.4, Fields, for details on field statement grammar rules.

Each node type defines the names and types of the fields that each node of that type contains. The same field name may be used by multiple node types. See 5, Field and event reference, for the definition and syntax of specific field types.

See 4.7, Field, eventIn, and eventOut semantics, for a description of field statement semantics.

4.3.5 PROTO statement syntax

A PROTO statement consists of the PROTO keyword, followed in order by the prototype name, prototype interface declaration, and prototype definition:

```
PROTO <name> [ <declaration> ] { <definition> }
```

See A.2, General, for details on prototype statement grammar rules.

A prototype interface declaration consists of eventIn, eventOut, field, and exposedField declarations (see 4.7, Field, eventIn, and eventOut semantics) enclosed in square brackets. Whitespace is not required before or after the brackets.

EventIn declarations consist of the keyword "eventIn" followed by an event type and a name:

```
eventIn <eventType> <name>
```

EventOut declarations consist of the keyword "eventOut" followed by an event type and a name:

```
eventOut <eventType> <name>
```

Field and exposedField declarations consist of either the keyword "field" or "exposedField" followed by a field type, a name, and an initial field value of the given field type.

```
field <fieldType> <name> <initial field value>

exposedField <fieldType> <name> <initial field value>
```

Field, eventIn, eventOut, and exposedField names shall be unique in each PROTO statement, but are not required to be unique between different PROTO statements. If a PROTO statement contains an exposedField with a given name (e.g., *zzz*), it shall not contain eventIns or eventOuts with the prefix *set_* or the suffix *_changed* and the given name (e.g., *set_zzz* or *zzz_changed*).

A prototype definition consists of at least one node statement and any number of ROUTE statements, PROTO statements, and EXTERNPROTO statements in any order.

See 4.8, Prototype semantics, for a description of prototype semantics.

4.3.6 IS statement syntax

The body of a node statement that is inside a prototype definition may contain IS statements. An IS statement consists of the name of a field, exposedField, eventIn or eventOut from the node's public interface followed by the keyword IS followed by the name of a field, exposedField, eventIn or eventOut from the prototype's interface declaration:

```
<field/eventName> IS <field/eventName>
```

See A.3, Nodes, for details on prototype node body grammar rules. See 4.8, Prototype semantics, for a description of IS statement semantics.

4.3.7 EXTERNPROTO statement syntax

An EXTERNPROTO statement consists of the EXTERNPROTO keyword followed in order by the prototype's name, its interface declaration, and a list (possibly empty) of double-quoted strings enclosed in square brackets. If there is only one member of the list, the brackets are optional.

```
EXTERNPROTO <name> [ <external declaration> ] URL or [ URLs ]
```

See A.2, General, for details on external prototype statement grammar rules.

An EXTERNPROTO interface declaration is the same as a PROTO interface declaration, with the exception that field and exposedField initial values are not specified and the prototype definition is specified in a separate VRML file to which the URL(s) refer.

4.3.8 USE statement syntax

A USE statement consists of the USE keyword followed by a node name:

```
USE <name>
```

See A.2, General, for details on USE statement grammar rules.

4.3.9 ROUTE statement syntax

A ROUTE statement consists of the ROUTE keyword followed in order by a node name, a period character, a field name, the TO keyword, a node name, a period character, and a field name. Whitespace is allowed but not required before or after the period characters:

```
ROUTE <name>.<field/eventName> TO <name>.<field/eventName>
```

See A.2, General, for details on ROUTE statement grammar rules.

4.4 Scene graph structure

4.4.1 Root nodes

A VRML file contains zero or more root nodes. The root nodes for a VRML file are those nodes defined by the node statements or USE statements that are not contained in other node or PROTO statements. Root nodes shall be children nodes (see 4.6.5, Grouping and children nodes).

4.4.2 Scene graph hierarchy

A VRML file contains a directed acyclic graph. Node statements can contain SFNode or MFNode field statements that, in turn, contain node (or USE) statements. This hierarchy of nodes is called the *scene graph*. Each arc in the graph from A to B means that node A has an SFNode or MFNode field whose value directly contains node B. See E.[FOLE] for details on hierarchical scene graphs.

4.4.3 Descendant and ancestor nodes

The *descendants* of a node are all of the nodes in its SFNode or MFNode fields, as well as all of those nodes' descendants. The *ancestors* of a node are all of the nodes that have the node as a descendant.

4.4.4 Transformation hierarchy

The transformation hierarchy includes all of the root nodes and root node descendants that are considered to have one or more particular locations in the virtual world. VRML includes the notion of *local coordinate systems*, defined in terms of transformations from ancestor coordinate systems (using Transform or Billboard nodes). The coordinate system in which the root nodes are displayed is called the *world coordinate system*.

A VRML browser's task is to present a VRML file to the user; it does this by presenting the transformation hierarchy to the user. The transformation hierarchy describes the directly perceptible parts of the virtual world.

The following node types are in the scene graph but not affected by the transformation hierarchy: ColorInterpolator, CoordinateInterpolator, NavigationInfo, NormalInterpolator, OrientationInterpolator, PositionInterpolator, Script, ScalarInterpolator, TimeSensor, and WorldInfo. Of these, only Script nodes may have descendants. A descendant of a Script node is not part of the transformation hierarchy unless it is also the descendant of another node that is part of the transformation hierarchy or is a root node.

Nodes that are descendants of LOD or Switch nodes are affected by the transformation hierarchy, even if the settings of a Switch node's *whichChoice* field or the position of the viewer with respect to a LOD node makes them imperceptible.

The transformation hierarchy shall be a directed acyclic graph; results are undefined if a node in the transformation hierarchy is its own ancestor.

4.4.5 Standard units and coordinate system

ISO/IEC 14772 defines the unit of measure of the world coordinate system to be metres. All other coordinate systems are built from transformations based from the world coordinate system. Table 4.2 lists standard units for ISO/IEC 14772.

Table 4.2 -- Standard units

Category	Unit
Linear distance	Metres
Angles	Radians
Time	Seconds
Colour space	RGB ([0.,1.], [0.,1.], [0., 1.])

ISO/IEC 14772 uses a Cartesian, right-handed, three-dimensional coordinate system. By default, the viewer is on the Z-axis looking down the -Z-axis toward the origin with +X to the right and +Y straight up. A modelling transformation (see 6.52, Transform, and 6.6, Billboard) or viewing transformation (see 6.53, Viewpoint) can be used to alter this default projection.

4.4.6 Run-time name scope

Each VRML file defines a run-time name scope that contains all of the root nodes of the file and all of the descendent nodes of the root nodes, with the exception of:

a. descendent nodes that are inside Inline nodes;

b. descendent nodes that are inside a prototype instance and are not part of the prototype's interface (i.e., are not in an SF/MFNode field or eventOut of the prototype).

Each Inline node and prototype instance also defines a run-time name scope, consisting of all of the root nodes of the file referred to by the Inline node or all of the root nodes of the prototype definition, restricted as above.

Nodes created dynamically (using a Script node invoking the Browser.createVrml methods) are not part of any name scope, until they are added to the scene graph, at which point they become part of the same name scope of their parent node(s). A node may be part of more than one run-time name scope. A node shall be removed from a name scope when it is removed from the scene graph.

4.5 VRML and the World Wide Web

4.5.1 File extension and MIME types

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

The official MIME type for VRML files is defined as:

```
model/vrml
```

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

For compatibility with earlier versions of VRML, the following MIME type shall also be supported:

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

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

See E.[MIME] for details.

4.5.2 URLs

A *URL* (Uniform Resource Locator), described in 2.[URL], specifies a file located on a particular server and accessed through a specified protocol (e.g., http). In ISO/IEC 14772, the upper-case term URL refers to a Uniform Resource Locator, while the italicized lower-case version *url* refers to a field which may contain URLs or in-line encoded data.

All *url* fields are of type MFString. The strings in these fields indicate multiple locations to search for data in decreasing order of preference. If the browser cannot locate or interpret the data specified by the first location, it shall try the second and subsequent locations in order until a URL containing interpretable data is encountered. If no interpretable URL's are located, the node type defines the resultant default behaviour. The *url* field entries are delimited by double quotation marks " ". Due to 4.5.4, Scripting language protocols, *url* fields use a superset of the standard URL syntax defined in 2.[URL]. Details on the string field are located in 5.9, SFString and MFString.

More general information on URLs is described in 2.[URL].

4.5.3 Relative URLs

Relative URLs are handled as described in 2.[RURL]. The base document for EXTERNPROTO statements or nodes that contain URL fields is:

a. The VRML file in which the prototype is instantiated, if the statement is part of a prototype definition.

b. The file containing the script code, if the statement is part of a string passed to the createVrmlFromURL() or createVrmlFromString() browser calls in a Script node.

c. Otherwise, the VRML file from which the statement is read, in which case the RURL information provides the data itself.

4.5.4 Scripting language protocols

The Script node's *url* field may also support custom protocols for the various scripting languages. For example, a script *url* prefixed with *javascript:* shall contain ECMAScript source, with line terminators allowed in the string. The details of each language protocol are defined in the annex for each language. Browsers are not required to support any specific scripting language. However, browsers shall adhere to the protocol defined in the corresponding annex of ISO/IEC 14772 for any scripting language which is supported. The following example illustrates the use of mixing custom protocols and standard protocols in a single *url* field (order of precedence determines priority):

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

In the example above, the "..." represents in-line ECMAScript source code.

4.6 Node semantics

4.6.1 Introduction

Each node has the following characteristics:

a. **A type name.** Examples include Box, Color, Group, Sphere, Sound, or SpotLight.

b. **Zero or more fields that define how each node differs from other nodes of the same type.** Field values are stored in the VRML file along with the nodes, and encode the state of the virtual world.

c. **A set of events that it can receive and send.** Each node may receive zero or more different kinds of events which will result in some change to the node's state. Each node may also generate zero or more different kinds of events to report changes in the node's state.

d. **An implementation.** The implementation of each node defines how it reacts to events it can receive, when it generates events, and its visual or auditory appearance in the virtual world (if any). The VRML standard defines the semantics of built-in nodes (i.e., nodes with implementations that are provided by the VRML browser). The PROTO statement may be used to define new types of nodes, with behaviours defined in terms of the behaviours of other nodes.

e. **A name.** Nodes can be named. This is used by other statements to reference a specific instantiation of a node.

4.6.2 DEF/USE semantics

A node given a name using the DEF keyword may be referenced by name later in the same file with USE or ROUTE statements. The USE statement does not create a copy of the node. Instead, the same node is inserted into the scene graph a second time, resulting in the node having multiple parents. Using an instance of a node multiple times is called *instantiation*.

Node names are limited in scope to a single VRML file, prototype definition, or string submitted to either the CreateVrmlFromString browser extension or a construction mechanism for SFNodes within a script. Given a node named "NewNode" (i.e., `DEF NewNode`), any "`USE NewNode`" statements in SFNode or MFNode fields inside NewNode's scope refer to NewNode (see 4.4.4, Transformation hierarchy, for restrictions on self-referential nodes).

If multiple nodes are given the same name, each USE statement refers to the closest node with the given name preceding it in either the VRML file or prototype definition.

4.6.3 Shapes and geometry

4.6.3.1 Introduction

The Shape node associates a geometry node with nodes that define that geometry's appearance. Shape nodes shall be part of the transformation hierarchy to have any visible result, and the transformation hierarchy shall contain Shape nodes for any geometry to be visible (the only nodes that render visible results are Shape nodes and the Background node). A Shape node contains exactly one geometry node in its *geometry* field. The following node types are *geometry* nodes:

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

4.6.3.2 Geometric property nodes

Several geometry nodes contain Coordinate, Color, Normal, and TextureCoordinate as geometric property nodes. The geometric property nodes are defined as individual nodes so that instancing and sharing is possible between different geometry nodes.

4.6.3.3 Appearance nodes

Shape nodes may specify an Appearance node that describes the appearance properties (material and texture) to be applied to the Shape's geometry. Nodes of the following type may be specified in the *material* field of the Appearance node:

- Material

Nodes of the following types may be specified by the *texture* field of the Appearance node:

- ImageTexture
- PixelTexture
- MovieTexture

Nodes of the following types may be specified in the *textureTransform* field of the Appearance node:

- TextureTransform

The interaction between such appearance nodes and the Color node is described in 4.14, Lighting Model.

4.6.3.4 Shape hint fields

The Extrusion and IndexedFaceSet nodes each have three SFBool fields that provide hints about the geometry. These hints specify the vertex ordering, if the shape is solid, and if the shape contains convex faces. These fields are *ccw*, *solid*, and *convex*, respectively. The ElevationGrid node has the *ccw* and *solid* fields.

The *ccw* field defines the ordering of the vertex coordinates of the geometry with respect to user-given or automatically generated normal vectors used in the lighting model equations. If *ccw* is TRUE, the normals shall follow the right hand rule; the orientation of each normal with respect to the vertices (taken in order) shall be such that the vertices appear to be oriented in a counterclockwise order when the vertices are viewed (in the local coordinate system of the Shape) from the opposite direction as the normal. If *ccw* is FALSE, the normals shall be oriented in the opposite direction. If normals are not generated but are supplied using a Normal node, and the orientation of the normals does not match the setting of the *ccw* field, results are undefined.

The *solid* field determines whether one or both sides of each polygon shall be displayed. If *solid* is FALSE, each polygon shall be visible regardless of the viewing direction (i.e., no backface culling shall be done, and two-sided lighting shall be performed to illuminate both sides of lit surfaces). If *solid* is TRUE, the visibility of each polygon shall be determined as follows: Let ***V*** be the position of the viewer in the local coordinate system of the geometry. Let ***N*** be the geometric normal vector of the polygon, and let ***P*** be any point (besides the local origin) in the plane defined by the polygon's vertices. Then if (***V*** dot ***N***) **-** (***N*** dot ***P***) is greater than zero, the polygon shall be visible; if it is less than or equal to zero, the polygon shall be invisible (backface culled).

The *convex* field indicates whether all polygons in the shape are convex (TRUE). A polygon is convex if it is planar, does not intersect itself, and all of the interior angles at its vertices are less than 180 degrees. Non-planar and self-intersecting polygons may produce undefined results even if the *convex* field is FALSE.

4.6.3.5 Crease angle field

The *creaseAngle* field, used by the ElevationGrid, Extrusion, and IndexedFaceSet nodes, affects how default normals are generated. If the angle between the geometric normals of two adjacent faces is less than the crease angle, normals shall be calculated so that the faces are smooth-shaded across the edge; otherwise, normals shall be calculated so that a lighting discontinuity across the edge is produced. For example, a crease angle of 0.5 radians means that an edge between two adjacent polygonal faces will be smooth shaded if the geometric normals of the two faces form an angle that is less than 0.5 radians. Otherwise, the faces will appear faceted. Crease angles shall be greater than or equal to 0.0.

4.6.4 Bounding boxes

Several of the nodes include a bounding box specification comprised of two fields, *bboxSize* and *bboxCenter*. A bounding box is a rectangular parallelepiped of dimension *bboxSize* centred on the location *bboxCenter* in the local coordinate system. This is typically used by grouping nodes to provide a hint to the browser on the group's approximate size for culling optimizations. The default size for bounding boxes (-1, -1, -1) indicates that the user did not specify the bounding box and the effect shall be as if the bounding box were infinitely large. A *bboxSize* value of (0, 0, 0) is valid and represents a point in space (i.e., an infinitely small box). Specified *bboxSize* field values shall be >= 0.0 or equal to (-1, -1, -1). The *bboxCenter* fields specify a position offset from the local coordinate system.

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. The bounding box shall be large enough at all times to enclose the union of the group's children's bounding boxes; it shall not include any transformations performed by the group itself (i.e., the bounding box is defined in the local coordinate system of the children). Results are undefined if the specified bounding box is smaller than the true bounding box of the group.

4.6.5 Grouping and children nodes

Grouping nodes have a field that contains a list of children nodes. Each grouping node defines a coordinate space for its children. This coordinate space is relative to the coordinate space of the node of which the group node is a child. Such a node is called a *parent* node. This means that transformations accumulate down the scene graph hierarchy.

The following node types are grouping nodes:

- Anchor
- Billboard
- Collision
- Group
- Inline

- LOD
- Switch
- Transform

The following node types are children nodes:

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

The following node types are not valid as children nodes:

- Appearance
- AudioClip
- Box
- Color
- Cone
- Coordinate
- Cylinder
- ElevationGrid
- Extrusion
- ImageTexture
- IndexedFaceSet
- IndexedLineSet
- Material
- MovieTexture
- Normal
- PointSet
- Sphere
- Text
- TextureCoordinate
- TextureTransform

All grouping nodes except Inline, LOD, and Switch also have *addChildren* and *removeChildren* eventIn definitions. The *addChildren* event appends nodes to the grouping node's *children* field. Any nodes passed to the *addChildren* event that are already in the group's children list are ignored. For example, if the *children* field contains the nodes Q, L and S (in order) and the group receives an *addChildren* eventIn containing (in order) nodes A, L, and Z, the result is a *children* field containing (in order) nodes Q, L, S, A, and Z.

The *removeChildren* event removes nodes from the grouping node's *children* field. Any nodes in the *removeChildren* event that are not in the grouping node's *children* list are ignored. If the *children* field contains the nodes Q, L, S, A and Z and it receives a *removeChildren* eventIn containing nodes A, L, and Z, the result is Q, S.

Note that a variety of node types reference other node types through fields. Some of these are parent-child relationships, while others are not (there are node-specific semantics). Table 4.3 lists all node types that reference other nodes through fields.

Table 4.3 -- Nodes with SFNode or MFNode fields

Node Type	Field	Valid Node Types for Field
Anchor	*children*	Valid children nodes
Appearance	*material*	Material
	texture	ImageTexture, MovieTexture, Pixel Texture
Billboard	*children*	Valid children nodes
Collision	*children*	Valid children nodes
ElevationGrid	*color*	Color
	normal	Normal
	texCoord	TextureCoordinate
Group	*children*	Valid children nodes
IndexedFaceSet	*color*	Color
	coord	Coordinate
	normal	Normal
	texCoord	TextureCoordinate
IndexedLineSet	*color*	Color
	coord	Coordinate
LOD	*level*	Valid children nodes
Shape	*appearance*	Appearance
	geometry	Box, Cone, Cylinder, ElevationGrid, Extrusion, IndexedFaceSet, IndexedLineSet, PointSet, Sphere, Text
Sound	*source*	AudioClip, MovieTexture
Switch	*choice*	Valid children nodes
Text	*fontStyle*	FontStyle

Transform	*children*	Valid children nodes

4.6.6 Light sources

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

The following node types are light source nodes:

- DirectionalLight
- PointLight
- SpotLight

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

PointLight and SpotLight illuminate all objects in the world that fall within their volume of lighting influence regardless of location within the transformation hierarchy. PointLight defines this volume of influence as a sphere centred at the light (defined by a radius). SpotLight defines the volume of influence as a solid angle defined by a radius and a cutoff angle. DirectionalLight nodes illuminate only the objects descended from the light's parent grouping node, including any descendent children of the parent grouping nodes.

4.6.7 Sensor nodes

4.6.7.1 Introduction to sensors

The following node types are sensor nodes:

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

Sensors are children nodes in the hierarchy and therefore may be parented by grouping nodes as described in 4.6.5, Grouping and children nodes.

Each type of sensor defines when an event is generated. The state of the scene graph after several sensors have generated events shall 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 depend on the ordering of the events.

It is possible to create dependencies between various types of sensors. For example, a TouchSensor may result in a change to a VisibilitySensor node's transformation, which in turn may cause the VisibilitySensor node's visibility status to change.

The following two sections classify sensors into two categories: *environmental sensors* and *pointing-device sensors*.

4.6.7.2 Environmental sensors

The following node types are environmental sensors:

- Collision
- ProximitySensor
- TimeSensor
- VisibilitySensor

The ProximitySensor detects when the user navigates into a specified region in the world. The ProximitySensor itself is not visible. The TimeSensor is a clock that has no geometry or location associated with it; it is used to start and stop time-based nodes such as interpolators. The VisibilitySensor detects when a specific part of the world becomes visible to the user. The Collision grouping node detects when the user collides with objects in the virtual world. Proximity, time, collision, and visibility sensors are each processed independently of whether others exist or overlap.

When environmental sensors are inserted into the transformation hierarchy and before the presentation is updated (i.e., read from file or created by a script), they shall generate events indicating any conditions which the sensor is intended to detect (see 4.10.3, Execution model). The conditions for individual sensor types to generate these initial events are defined in the individual node specifications in 6, Node reference.

4.6.7.3 Pointing-device sensors

Pointing-device sensors detect user pointing events such as the user clicking on a piece of geometry (i.e., TouchSensor). The following node types are pointing-device sensors:

- Anchor
- CylinderSensor
- PlaneSensor
- SphereSensor
- TouchSensor

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

The appearance properties of the geometry do not affect activation of the sensor. In particular, transparent materials or textures shall be treated as opaque with respect to activation of pointing-device sensors.

For a given user activation, the lowest enabled pointing-device sensor in the hierarchy is activated. All other pointing-device sensors above the lowest enabled pointing-device sensor are ignored. The hierarchy is defined by the geometry node over which the pointing-device sensor is located and the entire hierarchy upward. If there are multiple pointing-device sensors tied for lowest, each of these is activated simultaneously and independently, possibly resulting in multiple sensors activating and generating output simultaneously. This feature allows combinations of pointing-device sensors (e.g., TouchSensor and PlaneSensor). If a pointing-device sensor appears in the transformation hierarchy multiple times (DEF/USE), it shall be tested for activation in all of the coordinate systems in which it appears.

If a pointing-device sensor is not enabled when the pointing-device button is activated, it will not generate events related to the pointing device until after the pointing device is deactivated and the sensor is enabled (i.e., enabling a sensor in the middle of dragging does not result in the sensor activating immediately).

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

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

4.6.7.4 Drag sensors

Drag sensors are a subset of pointing-device sensors. There are three types of drag sensors: CylinderSensor, PlaneSensor, and SphereSensor. Drag sensors have two eventOuts in common, *trackPoint_changed* and *<value>_changed*. These eventOuts send events for each movement of the activated pointing device according to their "virtual geometry" (e.g., cylinder for CylinderSensor). The *trackPoint_changed* eventOut sends the intersection point of the *bearing* with the drag sensor's virtual geometry. The *<value>_changed* eventOut sends the sum of the relative change since activation plus the sensor's *offset* field. The type and name of *<value>_changed* depends on the drag sensor type: *rotation_changed* for CylinderSensor, *translation_changed* for PlaneSensor, and *rotation_changed* for SphereSensor.

To simplify the application of these sensors, each node has an *offset* and an *autoOffset* exposed field. When the sensor generates events as a response to the activated pointing device motion, *<value>_changed* sends the sum of the relative change since the initial activation plus the *offset* field value. If *autoOffset* is TRUE when the pointing-device is deactivated, the *offset* field is set to the sensor's last *<value>_changed* value and *offset* sends an *offset_changed* eventOut. This enables subsequent grabbing operations to accumulate the changes. If *autoOffset* is FALSE, the sensor does not set the *offset* field value at deactivation (or any other time).

4.6.7.5 Activating and manipulating sensors

The pointing device controls a pointer in the virtual world. While activated by the pointing device, a sensor will generate events as the pointer moves. Typically the pointing device may be categorized as either 2D (e.g., conventional mouse) or 3D (e.g., wand). It is suggested that the pointer controlled by a 2D device is mapped onto a plane a fixed distance from the viewer and perpendicular to the line of sight. The mapping of a 3D device may describe a 1:1 relationship between movement of the pointing device and movement of the pointer.

The position of the pointer defines a bearing which is used to determine which geometry is being indicated. When implementing a 2D pointing device it is suggested that the bearing is defined by the vector from the viewer position through the location of the pointer. When implementing a 3D pointing device it is suggested that the bearing is defined by extending a vector from the current position of the pointer in the direction indicated by the pointer.

In all cases the pointer is considered to be indicating a specific geometry when that geometry is intersected by the bearing. If the bearing intersects multiple sensors' geometries, only the sensor nearest to the pointer will be eligible for activation.

4.6.8 Interpolator nodes

Interpolator nodes are designed for linear keyframed animation. 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 shall be monotonically non-decreasing, otherwise the results are undefined. The keys are not restricted to any interval.

An interpolator node evaluates *f(t)* given any value of *t* (via the *set_fraction* eventIn) as follows: Let the *n* keys t_0, t_1, t_2, ..., t_{n-1} partition the domain (*-infinity*, *+infinity*) into the *n*+1 subintervals given by (*-infinity*, t_0), [t_0, t_1), [t_1, t_2), ... , [t_{n-1}, *+infinity*). Also, let the *n* values v_0, v_1, v_2, ..., v_{n-1} be the values of *f(t)* at the associated key values. The piecewise-linear interpolating function, *f(t)*, is defined to be

```
f(t) = $v_0$, if t <= $t_0$,
     = $v_{n-1}$, if t >= $t_{n-1}$,
     = linterp(t, $v_i$, $v_{i+1}$), if $t_i$ <= t <= $t_{i+1}$

where linterp(t,x,y) is the linear interpolant,
      i belongs to {0,1,..., n-2}.
```

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 is used as the limit of *f(t)* from the left, and the last specified value is 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.

The following node types are interpolator nodes, each based on the type of value that is interpolated:

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

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

```
    eventIn      SFFloat      set_fraction
    exposedField MFFloat      key           [...]
    exposedField MF<type>     keyValue      [...]
    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).

The *set_fraction* eventIn receives an SFFloat event and causes the interpolator function to evaluate, resulting in a *value_changed* eventOut with the same timestamp as the *set_fraction* event.

ColorInterpolator, OrientationInterpolator, PositionInterpolator, and ScalarInterpolator output a single-value field to *value_changed*. Each value in the *keyValue* field corresponds in order to the parameter value in the *key* field. Results are undefined if the number of values in the *key* field of an interpolator is not the same as the number of values in the *keyValue* field.

CoordinateInterpolator and NormalInterpolator send multiple-value results to *value_changed*. In this case, the *keyValue* field is an *n* x *m* array of values, where *n* is the number of values in the key field and *m* is the number of values at each keyframe. Each *m* values in the *keyValue* field correspond, in order, to a parameter value in the *key* field. Each *value_changed* event shall contain *m* interpolated values. Results are undefined if the number of values in the *keyValue* field divided by the number of values in the *key* field is not a positive integer.

If an interpolator node's *value* eventOut is read before it receives any inputs, *keyValue*[0] is returned if *keyValue* is not empty. If *keyValue* is empty (i.e., [ ]), the initial value for the eventOut type is returned (e.g., (0, 0, 0) for SFVec3f); see 5, Field and event reference, for initial event values.

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

4.6.9 Time-dependent nodes

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

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

A time-dependent node generates an *isActive* TRUE event when it becomes active and generates an *isActive* FALSE event when it becomes inactive. These are the only times at which an *isActive* event is generated. In particular, *isActive* events are not sent at each tick of a simulation.

A time-dependent node is inactive until its *startTime* is reached. When time *now* becomes greater than or equal to *startTime,* an *isActive* TRUE event is generated and the time-dependent node becomes active (*now* refers to the time at which the browser is simulating and displaying the virtual world). When a time-dependent node is read from a VRML file and the ROUTEs specified within the VRML file have been established, the node should determine if it is active and, if so, generate an *isActive* TRUE event and begin generating any other necessary events. However, if a node would have become inactive at any time before the reading of the VRML file, no events are generated upon the completion of the read.

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

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

A time-dependent node may be restarted while it is active by sending a *set_stopTime* event equal to the current time (which will cause the node to become inactive) and a *set_startTime* event, setting it to the current time or any time in the future. These events will have the same time stamp and should be processed as *set_stopTime,* then *set_startTime* to produce the correct behaviour.

The default values for each of the time-dependent nodes are specified such that any node with default values is already inactive (and, therefore, will generate no events upon loading). A time-dependent node can be defined such that it will be active upon reading by specifying *loop* TRUE. This use of a non-terminating time-dependent node should be used with caution since it incurs continuous overhead on the simulation.

Figure 4.2 illustrates the behavior of several common cases of time-dependent nodes. In each case, the initial conditions of *startTime*, *stopTime*, *loop*, and the time-dependent node's cycle interval are labelled, the red region denotes the time period during which the time-dependent node is active, the arrows represent eventIns received by and eventOuts sent by the time-dependent node, and the horizontal axis represents time.

4.6.10 Bindable children nodes

The Background, Fog, NavigationInfo, and Viewpoint nodes have the unique behaviour that only one of each type can be bound (i.e., affecting the user's experience) at any instant in time. The browser shall maintain an independent, separate stack for each type of bindable node. Each of these nodes includes a *set_bind* eventIn and an *isBound* eventOut. The *set_bind* eventIn is used to move a given node to and from its respective top of stack. A TRUE value sent to the *set_bind* eventIn moves the node to the top of the stack; sending a FALSE value removes it from the stack. The *isBound* event is output when a given node is:

a. moved to the top of the stack;

b. removed from the top of the stack;

c. pushed down from the top of the stack by another node being placed on top.

That is, *isBound* events are sent when a given node becomes, or ceases to be, the active node. The node at the top of stack, (the most recently bound node), is the active node for its type and is used by the browser to set the world state. If the stack is empty (i.e., either the VRML file has no bindable nodes for a given type or the stack has been popped until empty), the default field values for that node type are used to set world state. The results are undefined if a multiply instanced (DEF/USE) bindable node is bound.

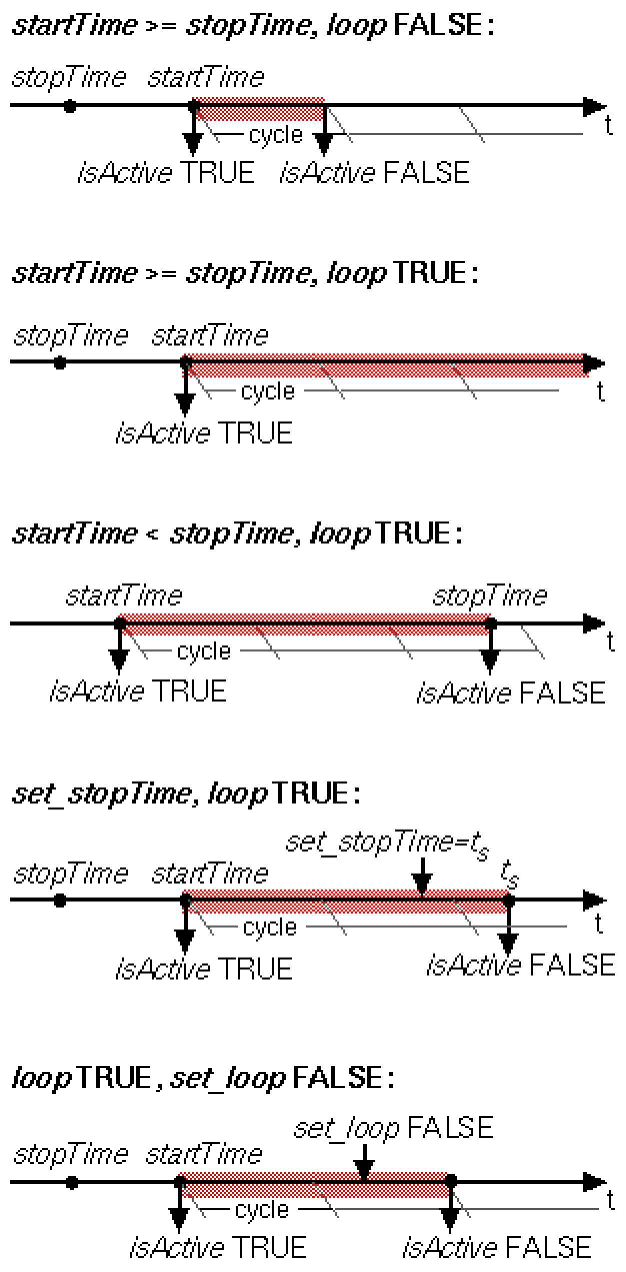

startTime >= stopTime, loop FALSE:
stopTime startTime
cycle
t
isActive TRUE isActive FALSE
startTime >= stopTime, loop TRUE:
stopTime startTime
cycle
t
isActive TRUE
startTime < stopTime, loop TRUE:
startTime stopTime
cycle
t
isActive TRUE isActive FALSE
set_stopTime, loop TRUE:
set_stopTime=t_s
stopTime startTime t_s
cycle
t
isActive TRUE isActive FALSE
loop TRUE, set_loop FALSE:
set_loop FALSE
stopTime startTime
cycle
t
isActive TRUE isActive FALSE

Figure 4.2 -- Examples of time-dependent node execution

The following rules describe the behaviour of the binding stack for a node of type <*bindable node*>, (Background, Fog, NavigationInfo, or Viewpoint):

a. During read, the first encountered <*bindable node*> is bound by pushing it to the top of the <*bindable node*> stack. Nodes contained within Inlines, within the strings passed to the Browser.createVrmlFromString() method, or within VRML files passed to the Browser.createVrmlFromURL() method (see 4.12.10, Browser script interface)are not candidates for the first encountered <*bindable node*>. The first node within a prototype instance is a valid candidate for the first encountered <*bindable node*>. The first encountered <*bindable node*> sends an *isBound* TRUE event.

b. When a *set_bind* TRUE event is received by a <*bindable node*>,

   1. If it is not on the top of the stack: the current top of stack node sends an *isBound* FALSE event. The new node is moved to the top of the stack and becomes the currently bound <*bindable node*>. The new <*bindable node*> (top of stack) sends an *isBound* TRUE event.

   2. If the node is already at the top of the stack, this event has no effect.

c. When a *set_bind* FALSE event is received by a <*bindable node*> in the stack, it is removed from the stack. If it was on the top of the stack,

   1. it sends an *isBound* FALSE event;

   2. the next node in the stack becomes the currently bound <*bindable node*> (i.e., pop) and issues an *isBound* TRUE event.

d. If a *set_bind* FALSE event is received by a node not in the stack, the event is ignored and *isBound* events are not sent.

e. When a node replaces another node at the top of the stack, the *isBound* TRUE and FALSE eventOuts from the two nodes are sent simultaneously (i.e., with identical timestamps).

f. If a bound node is deleted, it behaves as if it received a *set_bind* FALSE event (see f above).

4.6.11 Texture maps

4.6.11.1 Texture map formats

Four node types specify texture maps: Background, ImageTexture, MovieTexture, and PixelTexture. In all cases, texture maps are defined by 2D images that contain an array of colour values describing the texture. The texture map values are interpreted differently depending on the number of components in the texture map and the specifics of the image format. In general, texture maps may be described using one of the following forms:

a. *Intensity textures* (one-component)

b. *Intensity plus alpha opacity textures* (two-component)

c. *Full RGB textures* (three-component)

d. *Full RGB plus alpha opacity textures* (four-component)

Note that most image formats specify an alpha opacity, not transparency (where alpha = 1 - transparency).

See Table 4.5 and Table 4.6 for a description of how the various texture types are applied.

4.6.11.2 Texture map image formats

Texture nodes that require support for the PNG (see 2.[PNG]) image format (6.5, Background, and 6.22, ImageTexture) shall interpret the PNG pixel formats in the following way:

a. Greyscale pixels without alpha or simple transparency are treated as intensity textures.

b. Greyscale pixels with alpha or simple transparency are treated as intensity plus alpha textures.

c. RGB pixels without alpha channel or simple transparency are treated as full RGB textures.

d. RGB pixels with alpha channel or simple transparency are treated as full RGB plus alpha textures.

If the image specifies colours as indexed-colour (i.e., palettes or colourmaps), the following semantics should be used (note that `greyscale' refers to a palette entry with equal red, green, and blue values):

a. If all the colours in the palette are greyscale and there is no transparency chunk, it is treated as an intensity texture.

b. If all the colours in the palette are greyscale and there is a transparency chunk, it is treated as an intensity plus opacity texture.

c. If any colour in the palette is not grey and there is no transparency chunk, it is treated as a full RGB texture.

d. If any colour in the palette is not grey and there is a transparency chunk, it is treated as a full RGB plus alpha texture.

Texture nodes that require support for JPEG files (see 2.[JPEG], 6.5, Background, and 6.22, ImageTexture) shall interpret JPEG files as follows:

i. Greyscale files (number of components equals 1) are treated as intensity textures.

j. YCbCr files are treated as full RGB textures.

k. No other JPEG file types are required. It is recommended that other JPEG files are treated as a full RGB textures.

Texture nodes that support MPEG files (see 2.[MPEG] and 6.28, MovieTexture) shall treat MPEG files as full RGB textures.

Texture nodes that recommend support for GIF files (see E.[GIF], 6.5, Background, and 6.22, ImageTexture) shall follow the applicable semantics described above for the PNG format.

4.7 Field, eventIn, and eventOut semantics

Fields are placed inside node statements in a VRML file, and define the persistent state of the virtual world. Results are undefined if multiple values for the same field in the same node (e.g., **`Sphere { radius 1.0 radius 2.0 }`**) are declared.

EventIns and eventOuts define the types and names of events that each type of node may receive or generate. Events are transient and event values are not written to VRML files. Each node interprets the values of the events sent to it or generated by it according to its implementation.

Field, eventIn, and eventOut types, and field encoding syntax, are described in 5, Field and event reference.

An *exposedField* can receive events like an eventIn, can generate events like an eventOut, and can be stored in VRML files like a field. An exposedField named *zzz* can be referred to as '*set_zzz*' and treated as an eventIn, and can be referred to as '*zzz_changed*' and treated as an eventOut. The initial value of an exposedField is its value in the VRML file, or the default value for the node in which it is contained, if a value is not specified. When an exposedField receives an event it shall generate an event with the same value and timestamp. The following sources, in precedence order, shall be used to determine the initial value of the exposedField:

a. the user-defined value in the instantiation (if one is specified);

b. the default value for that field as specified in the node or prototype definition.

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

c. All names containing multiple words start with a lower case letter, and the first letter of all subsequent words is capitalized (e.g., *addChildren*), with the exception of *set_* and *_changed*, as described below.

d. All eventIns have the prefix "*set_*", with the exception of the *addChildren* and *removeChildren* eventIns.

e. Certain eventIns and eventOuts of type SFTime do not use the "*set_*" prefix or "*_changed*" suffix.

f. All other eventOuts have the suffix "*_changed*" appended, with the exception of eventOuts of type SFBool. Boolean eventOuts begin with the word "*is*" (e.g., *isFoo*) for better readability.

4.8 Prototype semantics

4.8.1 Introduction

The PROTO statement defines a new node type in terms of already defined (built-in or prototyped) node types. Once defined, prototyped node types may be instantiated in the scene graph exactly like the built-in node types.

Node type names shall be unique in each VRML file. The results are undefined if a prototype is given the same name as a built-in node type or a previously defined prototype in the same scope.

4.8.2 PROTO interface declaration semantics

The prototype interface defines the fields, eventIns, and eventOuts for the new node type. The interface declaration includes the types and names for the eventIns and eventOuts of the prototype, as well as the types, names, and default values for the prototype's fields.

The interface declaration may contain exposedField declarations, which are a convenient way of defining a field, eventIn, and eventOut at the same time. If an exposedField named *zzz* is declared, it is equivalent to declaring a field named *zzz*, an eventIn named *set_zzz*, and an eventOut named *zzz_changed*.

Each prototype instance can be considered to be a complete copy of the prototype, with its own fields, events, and copy of the prototype definition. A prototyped node type is instantiated using standard node syntax. For example, the following prototype (which has an empty interface declaration):

```
PROTO Cube [ ] { Box { } }
```

may be instantiated as follows:

```
Shape { geometry Cube { } }
```

It is recommended that user-defined field or event names defined in PROTO interface declarations statements follow the naming conventions described in 4.7, Field, eventIn, and eventOut semantics.

If an eventOut in the prototype declaration is associated with an exposedField in the prototype definition, the initial value of the eventOut shall be the initial value of the exposedField. If the eventOut is associated with multiple exposedFields, the results are undefined.

4.8.3 PROTO definition semantics

A prototype definition consists of one or more nodes, nested PROTO statements, and ROUTE statements. The first node type determines how instantiations of the prototype can be used in a VRML file. An instantiation is created by filling in the parameters of the prototype declaration and inserting copies of the first node (and its scene graph) wherever the prototype instantiation occurs. For example, if the first node in the prototype definition is a Material node, instantiations of the prototype can be used wherever a Material node can be used. Any other nodes and accompanying scene graphs are not part of the transformation hierarchy, but may be referenced by ROUTE statements or Script nodes in the prototype definition.

Nodes in the prototype definition may have their fields, eventIns, or eventOuts associated with the fields, eventIns, and eventOuts of the prototype interface declaration. This is accomplished using IS statements in the body of the node. When prototype instances are read from a VRML file, field values for the fields of the prototype interface may be given. If given, the field values are used for all nodes in the prototype definition that have IS statements for those fields. Similarly, when a prototype instance is sent an event, the event is delivered to all nodes that have IS statements for that event. When a node in a prototype instance generates an event that has an IS statement, the event is sent to any eventIns connected (via ROUTE) to the prototype instance's eventOut.

IS statements may appear inside the prototype definition wherever fields may appear. IS statements shall refer to fields or events defined in the prototype declaration. Results are undefined if an IS statement refers to a non-existent declaration. Results are undefined if the type of the field or event being associated by the IS statement does not match the type declared in the prototype's interface declaration. For example, it is illegal to associate an SFColor with an SFVec3f. It is also illegal to associate an SFColor with an MFColor or *vice versa*.

Results are undefined if an IS statement:

- eventIn is associated with a field or an eventOut;
- eventOut is associated with a field or eventIn;
- field is associated with an eventIn or eventOut.

An exposedField in the prototype interface may be associated only with an exposedField in the prototype definition, but an exposedField in the prototype definition may be associated with either a field, eventIn, eventOut or exposedField in the prototype interface. When associating an exposedField in a prototype definition with an eventIn or eventOut in the prototype declaration, it is valid to use either the shorthand exposedField name (e.g., *translation*) or the explicit event name (e.g., *set_translation* or *translation_changed*). Table 4.4 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).

Table 4.4 -- Rules for mapping PROTOTYPE declarations to node instances

		Prototype declaration			
		exposedField	**field**	**eventIn**	**eventOut**
Prototype definition	**exposedField**	yes	yes	yes	yes
	field	no	yes	no	no
	eventIn	no	no	yes	no
	eventOut	no	no	no	yes

Results are undefined if a field, eventIn, or eventOut of a node in the prototype definition is associated with more than one field, eventIn, or eventOut in the prototype's interface (i.e., multiple IS statements for a field, eventIn, and eventOut in a node in the prototype definition), but multiple IS statements for the fields, eventIns, and eventOuts in the prototype interface declaration is valid. Results are undefined if a field of a node in a prototype definition is both defined with initial values (i.e., field statement) and associated by an IS statement with a field in the prototype's interface. If a prototype interface has an eventOut E associated with multiple eventOuts in the prototype definition ED_i , the value of E is the value of the eventOut that generated the event with the greatest timestamp. If two or more of the eventOuts generated events with identical timestamps, results are undefined.

4.8.4 Prototype scoping rules

Prototype definitions appearing inside a prototype definition (i.e., nested) are local to the enclosing prototype. IS statements inside a nested prototype's implementation may refer to the prototype declarations of the innermost prototype.

A PROTO statement establishes a DEF/USE name scope separate from the rest of the scene and separate from any nested PROTO statements. Nodes given a name by a DEF construct inside the prototype may not be referenced in a USE construct outside of the prototype's scope. Nodes given a name by a DEF construct outside the prototype scope may not be referenced in a USE construct inside the prototype scope.

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).

4.9 External prototype semantics

4.9.1 Introduction

The EXTERNPROTO statement defines a new node type. It is equivalent to the PROTO statement, with two exceptions. First, the implementation of the node type is stored externally, either in a VRML file containing an appropriate PROTO statement or using some other implementation-dependent mechanism. Second, default values for fields are not given since the implementation will define appropriate defaults.

4.9.2 EXTERNPROTO interface semantics

The semantics of the EXTERNPROTO are exactly the same as for a PROTO statement, except that default field and exposedField values are not specified locally. In addition, events sent to an instance of an externally prototyped node may be ignored until the implementation of the node is found.

Until the definition has been loaded, the browser shall determine the initial value of exposedFields using the following rules (in order of precedence):

a. the user-defined value in the instantiation (if one is specified);

b. the default value for that field type.

For eventOuts, the initial value on startup will be the default value for that field type. During the loading of an EXTERNPROTO, if an initial value of an eventOut is found, that value is applied to the eventOut and no event is generated.

The names and types of the fields, exposedFields, eventIns, and eventOuts of the interface declaration shall be a subset of those defined in the implementation. Declaring a field or event with a non-matching name is an error, as is declaring a field or event with a matching name but a different type.

It is recommended that user-defined field or event names defined in EXTERNPROTO interface statements follow the naming conventions described in 4.7, Field, eventIn, and eventOut semantics.

4.9.3 EXTERNPROTO URL semantics

The string or strings specified after the interface declaration give the location of the prototype's implementation. If multiple strings are specified, the browser searches in the order of preference (see 4.5.2, URLs).

If a URL in an EXTERNPROTO statement refers to a VRML file, the first PROTO statement found in the VRML file (excluding EXTERNPROTOs) is used to define the external prototype's definition. The name of that prototype does not need to match the name given in the EXTERNPROTO statement. Results are undefined if a URL in an EXTERNPROTO statement refers to a non-VRML file

To enable the creation of libraries of reusable PROTO definitions, browsers shall recognize EXTERNPROTO URLs that end with "#*name*" to mean the PROTO statement for "name" in the given VRML file. For example, a library of standard materials might be stored in a VRML 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 GoldFromLibrary [] "http://.../materials.wrl#Gold"
...
Shape {
    appearance Appearance { material GoldFromLibrary {} }
    geometry   ...
}
...
```

4.10 Event processing

4.10.1 Introduction

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

If an eventOut is read before it has sent any events, the *initial value* as specified in 5, Field and event reference, for each field/event type is returned.

4.10.2 Route semantics

The connection between the node generating the event and the node receiving the event is called a *route*. Routes are not nodes. The ROUTE statement is a construct for establishing event paths between nodes. ROUTE statements may either appear at the top level of a VRML file, in a prototype definition, or inside a node wherever fields may appear. Nodes referenced in a ROUTE statement shall be defined before the ROUTE statement.

The types of the eventIn and the eventOut shall 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. For convenience, when routing to or from an eventIn or eventOut (or the eventIn or eventOut part of an exposedField), the *set_* or *_changed* part of the event's name is optional. If the browser is trying to establish a ROUTE to an eventIn named *zzz* and an eventIn of that name is not found, the browser shall then try to establish the ROUTE to the eventIn named *set_zzz*. Similarly, if establishing a ROUTE from an eventOut named *zzz* and an eventOut of that name is not found, the browser shall try to establish the ROUTE from *zzz_changed*.

Redundant routing is ignored. If a VRML file repeats a routing path, the second and subsequent identical routes are ignored. This also applies for routes created dynamically via a scripting language supported by the browser.

4.10.3 Execution model

Once a sensor or Script has generated an *initial event*, the event is propagated from the eventOut producing the event along any ROUTEs to other nodes. These other nodes may respond by generating additional events, continuing until all routes have been honoured. This process is called an *event cascade*. All events generated during a given event cascade are assigned the same timestamp as the initial event, since all are considered to happen instantaneously.

Some sensors generate multiple events simultaneously. Similarly, it is possible that asynchronously generated events could arrive at the identical time as one or more sensor generated event. In these cases, all events generated are part of the same initial event cascade and each event has the same timestamp.

After all events of the initial event cascade are honored, post-event processing performs actions stimulated by the event cascade. The entire sequence of events occuring in a single timestamp are:

a. Perform event cascade evaluation.

b. Call *shutdown( )* on scripts that have received *set_url* events or are being removed from the scene.

c. Send final events from environmental sensors being removed from the transformation hierarchy.

d. Add or remove routes specified in *addRoute( )* or *deleteRoute( )* from any script execution in the preceeding event cascade.

e. Call *eventsProcessed( )* for scripts that have sent events in the just ended event cascade.

f. Send initial events from any dynamically created environmental sensors.

g. Call *initialize( )* of newly loaded script code.

h. If any events were generated from steps 2 through 7, go to step 2 and continue.

Figure 4.3 provides a conceptual illustration of the execution model.

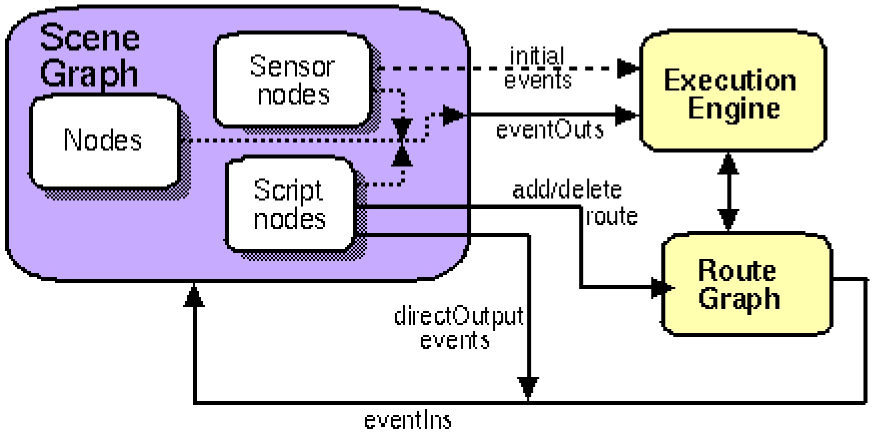


Figure 4.3 -- Conceptual execution model

Nodes that contain eventOuts or exposedFields shall produce at most one event per timestamp. If a field is connected to another field via a ROUTE, an implementation shall send only one event per ROUTE per timestamp. This also applies to scripts where the rules for determining the appropriate action for sending eventOuts are defined in 4.12.9.3, Sending eventOuts.

D.19, Execution model, provides an example that demonstrates the execution model. Figure 4.4 illustrates event processing for a single timestamp in example in D.19, Execution model:

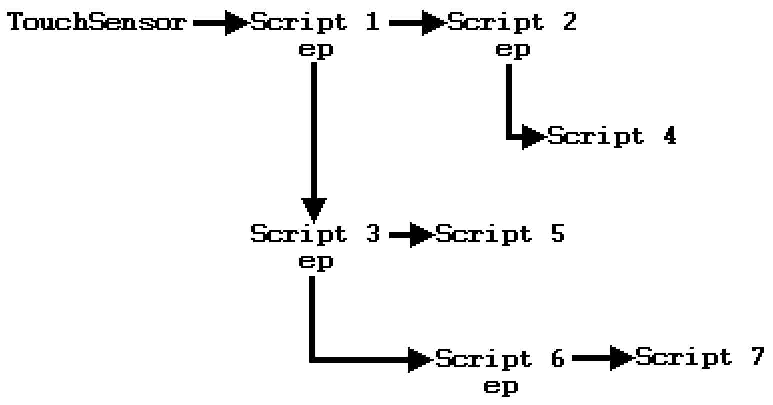


Figure 4.4 -- Example D.19, event processing order

In Figure 4.4, arrows coming out of a script at **ep** are events generated during the *eventsProcessed()* call for the script. The other arrows are events sent during an eventIn method. One possible compliant order of execution is as follows:

i. User activates **TouchSensor**

j. Run initial event cascade (step 1)

  1. **Script 1** runs, generates an event for **Script 2**
  2. **Script 2** runs

  3. end of initial event cascade

k. Execute eventsProcessed calls (step 5)

  1. *eventsProcessed* for **Script 1** runs, sends event to **Script 3**
  2. **Script 3** runs, generates events for **Script 5**
  3. **Script 5** runs
  4. *eventsProcessed* for **Script 2** runs, sends events to **Script 4**
  5. **Script 4** runs
  6. end of *eventsProcessed* processing

l. Go to step 2 for generated events (step 8)

m. Execute *eventsProcessed* calls (step 5)

  1. *eventsProcessed* for **Script 3** runs, sends event to **Script 6**
  2. **Script 6** runs, sends event to **Script 7**
  3. **Script 7** runs
  4. *eventsProcessed* for **Script 4** runs, does not generate any events
  5. *eventsProcessed* for **Script 5** runs, does not generate any events
  6. end of *eventsProcessed* processing

n. Go to step 2 for generated events (step 8)

o. Execute *eventsProcessed* calls (step 5)

  1. *eventsProcessed* for **Script 6** runs, does not generate any events
  2. *eventsProcessed* for **Script 7** runs, does not generate any events
  3. end of *eventsProcessed* processing

p. No more events to handle.

The above is not the only possible compliant order of execution. If multiple *eventsProcessed()* methods are pending when step 4 is executed, the order in which these methods is called is not defined. For instance, in the third step of the example, the *eventsProcessed* method is pending for both **Script** 1 and **Script 2**. The order of execution in this case is not defined, so executing the *eventsProcessed* method of **Script 2** before that of **Script 1** would have been compliant. However, executing the *eventsProcessed* method for **Script 3** before that of **Script 2** would not have been compliant because any methods made pending during processing must wait until the next iteration of the event cascade for execution.

4.10.4 Loops

Event cascades may contain *loops* where an event *E* is routed to a node that generates an event that eventually results in *E* being generated again. See 4.10.3, Execution model, for the loop breaking rule that limits each eventOut to one event per timestamp. This rule shall also be used to break loops created by cyclic dependencies between different sensor nodes.

4.10.5 Fan-in and fan-out

Fan-in occurs when two or more routes write to the same eventIn. Events coming into an eventIn from different eventOuts with the same timestamp shall be processed, but the order of evaluation is implementation dependent.

Fan-out occurs when one eventOut routes to two or more eventIns. This results in sending any event generated by the eventOut to all of the eventIns.

4.11 Time

4.11.1 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 approximate "real" time. A world's creator should make no assumptions about how often a TimeSensor will generate events but can safely assume that each time event generated will have a timestamp greater than any previous time event.

4.11.2 Time origin

Time (0.0) is equivalent to 00:00:00 GMT January 1, 1970. Absolute times are specified in SFTime or MFTime fields as double-precision floating point numbers representing seconds. Negative absolute times are interpreted as happening before 1970.

Processing an event with timestamp *t* may only result in generating events with timestamps greater than or equal to *t*.

4.11.3 Discrete and continuous changes

ISO/IEC 14772 does not distinguish between discrete events (such as those generated by a TouchSensor) and events that are the result of sampling a conceptually continuous set of changes (such as the fraction events generated by a TimeSensor). An ideal VRML implementation would generate an infinite number of samples for continuous changes, each of which would be processed infinitely quickly.

Before processing a discrete event, all continuous changes that are occurring at the discrete event's timestamp shall 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, the sampling frequency of continuous changes is implementation dependent. 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.

4.12 Scripting

4.12.1 Introduction

Authors often require that VRML worlds change dynamically in response to user inputs, external events, and the current state of the world. The proposition "if the vault is currently closed AND the correct combination is entered, open the vault" illustrates the type of problem which may need addressing. These kinds of decisions are expressed as Script nodes (see 6.40, Script) that receive events from other nodes, process them, and send events to other nodes. A Script node can also keep track of information between subsequent executions (i.e., retaining internal state over time).

This subclause describes the general mechanisms and semantics of all scripting language access protocols. Note that no scripting language is required by ISO/IEC 14772. Details for two scripting languages are in annex B, Java platform scripting reference, and annex C, ECMAScript scripting reference, respectively. If either of these scripting languages are implemented, the Script node implementation shall conform with the definition described in the corresponding annex.

Event processing is performed by a program or script contained in (or referenced by) the Script node's *url* field. This program or script may be written in any programming language that the browser supports.

4.12.2 Script execution

A Script node is activated when it receives an event. The browser shall then execute 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 including sending out events (and thereby changing the scene), performing calculations, and communicating with servers elsewhere on the Internet. A detailed description of the ordering of event processing is contained in 4.10, Event processing.

Script nodes may also be executed after they are created (see 4.12.3, Initialize() and shutdown()). Some scripting languages may allow the creation of separate processes from scripts, resulting in continuous execution (see 4.12.6, Asynchronous scripts).

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

When a *set_url* event is received by a Script node that contains a script that has been previously initialized for a different URL, the *shutdown()* method of the current script is called (see 4.12.3, Initialize() and shutdown()). Until the new script becomes available, the script shall behave as though it has no executable content. When the new script becomes available, the *Initialize()* method is invoked as defined in 4.10.3, Execution model. The limiting case is when the URL contains inline code that can be immediately executed upon receipt of the *set_url* event (e.g., javascript: protocol). In this case, it can be assumed that the old code is unloaded and the new code loaded instantaneously, after any dynamic route requests have been performed.

4.12.3 *Initialize()* and *shutdown()*

The scripting language binding may define an *initialize()* method. This method shall be invoked before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script. Events generated by the *initialize()* method shall have timestamps less than any other events generated by the Script node. This allows script initialization tasks to be performed prior to the user interacting with the world.

Likewise, the scripting language binding may define a *shutdown()* method. This method shall be invoked when the corresponding Script node is deleted or the world containing the Script node is unloaded or replaced by another world. This method may be used as a clean-up operation, such as informing external mechanisms to remove temporary files. No other methods of the script may be invoked after the *shutdown()* method has completed, though the *shutdown()* method may invoke methods or send events while shutting down. Events generated by the *shutdown()* method that are routed to nodes that are being deleted by the same action that caused the *shutdown()* method to execute will not be delivered. The deletion of the Script node containing the *shutdown()* method is not complete until the execution of its *shutdown()* method is complete.

4.12.4 *EventsProcessed()*

The scripting language binding may define an *eventsProcessed()* method that is called after one or more events are received. This method 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 time-dependent way, *eventsProcessed()* may be nondeterministic, since different browser implementations may call *eventsProcessed()* at different times.

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

4.12.5 Scripts with direct outputs

Scripts that have access to other nodes (via SFNode/MFNode fields or eventIns) and that have their *directOutput* 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 shall be up-to-date; that is, it shall be the value immediately before the time of the current timestamp (the current timestamp returned is the timestamp of the event that caused the Script node to execute).

If multiple *directOutput* Scripts read from and/or write to the same node, the results are undefined.

4.12.6 Asynchronous scripts

Some languages supported by VRML browsers may allow 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 events that causes the event cascade, and the scripting language and/or the browser shall 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 including those for looping.

4.12.7 Script languages

The Script node's *url* field may specify a URL which refers to a file (e.g., using protocol http:) or incorporates scripting language code directly in-line. The MIME-type of the returned data defines the language type. Additionally, instructions can be included in-line using 4.5.4, Scripting language protocol, defined for the specific language (from which the language type is inferred).

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

```
Script {
  eventIn SFBool start
  url [ "http://foo.com/fooBar.class",
    "http://foo.com/fooBar.js",
    "javascript:function start(value, timestamp) { ... }"
  ]
}
```

In the above example when a *start* eventIn is received by the Script node, one of the scripts found in the *url* field is executed. The Java platform bytecode is the first choice, the ECMAScript code is the second choice, and the inline ECMAScript code the third choice. A description of order of preference for multiple valued URL fields may be found in 4.5.2, URLs.

4.12.8 EventIn handling

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

4.12.9 Accessing fields and events

The fields, eventIns, and eventOuts of a Script node are accessible from scripting language methods. Events can be routed to eventIns of Script nodes and the eventOuts of Script nodes can be routed to eventIns of other nodes. Another Script node with access to this node can access the eventIns and eventOuts just like any other node (see 4.12.5, Scripts with direct outputs).

It is recommended that user-defined field or event names defined in Script nodes follow the naming conventions described in 4.7, Field, eventIn, and eventOut semantics.

4.12.9.1 Accessing fields and eventOuts of the script

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

4.12.9.2 Accessing eventIns and eventOuts of other nodes

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

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

The language-dependent mechanism for accessing eventIns or eventOuts (or the eventIn or eventOut part of an exposedField) shall support accessing them without their "*set_*" or "*_changed*" prefix or suffix, to match the ROUTE statement semantics. When accessing an eventIn named "zzz" and an eventIn of that name is not found, the browser shall try to access the eventIn named "*set_zzz*". Similarly, if accessing an eventOut named "zzz" and an eventOut of that name is not found, the browser shall try to access the eventOut named "*zzz_changed*".

4.12.9.3 Sending eventOuts

Each scripting language provides a mechanism for allowing scripts to send a value through an eventOut defined by the Script node. For example, one scripting language may define an explicit method for sending each eventOut, while another language may use assignment statements to automatically defined eventOut variables to implicitly send the eventOut. Sending multiple values through an eventOut during a single script execution will result in the "last" event being sent, where "last" is determined by the semantics of the scripting language being used.

4.12.10 Browser script interface

4.12.10.1 Introduction

The browser interface provides a mechanism for scripts contained by Script nodes to get and set browser state (e.g., the URL of the current world). This subclause describes the semantics of methods that the browser interface supports. An arbitrary syntax is used to define the type of parameters and returned values. The specific annex for a language contains the actual syntax required. In this abstract syntax, types are given as VRML field types. Mapping of these types into those of the underlying language (as well as any type conversion needed) is described in the appropriate language annex.

4.12.10.2 SFString getName( ) and SFString getVersion( )

The **getName()** and **getVersion()** methods return a string representing 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.

4.12.10.3 SFFloat getCurrentSpeed( )

The **getCurrentSpeed()** method returns the average navigation speed for the currently bound NavigationInfo node in meters per second, in the coordinate system of the currently bound Viewpoint node. 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.

4.12.10.4 SFFloat getCurrentFrameRate( )

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

4.12.10.5 SFString getWorldURL( )

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

4.12.10.6 void replaceWorld(MFNode nodes)

The **replaceWorld()** method replaces the current world with the world represented by the passed nodes. An invocation of this method will usually not return since the world containing the running script is being replaced. Scripts that may call this method shall have *mustEvaluate* set to TRUE.

4.12.10.7 void loadURL(MFString url, MFString parameter)

The **loadURL()** method loads the first recognized URL from the specified *url* field with the passed parameters. The *parameter* and *url* arguments are treated identically to the Anchor node's *parameter* and *url* fields (see 6.2, Anchor). This method returns immediately. However, if the URL is loaded into this browser window (e.g., there is no TARGET parameter to redirect it to another frame), the current world will be terminated and replaced with the data from the specified URL at some time in the future. Scripts that may call this method shall set *mustEvaluate* to TRUE. If **loadUrl()** is invoked with a URL of the form "#name", the Viewpoint node with the given name ("name") in the Script' node's run-time name scope(s) shall be bound. However, if the Script node containing the script that invokes **loadURL("#name")** is not part of any run-time name scope or is part of more than one run-time name scope, results are undefined. See 4.4.6, Run-time name scope, for a description of run-time name scope.

4.12.10.8 void setDescription(SFString description)

The **setDescription()** method sets the passed string as the current description. This message is displayed in a browser dependent manner. An empty string clears the current description. Scripts that call this method shall have *mustEvaluate* set to TRUE.

4.12.10.9 MFNode createVrmlFromString(SFString vrmlSyntax)

The **createVrmlFromString**() method parses a string consisting of VRML statements, establishes any PROTO and EXTERNPROTO declarations and routes, and returns an MFNode value containing the set of nodes in those statements. The string shall be self-contained (i.e., USE statements inside the string may refer only to nodes DEF'ed in the string, and non-built-in node types used by the string shall be prototyped using EXTERNPROTO or PROTO statements inside the string).

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

The **createVrmlFromURL()** instructs the browser to load a VRML scene description from the given URL or URLs. The VRML file referred to shall be self-contained (i.e., USE statements inside the string may refer only to nodes DEF'ed in the string, and non-built-in node types used by the string shall be prototyped using EXTERNPROTO or PROTO statements inside the string). After the scene is loaded, *event* is sent to the passed *node* returning the root nodes of the corresponding VRML scene. The *event* parameter contains a string naming an MFNode eventIn on the passed node.

4.12.10.11 void addRoute(...) and void deleteRoute(...)

```
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. Scripts that call this method shall have *directOutput* set to TRUE. Routes that are added and deleted shall obey the execution order defined in 4.10.3, Execution model.

4.13 Navigation

4.13.1 Introduction

Conceptually speaking, every VRML world contains a *viewpoint* from which the world is currently being viewed. Navigation is the action taken by the user to change the position and/or orientation of this viewpoint thereby changing the user's view. This allows the user to move through a world or examine an object. The NavigationInfo node (see 6.29, NavigationInfo) specifies the characteristics of the desired navigation behaviour, but the exact user interface is browser-dependent. The Viewpoint node (see 6.53, Viewpoint) specifies key locations and orientations in the world to which the user may be moved via scripts or browser-specific user interfaces.

4.13.2 Navigation paradigms

The browser may allow the user to modify the location and orientation of the viewer in the virtual world using a navigation paradigm. Many different navigation paradigms are possible, depending on the nature of the virtual world and the task the user wishes to perform. For instance, a walking paradigm would be appropriate in an architectural walkthrough application, while a flying paradigm might be better in an application exploring interstellar space. Examination is another common use for VRML, where the world is considered to be a single object which the user wishes to view from many angles and distances.

The NavigationInfo node has a *type* field that specifies the navigation paradigm for this world. The actual user interface provided to accomplish this navigation is browser-dependent. See 6.29, NavigationInfo, for details.

4.13.3 Viewing model

The browser controls the location and orientation of the viewer in the world, based on input from the user (using the browser-provided navigation paradigm) and the motion of the currently bound Viewpoint node (and its coordinate system). The VRML author can place any number of viewpoints in the world at important places from which the user might wish to view the world. Each viewpoint is described by a Viewpoint node. Viewpoint nodes exist in their parent's coordinate system, and both the viewpoint and the coordinate system may be changed to affect the view of the world presented by the browser. Only one viewpoint is bound at a time. A detailed description of how the Viewpoint node operates is described in 4.6.10, Bindable children nodes, and 6.53, Viewpoint.

Navigation is performed relative to the Viewpoint's location and does not affect the location and orientation values of a Viewpoint node. The location of the viewer may be determined with a ProximitySensor node (see 6.38, ProximitySensor).

4.13.4 Collision detection and terrain following

A VRML file can contain Collision nodes (see 6.8, Collision) and NavigationInfo nodes that influence the browser's navigation paradigm. The browser is responsible for detecting collisions between the viewer and the objects in the virtual world, and is also responsible for adjusting the viewer's location when a collision occurs. Browsers shall not disable collision detection except for the special cases listed below. Collision nodes can be used to generate events when viewer and objects collide, and can be used to designate that certain objects should be treated as transparent to collisions. Support for inter-object collision is not specified. The NavigationInfo types of WALK, FLY, and NONE shall strictly support collision detection. However, the NavigationInfo types ANY and EXAMINE may temporarily disable collision detection during navigation, but shall not disable collision detection during the normal execution of the world. See 6.29, NavigationInfo, for details on the various navigation types.

NavigationInfo nodes can be used to specify certain parameters often used by browser navigation paradigms. The size and shape of the viewer's avatar determines how close the avatar may be to an object before a collision is considered to take place. These parameters can also be used to implement *terrain following* by keeping the avatar a certain distance above the ground. They can additionally be used to determine how short an object must be for the viewer to automatically step up onto it instead of colliding with it.

4.14 Lighting model

4.14.1 Introduction

The VRML lighting model provides detailed equations which define the colours to apply to each geometric object. For each object, the values of the Material node, Color node and texture currently being applied to the object are combined with the lights illuminating the object and the currently bound Fog node. These equations are designed to simulate the physical properties of light striking a surface.

4.14.2 Lighting 'off'

A Shape node is unlit if either of the following is true:

a. The shape's *appearance* field is NULL (default).

b. The *material* field in the Appearance node is NULL (default).

×Note the special cases of geometry nodes that do not support lighting (see 6.24, IndexedLineSet, and 6.36, PointSet, for details).

If the shape is unlit, the colour (I_{rgb}) and alpha (A, 1-transparency) of the shape at each point on the shape's geometry is given in Table 4.5.

Table 4.5 -- Unlit colour and alpha mapping

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

where:

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

4.14.3 Lighting 'on'

If the shape is lit (i.e., a Material and an Appearance node are specified for the Shape), the Color and Texture nodes determine the diffuse colour for the lighting equation as specified in Table 4.6.

Table 4.6 -- Lit colour and alpha mapping

Texture type	**Colour per-vertex or per-face**	**Color node NULL**
No texture	$O_{Drgb} = I_{Crgb}$ $A = 1-T_M$	$O_{Drgb} = I_{Drgb}$ $A = 1-T_M$
Intensity texture (one-component)	$O_{Drgb} = I_T \times I_{Crgb}$ $A = 1-T_M$	$O_{Drgb} = I_T \times I_{Drgb}$ $A = 1-T_M$
Intensity+Alpha texture (two-component)	$O_{Drgb} = I_T \times I_{Crgb}$ $A = A_T$	$O_{Drgb} = I_T \times I_{Drgb}$ $A = A_T$
RGB texture (three-component)	$O_{Drgb} = I_{Trgb}$ $A = 1-T_M$	$O_{Drgb} = I_{Trgb}$ $A = 1-T_M$
RGBA texture (four-component)	$O_{Drgb} = I_{Trgb}$ $A = A_T$	$O_{Drgb} = I_{Trgb}$ $A = A_T$

where:

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

All other terms are as defined in 4.14.2, Lighting `off'.

4.14.4 Lighting equations

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

$$I_{rgb} = I_{Frgb} \times (1 - f_0) + f_0 \times (O_{Ergb} + \mathrm{SUM}(\ on_i \times attenuation_i \times spot_i \times I_{Lrgb} \times (ambient_i + diffuse_i + specular_i)))$$

where:

attenuation$_i$ = 1 / max($c_1 + c_2 \times d_L + c_3 \times d_L$&sup2; , 1)
ambient$_i$ = $I_{ia} \times O_{Drgb} \times O_a$
diffuse$_i$ = $I_i \times O_{Drgb} \times (\mathbf{N} \bullet \mathbf{L})$
specular $_i$ = $I_i \times O_{Srgb} \times (\mathbf{N} \bullet ((\mathbf{L} + \mathbf{V}) / |\mathbf{L} + \mathbf{V}|))^{\text{shininess} \times 128}$

and:

$\bullet$ = modified vector dot product: if dot product < 0, then 0.0, otherwise, dot product

c_1 , c_2, c_3 = light i attenuation
d_V = distance from point on geometry to viewer's position, in coordinate system of current fog node
d_L = distance from light to point on geometry, in light's coordinate system
f_0 = Fog interpolant, see Table 4.8 for calculation
I_{Frgb} = currently bound fog's color
I_{Lrgb} = light i color

I_i = light i intensity
I_{ia} = light i ambientIntensity
***L** = (Point/SpotLight) normalized vector from point on geometry to light source i position*
***L** = (DirectionalLight) -direction of light source i*
***N** = normalized normal vector at this point on geometry (interpolated from vertex normals specified in Normal node or calculated by browser)*
O_a = Material ambientIntensity
O_{Drgb} = diffuse colour, from Material node, Color node, and/or texture node
O_{Ergb} = Material emissiveColor
O_{Srgb} = Material specularColor
on $_i$ = 1, if light source i affects this point on the geometry,

0, if light source i does not affect this geometry (if farther away than radius *for PointLight or SpotLight, outside of enclosing Group/Transform for DirectionalLights, or* on *field is FALSE)*

shininess = Material shininess
*spotAngle = acos(**-L** $\bullet$ **spotDir**$_i$)*
spot $_{BW}$ = SpotLight i beamWidth
spot $_{CO}$ = SpotLight i cutOffAngle
spot $_i$ = spotlight factor, see Table 4.7 for calculation
***spotDir**$_i$ = normalized SpotLight i* direction
SUM: sum over all light sources i
***V** = normalized vector from point on geometry to viewer's position*

Table 4.7 -- Calculation of the spotlight factor

Condition (in order)	**spot$_i$ =**
light$_i$ is PointLight or DirectionalLight	1
spotAngle >= spot$_{CO}$	0
spotAngle <= spot$_{BW}$	1
spot$_{BW}$ < spotAngle < spot $_{CO}$	(spotAngle - spot$_{CO}$) / (spot$_{BW}$-spot$_{CO}$)

Table 4.8 -- Calculation of the fog interpolant

Condition	**f_0 =**
no fog	1
fogType "LINEAR", d_V < fogVisibility	(fogVisibility-d_V) / fogVisibility
fogType "LINEAR", $d_V \geq$ fogVisibility	0
fogType "EXPONENTIAL", d_V < fogVisibility	exp(-d_V / (fogVisibility-d_V))
fogType "EXPONENTIAL", $d_V \geq$ fogVisibility	0

4.14.5 References

The VRML lighting equations are based on the simple illumination equations given in E.[FOLE] and E.[OPEN].

5 Field and event reference

5.1 Introduction

5.1.1 Table of contents

5.1.2 Description

This clause describes the syntax and general semantics of *fields* and *events*, the elemental data types used by VRML nodes to define objects (see 6, Node reference). Nodes are composed of fields and events (see 4, Concepts). The types defined in this annex are used by both fields and events.

There are two general classes of fields and events: fields and events that contain a single value (where a value may be a single number, a vector, or even an image), and fields and events that contain an ordered list of multiple values. Single-valued fields and events have names that begin with **SF.** Multiple-valued fields and events have names that begin with **MF**.

Multiple-valued fields/events are written as an ordered list of values enclosed in square brackets and separated by whitespace. If the field or event has zero values, only the square brackets ("[ ]") are written. The last value may optionally be followed by whitespace. If the field has exactly one value, the brackets may be omitted. 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 ]
```

5.2 SFBool

The SFBool field or event specifies a single boolean value. SFBools are written as TRUE or FALSE. For example,

```
fooBool FALSE
```

is an SFBool field, *fooBool*, defining a FALSE value.

The initial value of an SFBool eventOut is FALSE.

5.3 SFColor and MFColor

The SFColor field or event specifies one RGB (red-green-blue) colour triple. MFColor specifies zero or more RGB triples. Each colour is written to the VRML file as an RGB triple of floating point numbers in ISO C floating point format (see 2.[ISOC]) 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 colours red, green, and blue.

The initial value of an SFColor eventOut is (0 0 0). The initial value of an MFColor eventOut is [ ].

5.4 SFFloat and MFFloat

The SFFloat field or event specifies one single-precision floating point number. MFFloat specifies zero or more single-precision floating point numbers. SFFloats and MFFloats are written to the VRML file in ISO C floating point format (see 2.[ISOC]). For example:

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

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

The initial value of an SFFloat eventOut is 0.0. The initial value of an MFFloat eventOut is [ ].

5.5 SFImage

The SFImage field or event specifies a single uncompressed 2-dimensional pixel image. SFImage fields and events are written to the VRML file as three integers representing the width, height and number of components in the image, followed by width*height hexadecimal or integer values representing the pixels in the image, separated by whitespace:

```
fooImage <width> <height> <num components> <pixels values>
```

Pixel values are limited to 256 levels of intensity (i.e., 0-255 decimal or 0x00-0xFF hexadecimal). A one-component image specifies one-byte hexadecimal or integer values representing the intensity of the image. For example, `0xFF` is full intensity in hexadecimal (255 in decimal), `0x00` is no intensity (0 in decimal). A two-component image specifies the intensity in the first (high) byte and the alpha opacity in the second (low) byte. Pixels in a three-component image specify the red component in the first (high) byte, followed by the green and blue components (e.g., `0xFF0000` is red, `0x00FF00` is green, `0x0000FF` is blue). Four-component images specify the alpha opacity byte after red/green/blue (e.g., `0x0000FF80` is semi-transparent blue). A value of `0x00` is completely transparent, 0xFF is completely opaque. Note that alpha equals (1.0 - transparency), if alpha and transparency range from 0.0 to 1.0.

Each pixel is read as a single unsigned number. For example, a 3-component pixel with value `0x0000FF` may also be written as `0xFF` (hexadecimal) 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. As another example,

```
fooImage 2 4 3 0xFF0000 0xFF00 0 0 0 0 0xFFFFFF 0xFFFF00
                # red    green  black.. white    yellow
```

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

The initial value of an SFImage eventOut is (0 0 0).

5.6 SFInt32 and MFInt32

The SFInt32 field and event specifies one 32-bit integer. The MFInt32 field and event specifies zero or more 32-bit integers. SFInt32 and MFInt32 fields and events are written to the VRML file as an integer in decimal or hexadecimal (beginning with '0x') format. For example:

```
fooInt32 [ 17, -0xE20, -518820 ]
```

is an MFInt32 field containing three values.

The initial value of an SFInt32 eventOut is 0. The initial value of an MFInt32 eventOut is [ ].

5.7 SFNode and MFNode

The SFNode field and event specifies a VRML node. The MFNode field and event specifies zero or more nodes. The following example illustrates valid syntax for an MFNode field, *fooNode*, defining four nodes:

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

The SFNode field and event may contain the keyword NULL to indicate that it is empty.

The initial value of an SFNode eventOut is NULL. The initial value of an MFNode eventOut is [ ].

5.8 SFRotation and MFRotation

The SFRotation field and event specifies one arbitrary rotation. The MFRotation field and event specifies zero or more arbitrary rotations. An SFRotation is written to the VRML file as four ISO C floating point values (see 2.[ISOC]) separated by whitespace. The first three values specify a normalized rotation axis vector about which the rotation takes place. The fourth value specifies the amount of right-handed rotation about that axis in radians. For example, an SFRotation containing a PI radians rotation about the Y axis is:

```
   fooRot 0.0 1.0 0.0 3.14159265
```

The 3x3 matrix representation of a rotation (x y z a) is

$$\begin{bmatrix} tx^2+c & txy+sz & txz-sy \\ txy-sz & ty^2+c & tyz+sx \\ txz+sy & tyz-sx & tz^2+c \end{bmatrix}$$

```
    where c = cos(a), s = sin(a), and t = 1-c
```

The initial value of an SFRotation eventOut is (0 0 1 0). The initial value of an MFRotation eventOut is [ ].

5.9 SFString and MFString

The SFString and MFString fields and events contain strings formatted with the UTF-8 universal character set (see 2.[UTF8]). SFString specifies a single string. The MFString specifies zero or more strings. Strings are written to the VRML file as a sequence of UTF-8 octets enclosed in double quotes (e.g., **"string"**).

Any characters (including linefeeds and '#') may appear within the quotes. A double quote character within the string is preceded with a backslash. A backslash character within the string is also preceded with a backslash forming two backslashes. For example:

```
   fooString [ "One, Two, Three", "He said, \"Immel did it!\"" ]
```

is an MFString field, *fooString*, with two valid strings.

The initial value of an SFString eventOut is "" (the empty string). The initial value of an MFString eventOut is [ ].

5.10 SFTime and MFTime

The SFTime field or event specifies a single time value. The MFTime field or event specifies zero or more time values. Time values are written to the VRML file as a double-precision floating point number in ISO C floating point format (see 2.[ISOC]). Time values are specified as the number of seconds from a specific time origin. Typically, SFTime fields and events represent the number of seconds since Jan 1, 1970, 00:00:00 GMT. For example:

```
fooTime 0.0
```

is an SFTime field, *fooTime*, representing a time of 0.0 seconds.

The initial value of an SFTime eventOut is -1. The initial value of an MFTime eventOut is [ ].

5.11 SFVec2f and MFVec2f

The SFVec2f field or event specifies a two-dimensional (2D) vector. An MFVec2f field or event specifies zero or more 2D vectors. SFVec2f's and MFVec2f's are written to the VRML file as a pair of ISO C floating point values (see 2.[ISOC]) separated by whitespace. For example:

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

is an MFVec2f field, *fooVec2f*, with two valid vectors.

The initial value of an SFVec2f eventOut is (0 0). The initial value of an MFVec2f eventOut is [ ].

5.12 SFVec3f and MFVec3f

The SFVec3f field or event specifies a three-dimensional (3D) vector. An MFVec3f field or event specifies zero or more 3D vectors. SFVec3f's and MFVec3f's are written to the VRML file as three ISO C floating point values (see 2.[ISOC]) separated by whitespace. For example:

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

is an MFVec3f field, *fooVec3f*, with two valid vectors.

The initial value of an SFVec3f eventOut is (0 0 0). The initial value of an MFVec3f eventOut is [ ].

6 Node reference

6.1 Introduction

This clause provides a detailed definition of the syntax and semantics of each node in this part of ISO/IEC 14772. Table 6.1 lists the topics in this clause.

Table 6.1 -- Table of contents

6.1 Introduction	6.20 FontStyle	6.39 ScalarInterpolator
6.2 Anchor	6.21 Group	6.40 Script
6.3 Appearance	6.22 ImageTexture	6.41 Shape
6.4 AudioClip	6.23 IndexedFaceSet	6.42 Sound
6.5 Background	6.24 IndexedLineSet	6.43 Sphere
6.6 Billboard	6.25 Inline	6.44 SphereSensor
6.7 Box	6.26 LOD	6.45 SpotLight
6.8 Collision	6.27 Material	6.46 Switch
6.9 Color	6.28 MovieTexture	6.47 Text
6.10 ColorInterpolator	6.29 NavigationInfo	6.48 TextureCoordinate
6.11 Cone	6.30 Normal	6.49 TextureTransform
6.12 Coordinate	6.31 NormalInterpolator	6.50 TimeSensor
6.13 CoordinateInterpolator	6.32 OrientationInterpolator	6.51 TouchSensor
6.14 Cylinder	6.33 PixelTexture	6.52 Transform
6.15 CylinderSensor	6.34 PlaneSensor	6.53 Viewpoint
6.16 DirectionalLight	6.35 PointLight	6.54 VisibilitySensor
6.17 ElevationGrid	6.36 PointSet	6.55 WorldInfo
6.18 Extrusion	6.37 PositionInterpolator	
6.19 Fog	6.38 ProximitySensor	

In this clause, the first item in each subclause presents the public declaration for the node. This syntax is not the actual UTF-8 encoding syntax. The parts of the interface that are identical to the UTF-8 encoding syntax are in **bold**. The node declaration defines the names and types of the fields and events for the node, as well as the default values for the fields.

The node declarations also include value ranges for the node's fields and exposedFields (where appropriate). Parentheses imply that the range bound is exclusive, while brackets imply that the range value is inclusive. For example, a range of (-∞, 1] defines the lower bound as -∞ exclusively and the upper bound as 1 inclusively.

For example, the following defines the Collision node declaration:

```
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  # (0,∞) or -1,-1,-1
      field        SFNode   proxy         NULL
      eventOut     SFTime   collideTime
    }
```

The fields and events contained within the node declarations are ordered as follows:

e. eventIns, in alphabetical order;

f. exposedFields, in alphabetical order;

g. fields, in alphabetical order;

h. eventOuts, in alphabetical order.

6.2 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  # (0,∞) or -1,-1,-1
}
```

The Anchor grouping node retrieves the content of a URL when the user activates (e.g., clicks) some geometry contained within the Anchor node's children. If the URL points to a valid VRML file, that world replaces the world of which the Anchor node is a part (except when the *parameter* field, described below, alters this behaviour). If non-VRML data is retrieved, the browser shall determine how to handle that data; typically, it will be passed to an appropriate non-VRML browser.

Exactly how a user activates geometry contained by the Anchor node depends on the pointing device and is determined by the VRML browser. Typically, clicking with the pointing device will result in the new scene replacing the current scene. An Anchor node with an empty *url* does nothing when its children are chosen. A description of how multiple Anchors and pointing-device sensors are resolved on activation is contained in 4.6.7, Sensor nodes.

More details on the *children*, *addChildren*, and *removeChildren* fields and eventIns can be found in 4.6.5, Grouping and children nodes.

The *description* field in the Anchor node specifies a textual description of the Anchor node. This may be used by browser-specific user interfaces that wish to present users with more detailed information about the Anchor.

The *parameter* exposed field may be used to supply any additional information to be interpreted by the browser. Each string shall 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 an HTML document. The *parameter* field is then:

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

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

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

specifies an anchor that loads the VRML file "someScene.wrl" and binds the initial user view to the Viewpoint node named "OverView" when the Anchor node's geometry (Box) is activated. If the named Viewpoint node is not found in the VRML file, the VRML file is loaded using the default Viewpoint node binding stack rules (see 6.53, Viewpoint).

If the *url* field is specified in the form "#ViewpointName" (i.e. no file name), the Viewpoint node with the given name ("ViewpointName") in the Anchor's run-time name scope(s) shall be bound (*set_bind* TRUE). The results are undefined if there are multiple Viewpoints with the same name in the Anchor's run-time name scope(s). The results are undefined if the Anchor node is not part of any run-time name scope or is part of more than one run-time name scope. See 4.4.6, Run-time name scope, for a description of run-time name scopes. See 6.53, Viewpoint, for the Viewpoint transition rules that specify how browsers shall interpret the transition from the old Viewpoint node to the new one. For example:

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

binds the viewer to the viewpoint defined by the "Doorway" viewpoint in the current world when the sphere is activated. In this case, if the Viewpoint is not found, no action occurs on activation.

More details on the *url* field are contained in 4.5, VRML and the World Wide Web.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Anchor's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. The default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. More details on the *bboxCenter* and *bboxSize* fields can be found in 4.6.4, Bounding boxes.

6.3 Appearance

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

The Appearance node specifies the visual properties of geometry. The value for each of the fields in this node may be NULL. However, if the field is non-NULL, it shall contain one node of the appropriate type.

The *material* field, if specified, shall contain a Material node. If the *material* field is NULL or unspecified, lighting is off (all lights are ignored during rendering of the object that references this Appearance) and the unlit object colour is (1, 1, 1). Details of the VRML lighting model are in 4.14, Lighting model.

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

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

6.4 AudioClip

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

An AudioClip node specifies audio data that can be referenced by Sound nodes.

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

The *url* field specifies the URL from which the sound is loaded. Browsers shall support at least the *wavefile* format in uncompressed PCM format (see E.[WAV]). It is recommended that browsers also support the MIDI file type 1 sound format (see 2.[MIDI]); MIDI files are presumed to use the General MIDI patch set. Subclause 4.5, VRML and the World Wide Web, contains details on the *url* field. The results are undefined when no URLs refer to supported data types

The *loop*, *startTime*, and *stopTime* exposedFields and the *isActive* eventOut, and their effects on the AudioClip node, are discussed in detail in 4.6.9, Time-dependent nodes. The "*cycle*" of an AudioClip is the length of time in seconds for one playing of the audio at the specified *pitch*.

The *pitch* field specifies a multiplier for the rate at which sampled sound is played. Values for the *pitch* field shall be greater than zero. Changing the *pitch* field affects both the pitch and playback speed of a sound. A *set_pitch* event to an active AudioClip is ignored and no *pitch_changed* eventOut is generated. If *pitch* is set to 2.0, the sound shall be played one octave higher than normal and played twice as fast. For a sampled sound, the *pitch* field alters the sampling rate at which the sound is played. The proper implementation of pitch control for MIDI (or other note sequence sound clips) 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.

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

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

```
    t = (now - startTime) modulo (duration / pitch)
```

6.5 Background

```
Background {
  eventIn      SFBool   set_bind
  exposedField MFFloat  groundAngle  []          # [0,π/2]
  exposedField MFColor  groundColor  []          # [0,1]
  exposedField MFString backUrl      []
  exposedField MFString bottomUrl    []
  exposedField MFString frontUrl     []
  exposedField MFString leftUrl      []
  exposedField MFString rightUrl     []
  exposedField MFString topUrl       []
  exposedField MFFloat  skyAngle     []          # [0,π]
  exposedField MFColor  skyColor     0 0 0       # [0,1]
  eventOut     SFBool   isBound
}
```

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

Background nodes are bindable nodes as described in 4.6.10, Bindable children nodes. There exists a Background stack, in which the top-most Background on the stack is the currently active Background. To move a Background to the top of the stack, a TRUE value is sent to the *set_bind* eventIn. Once active, the Background is then bound to the browsers view. A FALSE value sent to *set_bind* removes the Background from the stack and unbinds it from the browser's view. More detail on the bind stack is described in 4.6.10, Bindable children nodes.

The backdrop is conceptually a partial sphere (the ground) enclosed inside of a full sphere (the sky) in the local coordinate system with the viewer placed at the centre of the spheres. Both spheres have infinite radius and each is painted with concentric circles of interpolated colour perpendicular to the local Y-axis of the sphere. The Background node is subject to the accumulated rotations of its ancestors' transformations. Scaling and translation transformations are ignored. The sky sphere is always slightly farther away from the viewer than the ground partial sphere causing the ground to appear in front of the sky where they overlap.

The *skyColor* field specifies the colour of the sky at various angles on the sky sphere. The first value of the *skyColor* field specifies the colour of the sky at 0.0 radians representing the zenith (i.e., straight up from the viewer). The *skyAngle* field specifies the angles from the zenith in which concentric circles of colour appear. The zenith of the sphere is implicitly defined to be 0.0 radians, the natural horizon is at π/2 radians, and the nadir (i.e., straight down from the viewer) is at π radians. *skyAngle* is restricted to non-decreasing values in the range [0.0,π]. There shall be one more *skyColor* value than there are *skyAngle* values. The first colour value is the colour at the zenith, which is not specified in the *skyAngle* field. If the last *skyAngle* is less than *pi*, then the colour band between the last *skyAngle* and the nadir is clamped to the last *skyColor*. The sky colour is linearly interpolated between the specified *skyColor* values.

The *groundColor* field specifies the colour of the ground at the various angles on the ground partial sphere. The first value of the *groundColor* field specifies the colour of the ground at 0.0 radians representing the nadir (i.e., straight down from the user). The *groundAngle* field specifies the angles from the nadir that the concentric circles of colour appear. The nadir of the sphere is implicitly defined at 0.0 radians. *groundAngle* is restricted to non-decreasing values in the range [0.0, π/2]. There shall be one more *groundColor* value than there are *groundAngle* values. The first colour value is for the nadir which is not specified in the *groundAngle* field. If the last *groundAngle* is less than π/2, the region between the last *groundAngle* and the equator is non-existant. The ground colour is linearly interpolated between the specified *groundColor* values.

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

Figure 6.1 illustrates the Background node backdrop and background textures.

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.

See 4.6.11, Texture maps, for a general description of texture maps.

Often, the *bottomUrl* and *topUrl* images will not be specified, to allow sky and ground to show. The other four images may depict surrounding mountains or other distant scenery. Browsers shall support the JPEG (see 2.[JPEG]) and PNG (see 2.[PNG]) image file formats, and in addition, may support any other image format (e.g., CGM) that can be rendered into a 2D image. Support for the GIF (see E.[GIF]) format is recommended (including transparency) . More detail on the *url* fields can be found in 4.5, VRML and the World Wide Web.

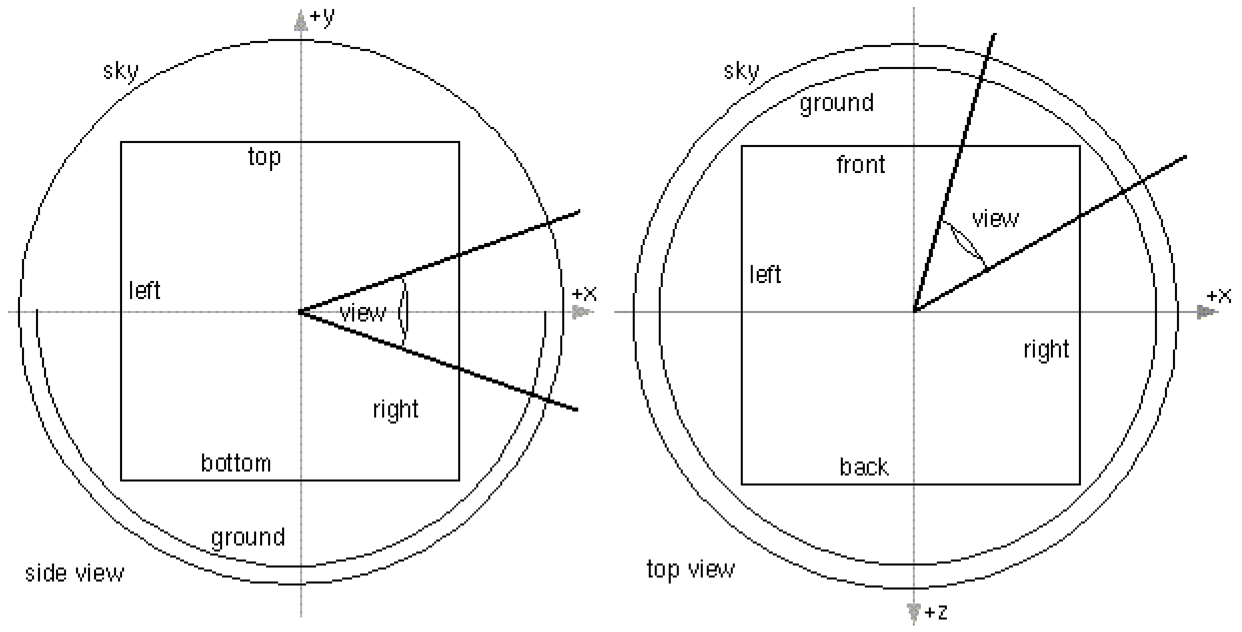


Figure 6.1 -- Background node

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

Ground colours, sky colours, 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 nodes are not affected by Fog nodes. Therefore, if a Background node is active (i.e., bound) while a Fog node is active, then the Background node will be displayed with no fogging effects. It is the author's responsibility to set the Background values to match the Fog values (e.g., ground colours fade to fog colour with distance and panorama images tinted with fog colour). Background nodes are not affected by light sources.

6.6 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  # (0,∞) or -1,-1,-1
}
```

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

The *axisOfRotation* field specifies which axis to use to perform the rotation. This axis is defined in the local coordinate system.

When the *axisOfRotation* field is not (0, 0, 0), the following steps describe how to rotate the billboard to face the viewer:

a. Compute the vector from the Billboard node's origin to the viewer's position. This vector is called the *billboard-to-viewer* vector.

b. Compute the plane defined by the *axisOfRotation* and the billboard-to-viewer vector.

c. Rotate the local Z-axis of the billboard into the plane from b., pivoting around the *axisOfRotation*.

When the axisOfRotation field is set to (0, 0, 0), the special case of *viewer-alignment* is indicated. In this case, the object rotates to keep the billboard's local Y-axis parallel with the Y-axis of the viewer. This special case is distinguished by setting the *axisOfRotation* to (0, 0, 0). The following steps describe how to align the billboard's Y-axis to the Y-axis of the viewer:

d. Compute the billboard-to-viewer vector.

e. Rotate the Z-axis of the billboard to be collinear with the billboard-to-viewer vector and pointing towards the viewer's position.

f. Rotate the Y-axis of the billboard to be parallel and oriented in the same direction as the Y-axis of the viewer.

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

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

Subclause 4.6.5, Grouping and children nodes, provides a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Billboard node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. A description of the *bboxCenter* and *bboxSize* fields is contained in 4.6.4, Bounding boxes.

6.7 Box

```
Box {
  field    SFVec3f size  2 2 2        # (0,∞)
}
```

The Box node specifies a rectangular parallelepiped box centred at (0, 0, 0) in the local coordinate system and aligned with the local coordinate axes. By default, the box measures 2 units in each dimension, from -1 to +1. The *size* field specifies the extents of the box along the X-, Y-, and Z-axes respectively and each component value shall be greater than zero. Figure 6.2 illustrates the Box node.

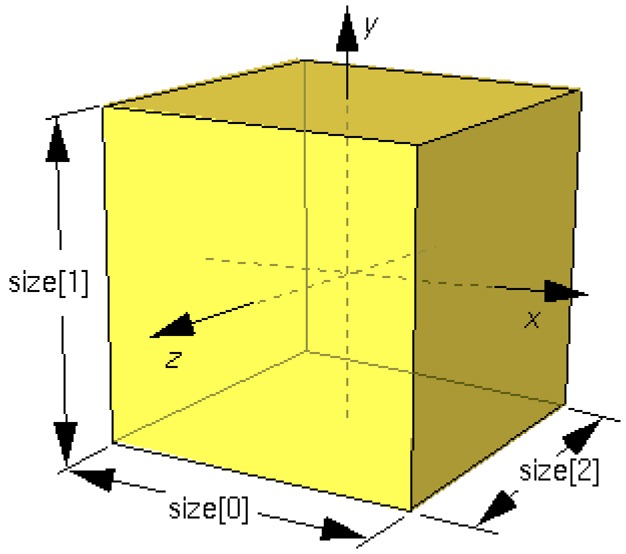


Figure 6.2 -- Box node

Textures are applied individually to each face of the box. On the front (+Z), back (-Z), right (+X), and left (-X) faces of the box, when viewed from the outside with the +Y-axis up, the texture is mapped onto each face with the same orientation as if the image were displayed normally in 2D. On the top face of the box (+Y), when viewed from above and looking down the Y-axis toward the origin with the -Z-axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. On the bottom face of the box (-Y), when viewed from below looking up the Y-axis toward the origin with the +Z-axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. TextureTransform affects the texture coordinates of the Box.

The Box node's geometry requires outside faces only. When viewed from the inside the results are undefined.

6.8 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   # (0,∞) or -1,-1,-1
  field        SFNode   proxy           NULL
  eventOut     SFTime   collideTime
}
```

The Collision node is a grouping node that specifies the collision detection properties for its children (and their descendants), specifies surrogate objects that replace its children during collision detection, and sends events signalling that a collision has occurred between the avatar and the Collision node's geometry or surrogate. By default, all geometric nodes in the scene are collidable with the viewer except IndexedLineSet, PointSet, and Text. Browsers shall detect geometric collisions between the avatar (see 6.29, NavigationInfo) and the scene's geometry and prevent the avatar from 'entering' the geometry. See 4.13.4, Collision detection and terrain following, for general information on collision detection.

If there are no Collision nodes specified in a VRML file, browsers shall detect collisions between the avatar and all objects during navigation.

Subclause 4.6.5, Grouping and children nodes, contains a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

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

Collision nodes with the *collide* field set to TRUE detect the nearest collision with their descendent geometry (or proxies). When the nearest collision is detected, the collided Collision node sends the time of the collision through its *collideTime* eventOut. If a Collision node contains a child, descendant, or proxy (see below) that is a Collision node, and both Collision nodes detect that a collision has occurred, both send a *collideTime* event at the same time. A *collideTime* event shall be generated if the avatar is colliding with collidable geometry when the Collision node is read from a VRML file or inserted into the transformation hierarchy.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Collision node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. More details on the *bboxCenter* and *bboxSize* fields can be found in 4.6.4, Bounding boxes.

The collision proxy, defined in the *proxy* field, is any legal children node as described in 4.6.5, Grouping and children nodes, that is used as a substitute for the Collision node's children during collision detection. The proxy is used strictly for collision detection; it is not drawn.

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

If *proxy* is specified, any descendent children of the Collision node are ignored during collision detection. If *children* is empty, *collide* is TRUE, and *proxy* is specified, collision detection is performed against the proxy but nothing is displayed. In this manner, invisible collision objects may be supported.

The *collideTime* eventOut generates an event specifying the time when the avatar (see 6.29, NavigationInfo) makes contact with the collidable children or proxy of the Collision node. An ideal implementation computes the exact time of collision. Implementations may approximate the ideal by sampling the positions of collidable objects and the user. The NavigationInfo node contains additional information for parameters that control the avatar size.

6.9 Color

```
Color {
  exposedField MFColor color   []         # [0,1]
}
```

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

Color nodes are only used to specify multiple colours for a single geometric shape, such as colours for the faces or vertices of an IndexedFaceSet. A Material node is used to specify the overall material parameters of lit geometry. If both a Material node and a Color node are specified for a geometric shape, the colours shall replace the diffuse component of the material.

RGB or RGBA textures take precedence over colours; specifying both an RGB or RGBA texture and a Color node for geometric shape will result in the Color node being ignored. Details on lighting equations can be found in 4.14, Lighting model.

6.10 ColorInterpolator

```
ColorInterpolator {
  eventIn      SFFloat set_fraction         # (-∞,∞)
  exposedField MFFloat key           []     # (-∞,∞)
  exposedField MFColor keyValue      []     # [0,1]
  eventOut     SFColor value_changed
}
```

This node interpolates among a list of MFColor key values to produce an SFColor (RGB) *value_changed* event. The number of colours in the *keyValue* field shall be equal to the number of keyframes in the *key* field. The *keyValue* field and *value_changed* events are defined in RGB colour space. A linear interpolation using the value of *set_fraction* as input is performed in HSV space (see E.[FOLE] for description of RGB and HSV colour spaces). The results are undefined when interpolating between two consecutive keys with complementary hues.

4.6.8, Interpolator nodes, contains a detailed discussion of interpolators.

6.11 Cone

```
Cone {
  field    SFFloat   bottomRadius 1        # (0,∞)
  field    SFFloat   height       2        # (0,∞)
  field    SFBool    side         TRUE
  field    SFBool    bottom       TRUE
}
```

The Cone node specifies a cone which is centred in the local coordinate system and whose central axis is aligned with the local Y-axis. The *bottomRadius* field specifies the radius of the cone's base, and the *height* field specifies the height of the cone from the centre of the base to the apex. By default, the cone has a radius of 1.0 at the bottom and a height of 2.0, with its apex at y = *height*/2 and its bottom at y = -*height*/2. Both *bottomRadius* and *height* shall be greater than zero. Figure 6.3 illustrates the Cone node.

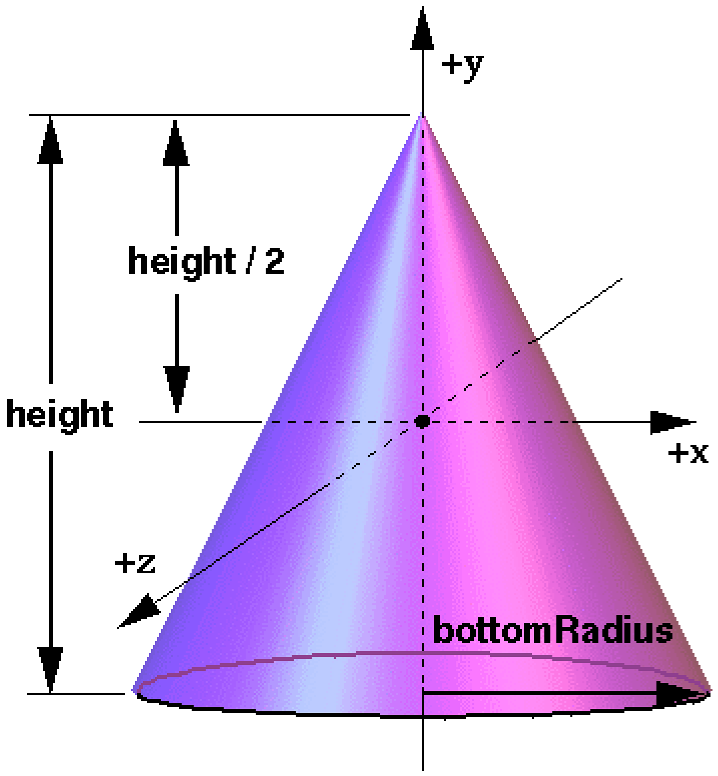


Figure 6.3 -- Cone node

The *side* field specifies whether sides of the cone are created and the *bottom* field specifies whether the bottom cap of the cone is created. A value of TRUE specifies that this part of the cone exists, while a value of FALSE specifies that this part does not exist (not rendered or eligible for collision or sensor intersection tests).

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

The Cone geometry requires outside faces only. When viewed from the inside the results are undefined.

6.12 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 including IndexedFaceSet, IndexedLineSet, and PointSet.

6.13 CoordinateInterpolator

```
CoordinateInterpolator {
  eventIn      SFFloat set_fraction        # (-∞,∞)
  exposedField MFFloat key            []   # (-∞,∞)
  exposedField MFVec3f keyValue       []   # (-∞,∞)
  eventOut     MFVec3f value_changed
}
```

This node linearly interpolates among a list of MFVec3f values. The number of coordinates in the *keyValue* field shall 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.

4.6.8, Interpolator nodes, contains a more detailed discussion of interpolators.

6.14 Cylinder

```
Cylinder {
  field    SFBool    bottom  TRUE
  field    SFFloat   height  2         # (0,∞)
  field    SFFloat   radius  1         # (0,∞)
  field    SFBool    side    TRUE
  field    SFBool    top     TRUE
}
```

The Cylinder node specifies a capped cylinder centred at (0,0,0) in the local coordinate system and with a central axis oriented along the local Y-axis. By default, the cylinder is sized at "-1" to "+1" in all three dimensions. The *radius* field specifies the radius of the cylinder and the *height* field specifies the height of the cylinder along the central axis. Both *radius* and *height* shall be greater than zero. Figure 6.4 illustrates the Cylinder node.

The cylinder has three *parts*: the *side*, the *top* (Y = +height/2) and the *bottom* (Y = -height/2). Each part has an associated SFBool field that indicates whether the part exists (TRUE) or does not exist (FALSE). Parts which do not exist are not rendered and not eligible for intersection tests (e.g., collision detection or sensor activation).

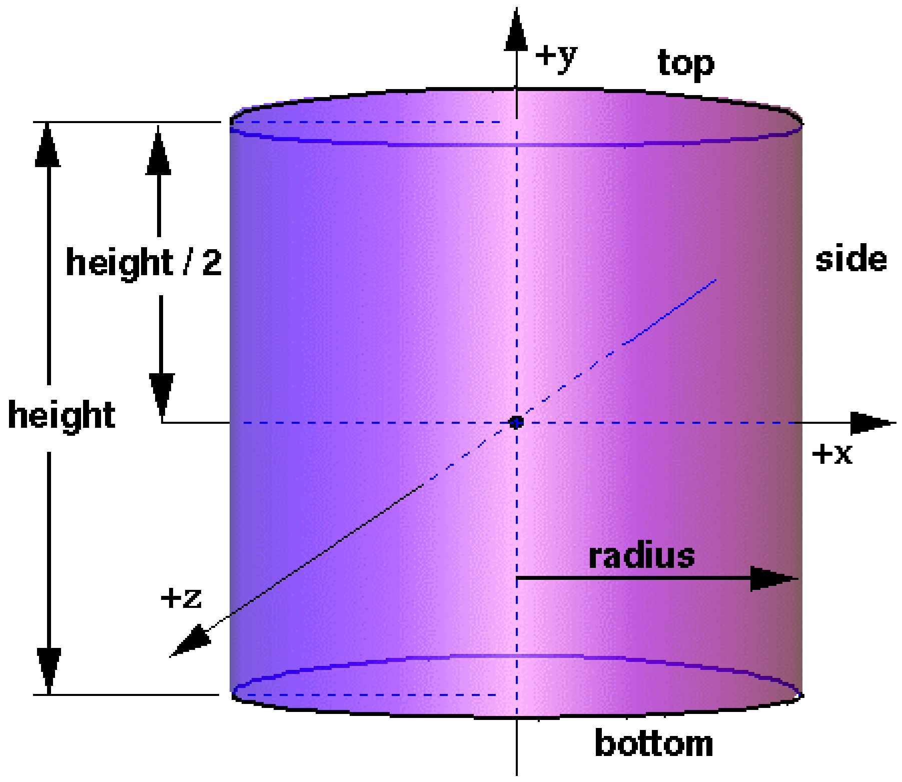


Figure 6.4 -- Cylinder node

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 X=0 plane. For the top and bottom caps, a circle is cut out of the unit texture squares centred at (0, +/- *height/2*, 0) with dimensions 2 × *radius* by 2 × *radius*. The top texture appears right side up when the top of the cylinder is tilted toward the +Z-axis, and the bottom texture appears right side up when the top of the cylinder is tilted toward the -Z-axis. TextureTransform affects the texture coordinates of the Cylinder node.

The Cylinder node's geometry requires outside faces only. When viewed from the inside the results are undefined.

6.15 CylinderSensor

```
CylinderSensor {
  exposedField SFBool     autoOffset TRUE
  exposedField SFFloat    diskAngle  0.262       # (0,π/2)
  exposedField SFBool     enabled    TRUE
  exposedField SFFloat    maxAngle   -1          # [-2π,2π]
  exposedField SFFloat    minAngle   0           # [-2π,2π]
  exposedField SFFloat    offset     0           # (-∞,∞)
  eventOut     SFBool     isActive
  eventOut     SFRotation rotation_changed
  eventOut     SFVec3f    trackPoint_changed
}
```

The CylinderSensor node maps pointer motion (e.g., a mouse or wand) into a rotation on an invisible cylinder that is aligned with the Y-axis of the local coordinate system. The CylinderSensor uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposed field enables and disables the CylinderSensor node. If TRUE, the sensor reacts appropriately to user events. If FALSE, the sensor does not track user input or send 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.

A CylinderSensor node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for more details on using the pointing device to activate the CylinderSensor.

Upon activation of the pointing device while indicating the sensor's geometry, an *isActive* TRUE event is sent. The initial acute angle between the bearing vector and the local Y-axis of the CylinderSensor node determines whether the sides of the invisible cylinder or the caps (disks) are used for manipulation. If the initial angle is less than the *diskAngle*, the geometry is treated as an infinitely large disk lying in the local Y=0 plane and coincident with the initial intersection point. Dragging motion is mapped into a rotation around the local +Y-axis vector of the sensor's coordinate system. The perpendicular vector from the initial intersection point to the Y-axis defines zero rotation about the Y-axis. For each subsequent position of the bearing, a *rotation_changed* event is sent that equals the sum of the rotation about the +Y-axis vector (from the initial intersection to the new intersection) 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 value of *rotation_changed* and an *offset_changed* event is generated. See 4.6.7.4, Drag sensors, for a more general description of *autoOffset* and *offset* fields.

If the initial acute angle between the bearing vector and the local Y-axis of the CylinderSensor node is greater than or equal to *diskAngle*, then the sensor behaves like a cylinder. 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 marks the zero rotation value. For each subsequent position of the bearing, a *rotation_changed* event is sent that equals the sum of the right-handed rotation from the original intersection about the +Y-axis vector plus the *offset* value. *trackPoint_changed* events reflect the unclamped drag position on the surface of the invisible cylinder. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last rotation angle and an *offset_changed* event is generated. More details are available in 4.6.7.4, Drag sensors.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is released and generates an *isActive* FALSE event (other pointing-device sensors shall 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 it is released). If a 3D pointing device (e.g., a wand) is in use, *isActive* events will typically reflect whether the pointer is within or in contact with the sensor's geometry.

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

The *minAngle* and *maxAngle* fields clamp *rotation_changed* events to a range of values. If *minAngle* is greater than *maxAngle*, *rotation_changed* events are not clamped. The *minAngle* and *maxAngle* fields are restricted to the range [-2π, 2π].

More information about this behaviour is described in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

6.16 DirectionalLight

```
DirectionalLight {
  exposedField SFFloat ambientIntensity  0        # [0,1]
  exposedField SFColor color             1 1 1    # [0,1]
  exposedField SFVec3f direction         0 0 -1   # (-∞,∞)
  exposedField SFFloat intensity         1        # [0,1]
  exposedField SFBool  on                TRUE
}
```

The DirectionalLight node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector. A description of the *ambientIntensity*, *color*, *intensity*, and *on* fields is in 4.6.6, Light sources.

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

DirectionalLight nodes do not attenuate with distance. A precise description of VRML's lighting equations is contained in 4.14, Lighting model.

6.17 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       # [0,∞)
  field        SFBool   normalPerVertex     TRUE
  field        SFBool   solid               TRUE
  field        SFInt32  xDimension          0       # [0,∞)
  field        SFFloat  xSpacing            1.0     # [0,∞)
  field        SFInt32  zDimension          0       # [0,∞)
  field        SFFloat  zSpacing            1.0     # [0,∞)
}
```

The ElevationGrid node specifies a uniform rectangular grid of varying height in the Y=0 plane of the local coordinate system. The geometry is described by a scalar array of height values that specify the height of a surface above each point of the grid.

The *xDimension* and *zDimension* fields indicate the number of elements of the grid *height* array in the X and Z directions. Both *xDimension* and *zDimension* shall be greater than or equal to zero. If either the *xDimension* or the *zDimension* is less than two, the ElevationGrid contains no quadrilaterals. 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 *xSpacing* and *zSpacing* fields indicate the distance between vertices in the X and Z directions respectively, and shall be greater than zero.

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 × xDimension]
    P[i,j].z = zSpacing × j

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and P[0,0] is height[0] units above/below the origin of the local
    coordinate system
```

The *set_height* eventIn allows the height MFFloat field to be changed to support animated ElevationGrid nodes.

The *color* field specifies per-vertex or per-quadrilateral colours for the ElevationGrid node depending on the value of *colorPerVertex*. If the *color* field is NULL, the ElevationGrid node is rendered with the overall attributes of the Shape node enclosing the ElevationGrid node (see 4.14, Lighting model).

The *colorPerVertex* field determines whether colours specified in the *color* field are applied to each vertex or each quadrilateral of the ElevationGrid node. If *colorPerVertex* is FALSE and the *color* field is not NULL, the *color* field shall specify a Color node containing at least (*xDimension-1)* × (*zDimension-1)* colours; one for each quadrilateral, ordered as follows:

```
    QuadColor[i,j] = Color[ i + j × (xDimension-1)]

    where 0 <= i < xDimension-1 and 0 <= j < zDimension-1,
    and QuadColor[i,j] is the colour for the quadrilateral defined
        by height[i+j × xDimension], height[(i+1)+j × xDimension],
        height[(i+1)+(j+1)× xDimension] and height[i+(j+1)× xDimension]
```

If *colorPerVertex* is TRUE and the *color* field is not NULL, the *color* field shall specify a Color node containing at least *xDimension* × *zDimension* colours, one for each vertex, ordered as follows:

```
    VertexColor[i,j] = Color[ i + j × xDimension]

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and VertexColor[i,j] is the colour for the vertex defined by
        height[i+j × xDimension]
```

The *normal* field specifies per-vertex or per-quadrilateral normals for the ElevationGrid node. If the *normal* field is NULL, the browser shall automatically generate normals, using the *creaseAngle* field to determine if and how normals are smoothed across the surface (see 4.6.3.5, Crease angle field).

The *normalPerVertex* field determines whether normals are applied to each vertex or each quadrilateral of the ElevationGrid node depending on the value of *normalPerVertex*. If *normalPerVertex* is FALSE and the *normal* node is not NULL, the *normal* field shall specify a Normal node containing at least (*xDimension-1)* × (*zDimension-1)* normals; one for each quadrilateral, ordered as follows:

```
    QuadNormal[i,j] = Normal[ i + j × (xDimension-1)]

    where 0 <= i < xDimension-1 and 0 <= j < zDimension-1,
    and QuadNormal[i,j] is the normal for the quadrilateral defined
        by height[i+j × xDimension], height[(i+1)+j × xDimension],
        height[(i+1)+(j+1)× xDimension] and height[i+(j+1)× xDimension]
```

If *normalPerVertex* is TRUE and the *normal* field is not NULL, the *normal* field shall specify a Normal node containing at least *xDimension* × *zDimension* normals; one for each vertex, ordered as follows:

```
    VertexNormal[i,j] = Normal[ i + j × xDimension]

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and VertexNormal[i,j] is the normal for the vertex defined
        by height[i+j×xDimension]
```

The *texCoord* field specifies per-vertex texture coordinates for the ElevationGrid node. If *texCoord* is NULL, default texture coordinates are applied to the geometry. The default texture coordinates range from (0,0) at the first vertex to (1,1) at the last vertex. The S texture coordinate is aligned with the positive X-axis, and the T texture coordinate with positive Z-axis. If *texCoord* is not NULL, it shall specify a TextureCoordinate node containing at least (*xDimension)* × (*zDimension)* texture coordinates; one for each vertex, ordered as follows:

```
    VertexTexCoord[i,j] = TextureCoordinate[ i + j × xDimension]

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and VertexTexCoord[i,j] is the texture coordinate for the vertex
        defined by height[i+j × xDimension]
```

The *ccw*, *solid*, and *creaseAngle* fields are described in 4.6.3, Shapes and geometry.

By default, the quadrilaterals are defined with a counterclockwise ordering. Hence, the Y-component of the normal is positive. Setting the *ccw* field to FALSE reverses the normal direction. Backface culling is enabled when the *solid* field is TRUE.

See Figure 6.5 for a depiction of the ElevationGrid node.

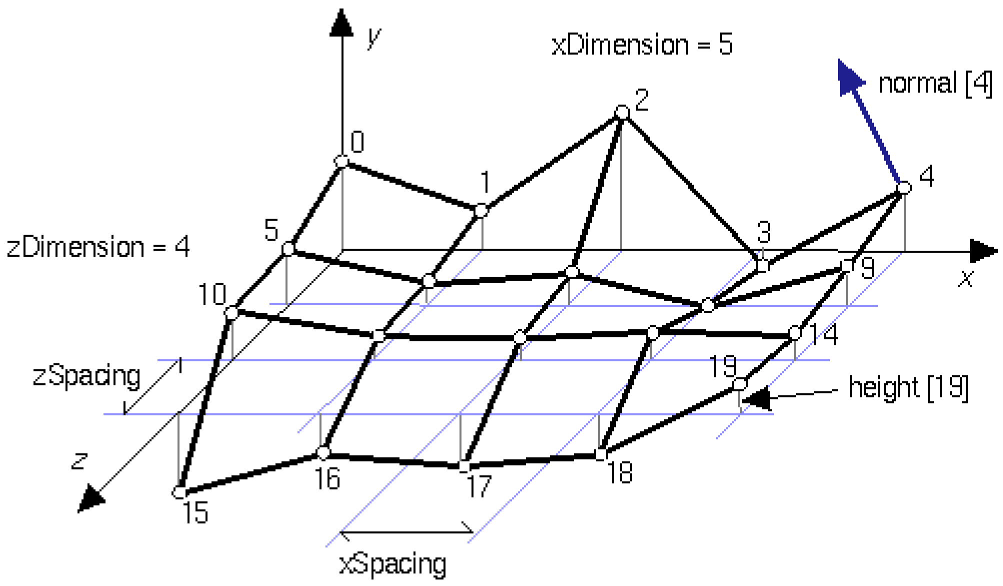


Figure 6.5 -- ElevationGrid node

6.18 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                  # [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            # [-1,1],(- ∞,∞)
  field   MFVec2f     scale             1 1                # (0,∞)
  field   SFBool      solid             TRUE
  field   MFVec3f     spine             [ 0 0 0, 0 1 0 ] # (-∞,∞)
}
```

6.18.1 Introduction

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

An Extrusion node is defined by:

a. a 2D *crossSection* piecewise linear curve (described as a series of connected vertices);

b. a 3D *spine* piecewise linear curve (also described as a series of connected vertices);

c. a list of 2D *scale* parameters;

d. a list of 3D *orientation* parameters.

6.18.2 Algorithmic description

Shapes are constructed as follows. The cross-section curve, which starts as a curve in the Y=0 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 translated by the first spine point and oriented using the first *orientation* parameter (as explained later). The same procedure is followed to place a cross-section at the second spine point, using the second scale and orientation values. Corresponding vertices of the first and second cross-sections are then connected, forming a quadrilateral polygon between each pair of vertices. This same procedure is then repeated for the rest of the spine points, resulting in a surface extrusion along the spine.

The final orientation of each cross-section is computed by first orienting it relative to the spine segments on either side of point at which the cross-section is placed. This is known as the *spine-aligned cross-section plane* (SCP), and is designed to provide a smooth transition from one spine segment to the next (see Figure 6.6). The SCP is then rotated by the corresponding *orientation* value. This rotation is performed relative to the SCP. For example, to impart twist in the cross-section, a rotation about the Y-axis (0 1 0) would be used. Other orientations are valid and rotate the cross-section out of the SCP.

The SCP is computed by first computing its Y-axis and Z-axis, then taking the cross product of these to determine the X-axis. These three axes are then used to determine the rotation value needed to rotate the Y=0 plane to the SCP. This results in a plane that is the approximate tangent of the spine at each point, as shown in Figure 6.6. First the Y-axis is determined, as follows:

Let n be the number of spines and let i be the index variable satisfying 0 <= i < n:

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

b. *If the spine curve is closed:* The SCP for the first and last points is the same and is found using (*spine*[1] - *spine*[n-2]) to compute the Y-axis.

c. *If the spine curve is not closed:* The Y-axis used for the first point is the vector from *spine*[0] to *spine*[1], and for the last it is the vector from *spine*[*n*-2] to *spine*[*n*-1].

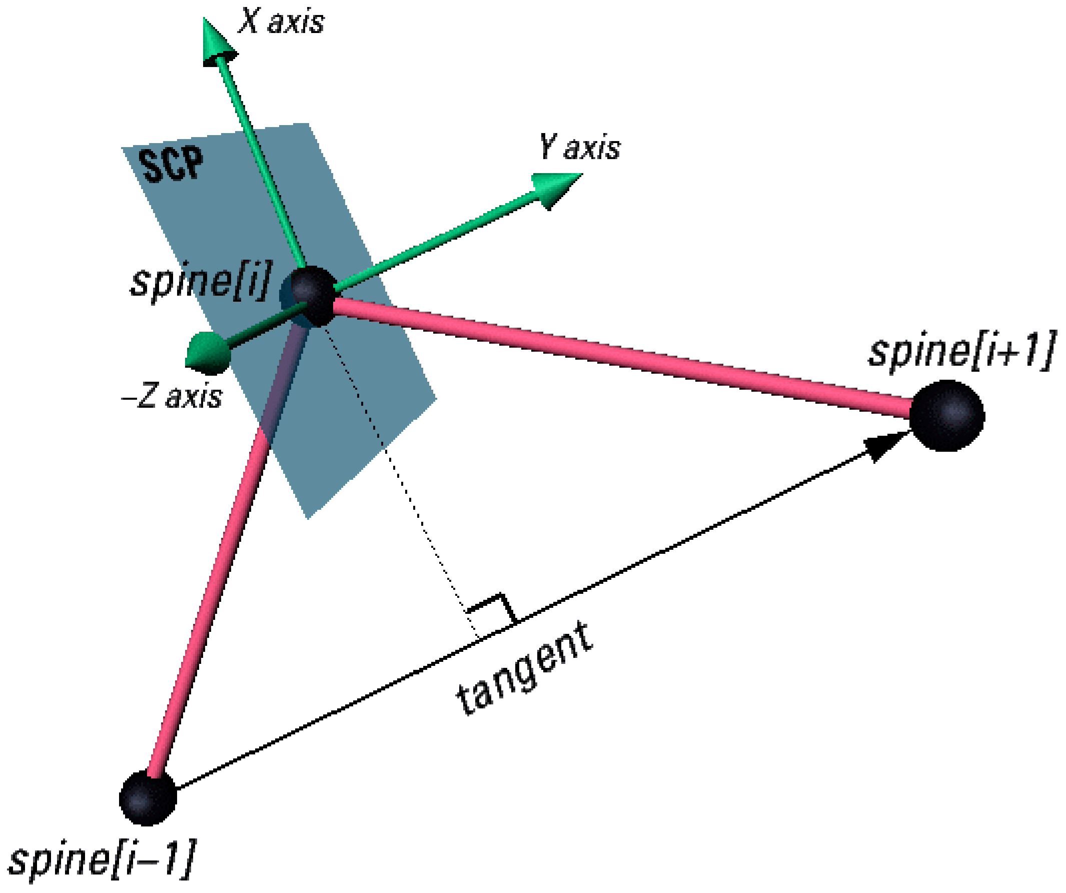


Figure 6.6 -- Spine-aligned cross-section plane at a spine point.

The Z-axis is determined as follows:

d. *For all points other than the first or last:* Take the following cross-product:

   `Z = (spine[i+1] - spine[i]) × (spine[i-1] - spine[i])`

e. *If the spine curve is closed:* The SCP for the first and last points is the same and is found by taking the following cross-product:

   `Z = (spine[1] - spine[0]) × (spine[n-2] - spine[0])`

f. *If the spine curve is not closed:* The Z-axis used for the first spine point is the same as the Z-axis for spine[1]. The Z-axis used for the last spine point is the same as the Z-axis for spine[n-2].

g. After determining the Z-axis, its dot product with the Z-axis of the previous spine point is computed. If this value is negative, the Z-axis is flipped (multiplied by -1). In most cases, this prevents small changes in the spine segment angles from flipping the cross-section 180 degrees.

Once the Y- and Z-axes have been computed, the X-axis can be calculated as their cross-product.

6.18.3 Special cases

If the number of *scale* or *orientation* values is greater than the number of spine points, the excess values are ignored. If they contain one value, it is applied at all spine points. The results are undefined if the number of scale or orientation values is greater than one but less than the number of spine points. The *scale* values shall be positive.

If the three points used in computing the Z-axis are collinear, the cross-product is zero so the value from the previous point is used instead.

If the Z-axis of the first point is undefined (because the spine is not closed and the first two spine segments are collinear) then the Z-axis for the first spine point with a defined Z-axis is used.

If the entire spine is collinear, the SCP is computed by finding the rotation of a vector along the positive Y-axis (`v1`) to the vector formed by the spine points (`v2`). The Y=0 plane is then rotated by this value.

If two points are coincident, they both have the same SCP. If each point has a different orientation value, then the surface is constructed by connecting edges of the cross-sections as normal. This is useful in creating revolved surfaces.

Note: combining coincident and non-coincident spine segments, as well as other combinations, can lead to interpenetrating surfaces which the extrusion algorithm makes no attempt to avoid.

6.18.4 Common cases

The following common cases are among the effects which are supported by the Extrusion node:

Surfaces of revolution:

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

Uniform extrusions:

> *If the* scale *is (1, 1) and the spine is straight, the cross-section is extruded uniformly without twisting or scaling along the spine. The result is a cylindrical shape with a uniform cross section.*

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 (given as rotations about the Y-axis) twist it around the spine, and the scale parameters taper it (by scaling about the spine).*

6.18.5 Other fields

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 each 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* is not a closed curve, the caps are generated by adding a final point to *crossSection* that is equal to the initial point. An open surface can still have a cap, resulting (for a simple case) in a shape analogous to 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.

Texture coordinates are automatically generated by Extrusion nodes. Textures are mapped so that the coordinates range in the U direction from 0 to 1 along the *crossSection* curve (with 0 corresponding to the first point in *crossSection* and 1 to the last) and in the V direction from 0 to 1 along the *spine* curve (with 0 corresponding to the first listed *spine* point and 1 to the last). If either the *endCap* or *beginCap* exists, the *crossSection* curve is uniformly scaled and translated so that the larger dimension of the cross-section (X or Z) produces texture coordinates that range from 0.0 to 1.0. The *beginCap* and *endCap* textures' S and T directions correspond to the X and Z directions in which the *crossSection* coordinates are defined.

The browser shall automatically generate normals for the Extrusion node,using the *creaseAngle* field to determine if and how normals are smoothed across the surface. Normals for the caps are generated along the Y-axis of the SCP, with the ordering determined by viewing the cross-section from above (looking along the negative Y-axis of the SCP). By default, a *beginCap* with a counterclockwise ordering shall have a normal along the negative Y-axis. An *endCap* with a counterclockwise ordering shall have a normal along the positive Y-axis.

Each quadrilateral making up the sides of the extrusion are ordered from the bottom cross-section (the one at the earlier spine point) to the top. So, one quadrilateral has the points:

```
    spine[0](crossSection[0], crossSection[1])
    spine[1](crossSection[1], crossSection[0])
```

in that order. By default, normals for the sides are generated as described in 4.6.3, Shapes and geometry.

For instance, a circular crossSection with counter-clockwise ordering and the default spine form a cylinder. With *solid* TRUE and *ccw* TRUE, the cylinder is visible from the outside. Changing *ccw* to FALSE makes it visible from the inside.

The *ccw*, *solid*, *convex*, and *creaseAngle* fields are described in 4.6.3, Shapes and geometry.

6.19 Fog

```
Fog {
  exposedField SFColor  color           1 1 1     # [0,1]
  exposedField SFString fogType         "LINEAR"
  exposedField SFFloat  visibilityRange 0         # [0,∞)
  eventIn      SFBool   set_bind
  eventOut     SFBool   isBound
}
```

The Fog node provides a way to simulate atmospheric effects by blending objects with the colour specified by the *color* field based on the distances of the various objects from the viewer. The distances are calculated in the coordinate space of the Fog node. The *visibilityRange* specifies the distance in metres (in the local coordinate system) at which objects are totally obscured by the fog. Objects located outside the *visibilityRange* from the viewer are drawn with a constant colour of *color*. Objects very close to the viewer are blended very little with the fog *color*. A *visibilityRange* of 0.0 disables the Fog node. The *visibilityRange* is affected by the scaling transformations of the Fog node's parents; translations and rotations have no affect on *visibilityRange*. Values of the *visibilityRange* field shall be in the range [0,∞).

Since Fog nodes are bindable children nodes (see 4.6.10, Bindable children nodes), a Fog node stack exists, 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 node is bound to the browser view. A FALSE value sent to *set_bind*, pops the Fog node from the stack and unbinds it from the browser viewer. More details on the Fog node stack can be found in 4.6.10, Bindable children nodes.

The *fogType* field controls how much of the fog colour is blended with the object as a function of distance. If *fogType* is "LINEAR", the amount of blending is a linear function of the distance, resulting in a depth cueing effect. If *fogType* is "EXPONENTIAL," an exponential increase in blending is used, resulting in a more natural fog appearance.

The effect of fog on lighting calculations is described in 4.14, Lighting model.

6.20 FontStyle

```
FontStyle {
  field MFString family       "SERIF"
  field SFBool   horizontal   TRUE
  field MFString justify      "BEGIN"
  field SFString language     ""
  field SFBool   leftToRight  TRUE
  field SFFloat  size         1.0          # (0,∞)
  field SFFloat  spacing      1.0          # [0,∞)
  field SFString style        "PLAIN"
  field SFBool   topToBottom  TRUE
}
```

6.20.1 Introduction

The FontStyle node defines the size, family, and style used for Text nodes, as well as the direction of the text strings and any language-specific rendering techniques used for non-English text. See 6.47, Text, for a description of the Text node.

The *size* field specifies the nominal height, in the local coordinate system of the Text node, of glyphs rendered and determines the spacing of adjacent lines of text. Values of the *size* field shall be greater than zero.

The *spacing* field determines the line spacing between adjacent lines of text. The distance between the baseline of each line of text is (*spacing* × *size)* in the appropriate direction (depending on other fields described below). The effects of the *size* and *spacing* field are depicted in Figure 6.7 (*spacing* greater than 1.0). Values of the *spacing* field shall be non-negative.

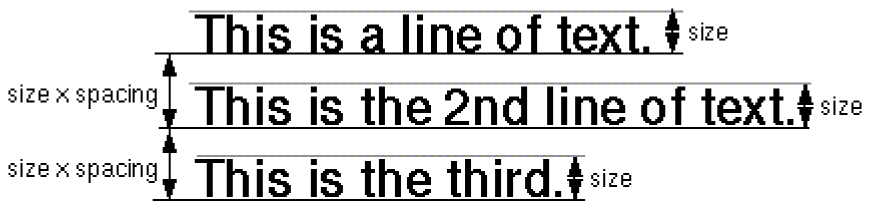


Figure 6.7 -- Text *size* and *spacing* fields

6.20.2 Font family and style

Font attributes are defined with the *family* and *style* fields. The browser shall map the specified font attributes to an appropriate available font as described below.

The *family* field contains a case-sensitive MFString value that specifies a sequence of font family names in preference order. The browser shall search the MFString value for the first font family name matching a supported font family. If none of the string values matches a supported font family, the default font family **"SERIF"** shall be used. All browsers shall support at least **"SERIF"** (the default) for a serif font such as Times Roman; **"SANS"** for a sans-serif font such as Helvetica; and **"TYPEWRITER"** for a fixed-pitch font such as Courier. An empty *family* value is identical to **["SERIF"]**.

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

6.20.3 Direction and justification

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

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

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

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

The major alignment is along the X-axis when *horizontal* is TRUE and along the Y-axis when *horizontal is* FALSE. The minor alignment is along the Y-axis when *horizontal* is TRUE and along the X-axis when *horizontal is* FALSE. The possible values for each enumerant of the *justify* field are **"FIRST"**, **"BEGIN"**, **"MIDDLE"**, and **"END"**. For major alignment, each line of text is positioned individually according to the major alignment enumerant. For minor alignment, the block of text representing all lines together is positioned according to the minor alignment enumerant. Tables 6.2-6.5 describe the behaviour in terms of which portion of the text is at the origin

Table 6.2 -- Major Alignment, *horizontal* = TRUE

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

Table 6.3 -- Major Alignment, *horizontal* = FALSE

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

Table 6.4 -- Minor Alignment, *horizontal* = TRUE

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

Table 6.5 -- Minor Alignment, *horizontal* = FALSE

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

The default minor alignment is **"FIRST"**. This is a special case of minor alignment when *horizontal* is TRUE. Text starts at the baseline at the Y-axis. In all other cases, **"FIRST"** is identical to **"BEGIN"**. In Tables 6.6 and 6.7, each colour-coded cross-hair indicates where the X-axis and Y-axis shall be in relation to the text. Figure 6.8 describes the symbols used in Tables 6.6 and 6.7.

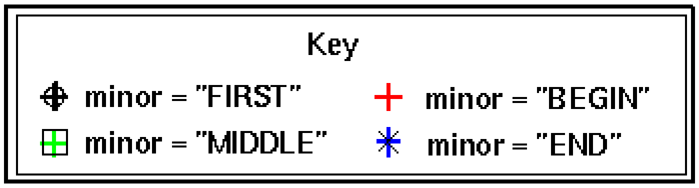
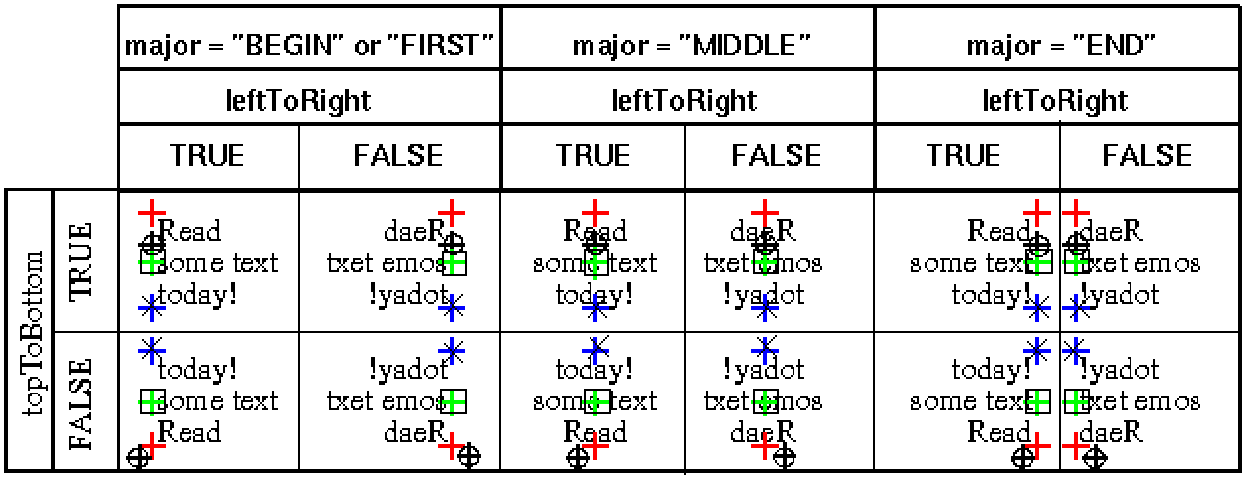


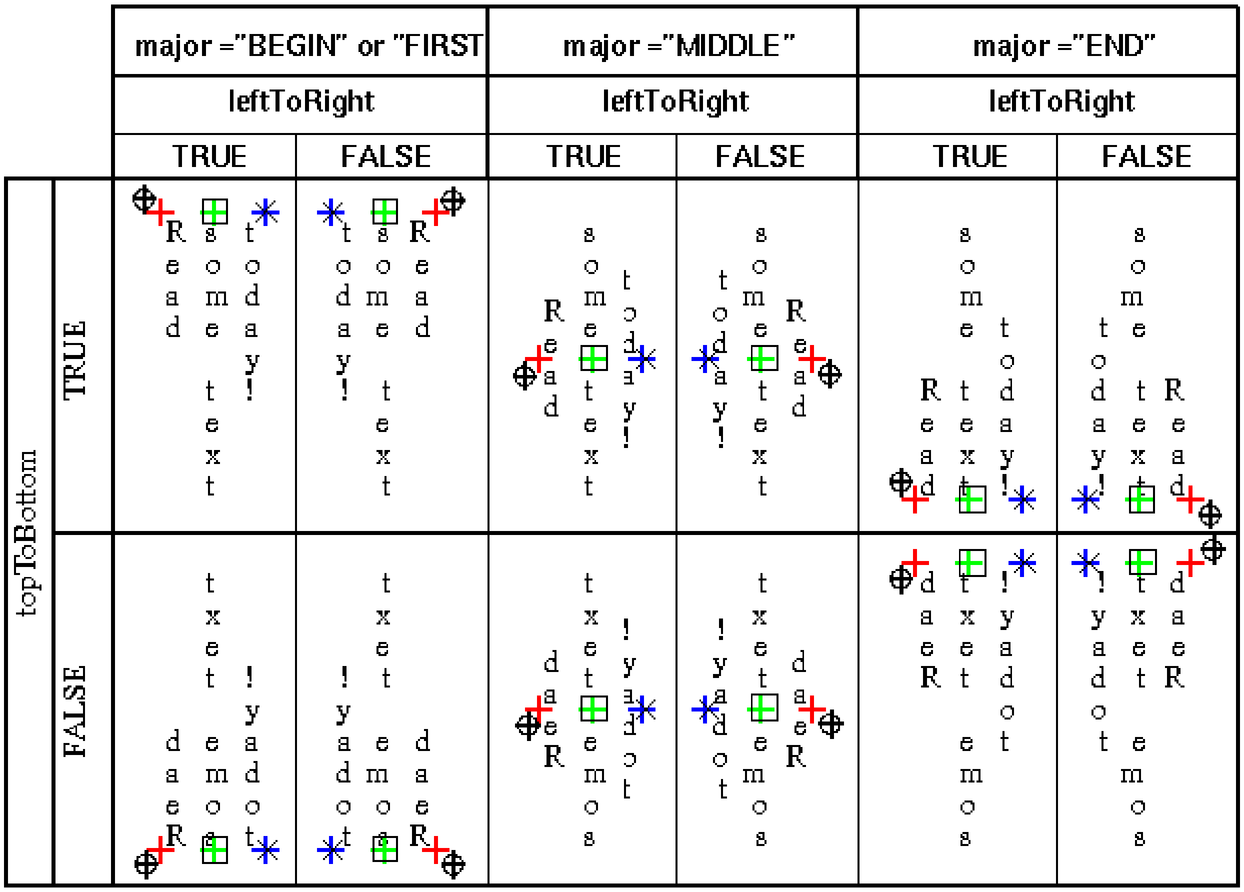
Figure 6.8 -- Key for Tables 6.6 and 6.7

Table 6.6 -- ***horizontal*** **= TRUE**



Note: The "FIRST" minor axis marker ⊕ is offset from the "BEGIN" minor axis marker + in cases that they are coincident for presentation purposes only.

Table 6.7 -- *horizontal = FALSE*



Note: In every case, the "FIRST" minor axis marker ⊕ is coincident with the "BEGIN" minor axis marker + (and is offset for presentation purposes only).

6.20.4 Language

The *language* field specifies the context of the language for the text string. Due to the multilingual nature of the ISO/IEC 10646-1:1993, the *language* field is needed to provide a proper language attribute of the text string. The format is based on RFC 1766: language[_territory] 2.[1766]. The value for the language tag is based on ISO 639:1988 (e.g., 'zh' for Chinese, 'jp' for Japanese, and 'sc' for Swedish.) The territory tag is based on ISO 3166:1993 country codes (e.g., 'TW' for Taiwan and 'CN' for China for the 'zh' Chinese language tag). If the *language* field is empty (""), local language bindings are used.

See 2, Normative references, for more information on RFC 1766 (2.[1766]), ISO/IEC 10646:1993 (2.[UTF8]), ISO/IEC 639:1998 (2.[I639]), and ISO 3166:1993 (2.[I3166]).

6.21 Group

```
Group {
  eventIn      MFNode  addChildren
  eventIn      MFNode  removeChildren
  exposedField MFNode  children      []
  field        SFVec3f bboxCenter    0 0 0     # (-∞,∞)
  field        SFVec3f bboxSize      -1 -1 -1  # (0,∞) or -1,-1,-1
}
```

A Group node contains children nodes without introducing a new transformation. It is equivalent to a Transform node containing an identity transform.

More details on the *children*, *addChildren*, and *removeChildren* fields and eventIns can be found in 4.6.5, Grouping and children nodes.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Group node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and, if needed, is calculated by the browser. A description of the *bboxCenter* and *bboxSize* fields is contained in 4.6.4, Bounding boxes.

6.22 ImageTexture

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

The ImageTexture node defines a texture map by specifying an image file and general parameters for mapping to geometry. Texture maps are defined in a 2D coordinate system (s, t) that ranges from [0.0, 1.0] in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to s=0, t=0, and the top-right pixel of the image corresponds to s=1, t=1. These relationships are depicted in Figure 6.9.

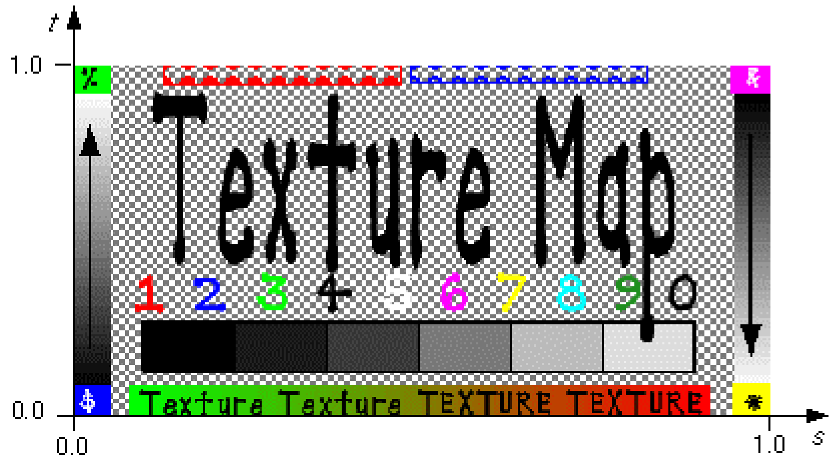


Figure 6.9 -- Texture map coordinate system

The texture is read from the URL specified by the *url* field. When the *url* field contains no values ([]), texturing is disabled. Browsers shall support the JPEG (see 2. [JPEG]) and PNG (see 2. [PNG]) image file formats. In addition, browsers may support other image formats (e.g. CGM, 2. [CGM]) which can be rendered into a 2D image. Support for the GIF format (see E. [GIF]) is also recommended (including transparency). Details on the *url* field can be found in 4.5, VRML and the World Wide Web.

See 4.6.11, Texture maps, for a general description of texture maps.

See 4.14, Lighting model, for a description of lighting equations and the interaction between textures, materials, and geometry appearance.

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 [0.0, 1.0] 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 [0.0, 1.0] range. The *repeatT* field is analogous to the *repeatS* field.

6.23 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        []        # [-1,∞)
  field         SFBool  colorPerVertex    TRUE
  field         SFBool  convex            TRUE
  field         MFInt32 coordIndex        []        # [-1,∞)
  field         SFFloat creaseAngle       0         # [0,∞)
  field         MFInt32 normalIndex       []        # [-1,∞)
  field         SFBool  normalPerVertex   TRUE
  field         SFBool  solid             TRUE
  field         MFInt32 texCoordIndex     []        # [-1,∞)
}
```

The IndexedFaceSet node represents a 3D shape formed by constructing faces (polygons) from vertices listed in the *coord* field. The *coord* field contains a Coordinate node that defines the 3D vertices referenced by the *coordIndex* field. IndexedFaceSet uses the indices in its *coordIndex* field to specify the polygonal faces by indexing into the coordinates in the Coordinate node. An index of "-1" indicates that the current face has ended and the next one begins. The last face may be (but does not have to be) followed by a "-1" index. If the greatest index in the *coordIndex* field is N, the Coordinate node shall contain N+1 coordinates (indexed as 0 to N). Each face of the IndexedFaceSet shall have:

a. at least three non-coincident vertices;

b. vertices that define a planar polygon;

c. vertices that define a non-self-intersecting polygon.

Otherwise, The results are undefined.

The IndexedFaceSet node is specified in the local coordinate system and is affected by the transformations of its ancestors.

Descriptions of the *coord*, *normal*, and *texCoord* fields are provided in the Coordinate, Normal, and TextureCoordinate nodes, respectively.

Details on lighting equations and the interaction between *color* field, *normal* field, textures, materials, and geometries are provided in 4.14, Lighting model.

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

d. If *colorPerVertex* is FALSE, colours are applied to each face, as follows:

    1. If the *colorIndex* field is not empty, then one colour is used for each face of the IndexedFaceSet. There shall 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 shall be N+1 colours in the Color node. The *colorIndex* field shall not contain any negative entries.

    2. If the *colorIndex* field is empty, then the colours in the Color node are applied to each face of the IndexedFaceSet in order. There shall be at least as many colours in the Color node as there are faces.

e. If *colorPerVertex* is TRUE, colours are applied to each vertex, as follows:

    1. If the *colorIndex* field is not empty, then colours are applied to 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 shall contain at least as many indices as the *coordIndex* field, and shall 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 shall be N+1 colours in the Color node.

    2. If the *colorIndex* field is empty, then the *coordIndex* field is used to choose colours from the Color node. If the greatest index in the *coordIndex* field is N, then there shall be N+1 colours in the Color node.

If the *color* field is NULL, the geometry shall be rendered normally using the Material and texture defined in the Appearance node (see 4.14, Lighting model, for details).

If the *normal* field is not NULL, it shall 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 colours to vertices/faces (where *normalPerVertex* corresponds to *colorPerVertex* and *normalIndex* corresponds to *colorIndex*). If the *normal* field is NULL, the browser shall automatically generate normals, using *creaseAngle* to determine if and how normals are smoothed across shared vertices (see 4.6.3.5, Crease angle field).

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

f. 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 shall contain at least as many indices as the *coordIndex* field, and shall 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 shall be N+1 texture coordinates in the TextureCoordinate node.

g. 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 shall be N+1 texture coordinates in the TextureCoordinate node.

If the *texCoord* field is NULL, a default texture coordinate mapping is calculated using the local coordinate system 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, ties shall 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. Figure 6.10 illustrates the default texture coordinates for a simple box shaped IndexedFaceSet with an X dimension twice as large as the Z dimension and four times as large as the Y dimension. Figure 6.11 illustrates the original texture image used on the IndexedFaceSet used in Figure 6.10.

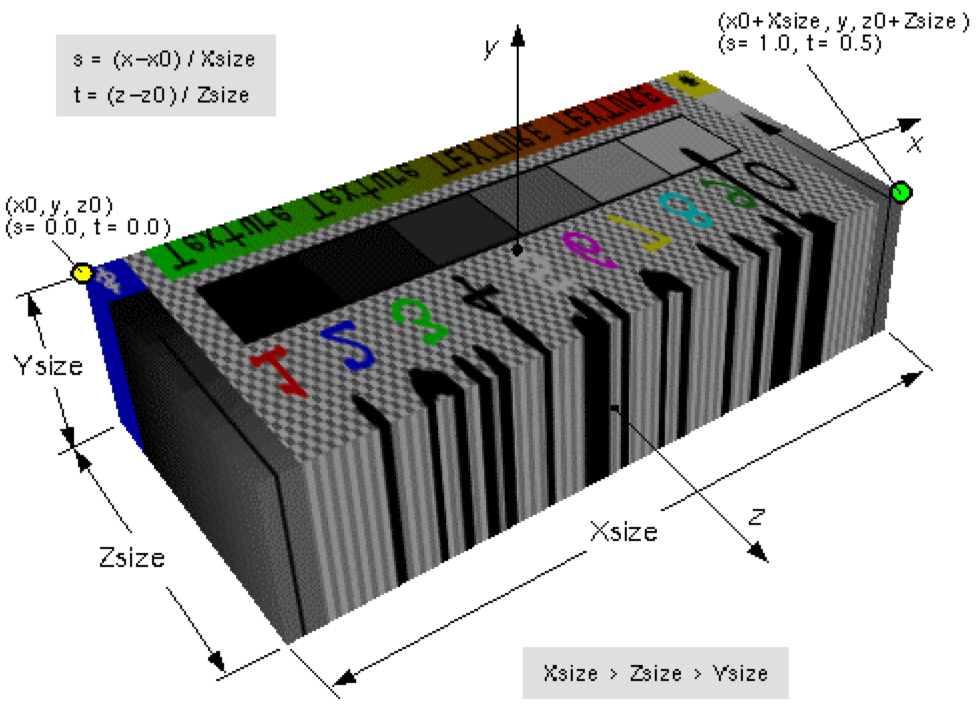


Figure 6.10 -- IndexedFaceSet texture default mapping

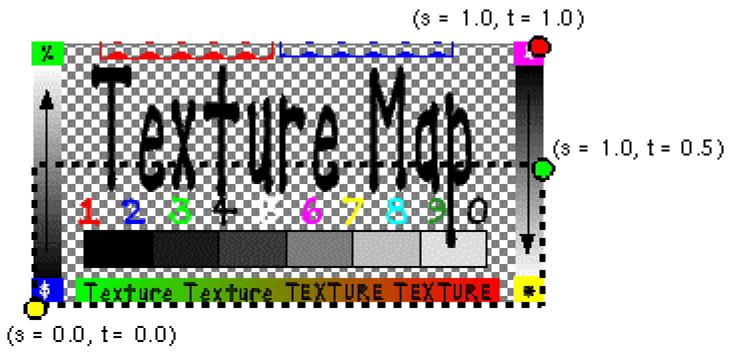


Figure 6.11 -- ImageTexture for IndexedFaceSet in Figure 6.10

Subclause 4.6.3, Shapes and geometry, provides a description of the *ccw*, *solid*, *convex*, and *creaseAngle* fields.

6.24 IndexedLineSet

```
IndexedLineSet {
  eventIn       MFInt32 set_colorIndex
  eventIn       MFInt32 set_coordIndex
  exposedField  SFNode  color             NULL
  exposedField  SFNode  coord             NULL
  field         MFInt32 colorIndex        []      # [-1,∞)
  field         SFBool  colorPerVertex    TRUE
  field         MFInt32 coordIndex        []      # [-1,∞)
}
```

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

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

Lines are not lit, are not texture-mapped, and do not participate in collision detection. The width of lines is implementation dependent and each line segment is solid (i.e., not dashed).

If the *color* field is not NULL, it shall contain a Color node. The colours are applied to the line(s) as follows:

a. If *colorPerVertex* is FALSE:

   1. If the *colorIndex* field is not empty, one colour is used for each polyline of the IndexedLineSet. There shall 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, there shall be N+1 colours in the Color node. The *colorIndex* field shall not contain any negative entries.

   2. If the *colorIndex* field is empty, the colours from the Color node are applied to each polyline of the IndexedLineSet in order. There shall be at least as many colours in the Color node as there are polylines.

b. If *colorPerVertex* is TRUE:

   1. If the *colorIndex* field is not empty, colours 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 shall contain at least as many indices as the *coordIndex* field and shall 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, there shall be N+1 colours in the Color node.

   2. If the *colorIndex* field is empty, the *coordIndex* field is used to choose colours from the Color node. If the greatest index in the *coordIndex* field is N, there shall be N+1 colours in the Color node.

If the *color* field is NULL and there is a Material defined for the Appearance affecting this IndexedLineSet, the *emissiveColor* of the Material shall be used to draw the lines. Details on lighting equations as they affect IndexedLineSet nodes are described in 4.14, Lighting model.

6.25 Inline

```
Inline {
  exposedField MFString url         []
  field        SFVec3f  bboxCenter 0 0 0     # (-∞,∞)
  field        SFVec3f  bboxSize   -1 -1 -1  # (0,∞) or -1,-1,-1
}
```

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

Each specified URL shall refer to a valid VRML file that contains a list of children nodes, prototypes, and routes at the top level as described in 4.6.5, Grouping and children nodes. The results are undefined if the URL refers to a file that is not VRML or if the VRML file contains non-children nodes at the top level.

If multiple URLs are specified, the browser may display a URL of a lower preference VRML file while it is obtaining, or if it is unable to obtain, the higher preference VRML file. Details on the *url* field and preference order can be found in 4.5, VRML and the World Wide Web.

The results are undefined if the contents of the URL change after it has been loaded.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Inline node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. A description of the *bboxCenter* and *bboxSize* fields is in 4.6.4, Bounding boxes.

6.26 LOD

```
LOD {
  exposedField MFNode   level    []
  field        SFVec3f  center   0 0 0    # (-∞,∞)
  field        MFFloat  range    []       # (0,∞)
}
```

The LOD node specifies various levels of detail or complexity for a given object, and provides hints allowing browsers to automatically choose the appropriate version of the object based on the distance from the user. The *level* field contains a list of nodes that represent the same object or objects at varying levels of detail, ordered from highest level of detail to the lowest level of detail. The *range* field specifies the ideal distances at which to switch between the levels. Subclause 4.6.5, Grouping and children nodes, contains details on the types of nodes that are legal values for *level*.

The *center* field is a translation offset in the local coordinate system that specifies the centre of the LOD node for distance calculations.

The number of nodes in the *level* field shall exceed the number of values in the *range* field by one (i.e., N+1 *level* values for N *range* values). The *range* field contains monotonic increasing values that shall be greater than zero. In order to calculate which level to display, first the distance is calculated from the viewer's location, transformed into the local coordinate system of the LOD node (including any scaling transformations), to the *center* point of the LOD node. Then, the LOD node evaluates the step function *L(d)* to choose a level for a given value of *d* (where *d* is the distance from the viewer position to the centre of the LOD node).

Let *n* ranges, R_0, R_1, R_2, ..., R_{n-1}, partition the domain (0, +*infinity*) into *n*+1 subintervals given by (0, R_0), [R_0, R_1)... , [R_{n-1}, +*infinity*). Also, let *n* levels L_0, L_1, L_2, ..., L_{n-1} be the values of the step function function *L(d)*. The level node, *L(d)*, for a given distance *d* is defined as follows:

```
L(d) = L0,    if d < R0,
     = Li+1,  if Ri <= d < Ri+1, for -1 < i < n-1,
     = Ln-1,  if d >= Rn-1.
```

Specifying too few levels will result in the last level being used repeatedly for the lowest levels of detail. If more levels than ranges are specified, the extra levels are ignored. An empty range field is an exception to this rule. This case is a hint to the browser that it may choose a level automatically to maintain a constant display rate. Each value in the *range* field shall be greater than the previous value.

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

6.27 Material

```
Material {
  exposedField SFFloat ambientIntensity  0.2         # [0,1]
  exposedField SFColor diffuseColor      0.8 0.8 0.8 # [0,1]
  exposedField SFColor emissiveColor     0 0 0       # [0,1]
  exposedField SFFloat shininess         0.2         # [0,1]
  exposedField SFColor specularColor     0 0 0       # [0,1]
  exposedField SFFloat transparency      0           # [0,1]
}
```

The Material node specifies surface material properties for associated geometry nodes and is used by the VRML lighting equations during rendering. Subclause 4.14, Lighting model, contains a detailed description of the VRML lighting model equations.

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

The fields in the Material node determine how light reflects off an object to create colour:

a. The *ambientIntensity* field specifies how much ambient light from light sources this surface shall reflect. Ambient light is omnidirectional and depends only on the number of light sources, not their positions with respect to the surface. Ambient colour is calculated as *ambientIntensity* × *diffuseColor*.

b. The *diffuseColor* field 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.

c. The *emissiveColor* field models "glowing" objects. This can be useful for displaying pre-lit models (where the light energy of the room is computed explicitly), or for displaying scientific data.

d. The *specularColor* and *shininess* fields determine the specular highlights (e.g., the shiny spots on an apple). When the angle from the light to the surface is close to the angle from the surface to the viewer, the *specularColor* is added to the diffuse and ambient colour calculations. Lower shininess values produce soft glows, while higher values result in sharper, smaller highlights.

e. The *transparency* field specifies how "clear" an object is, with 1.0 being completely transparent, and 0.0 completely opaque.

6.28 MovieTexture

```
MovieTexture {
  exposedField SFBool   loop             FALSE
  exposedField SFFloat  speed            1.0      # (-∞,∞)
  exposedField SFTime   startTime        0        # (-∞,∞)
  exposedField SFTime   stopTime         0        # (-∞,∞)
  exposedField MFString url              []
  field        SFBool   repeatS          TRUE
  field        SFBool   repeatT          TRUE
  eventOut     SFTime   duration_changed
  eventOut     SFBool   isActive
}
```

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

Texture maps are defined in a 2D coordinate system (s, t) that ranges from 0.0 to 1.0 in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to s=0.0, t=0.0, and the top-right pixel of the image corresponds to s=1.0, t=1.0. Figure 6.12 depicts the texture map coordinate system of the MovieTexture.

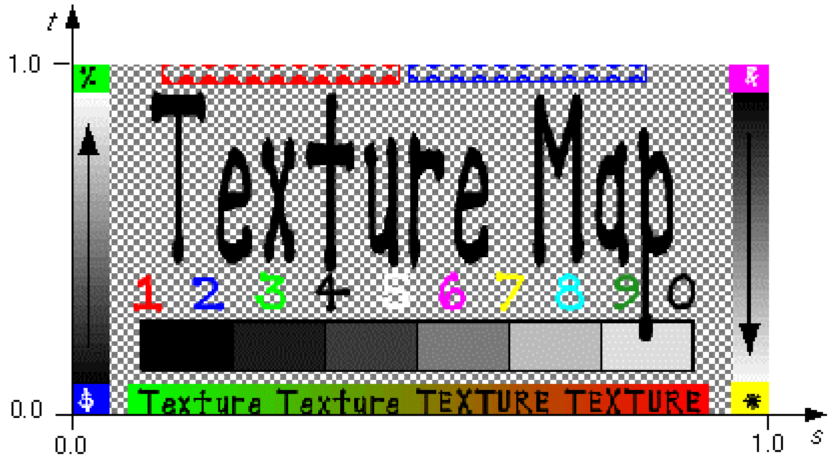


Figure 6.12 -- MovieTexture node coordinate system

The *url* field that defines the movie data shall support MPEG1-Systems (audio and video) or MPEG1-Video (video-only) movie file formats 2.[MPEG]. Details on the *url* field can be found in 4.5, VRML and the World Wide Web.

MovieTexture nodes can be referenced by an Appearance node's *texture* field (as a movie texture) and by a Sound node's *source* field (as an audio source only).

See 4.6.11, Texture maps, for a general description of texture maps.

4.14, Lighting model, contains 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 node) to determine the duration of a movie. A value of "-1" implies the movie has not yet loaded or the value is unavailable for some reason.

The *loop*, *startTime*, and *stopTime* exposedFields and the *isActive* eventOut, and their effects on the MovieTexture node, are discussed in detail in the 4.6.9, Time-dependent nodes, section. The cycle of a MovieTexture node is the length of time in seconds for one playing of the movie at the specified *speed*.

The *speed* exposedField indicates how fast the movie shall be played. A *speed* of 2 indicates the movie plays twice as fast. The *duration_changed* output is not affected by the *speed* exposedField. *set_speed* events are ignored while the movie is playing. A negative *speed* implies that the movie will play backwards.

If a MovieTexture node is inactive when the movie is first loaded, frame 0 of the movie texture is displayed if *speed* is non-negative or the last frame of the movie texture is shown if *speed* is negative (see 4.11.3, Discrete and continuous changes). A MovieTexture node shall display frame 0 if *speed* = 0. For positive values of *speed*, an active MovieTexture node displays the frame at movie time *t* as follows (i.e., in the movie's local time system with frame 0 at time 0 with *speed* = 1):

```
    t = (now - startTime) modulo (duration/speed)
```

If *speed* is negative, the MovieTexture node displays the frame at movie time:

```
    t = duration - ((now - startTime) modulo |duration/speed|)
```

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

6.29 NavigationInfo

```
NavigationInfo {
  eventIn      SFBool   set_bind
  exposedField MFFloat  avatarSize      [0.25, 1.6, 0.75] # [0,∞)
  exposedField SFBool   headlight       TRUE
  exposedField SFFloat  speed           1.0               # [0,∞)
  exposedField MFString type            ["WALK", "ANY"]
  exposedField SFFloat  visibilityLimit 0.0               # [0,∞)
  eventOut     SFBool   isBound
}
```

The NavigationInfo node contains information describing the physical characteristics of the viewer's avatar and viewing model. NavigationInfo node is a bindable node (see 4.6.10, Bindable children nodes*)*. Thus, there exists a NavigationInfo node stack in which the top-most NavigationInfo node on the stack is the currently bound NavigationInfo node. The current NavigationInfo node is considered to be a child of the current Viewpoint node regardless of where it is initially located in the VRML file. Whenever the current Viewpoint nodes changes, the current NavigationInfo node shall be re-parented to it by the browser. Whenever the current NavigationInfo node changes, the new NavigationInfo node shall be re-parented to the current Viewpoint node by the browser.

If a TRUE value is sent to the *set_bind* eventIn of a NavigationInfo node, the node is pushed onto the top of the NavigationInfo node stack. When a NavigationInfo node is bound, the browser uses the fields of the NavigationInfo node to set the navigation controls of its user interface and the NavigationInfo node is conceptually re-parented under the currently bound Viewpoint node. All subsequent scaling changes to the current Viewpoint node's coordinate system automatically change aspects (see below) of the NavigationInfo node values used in the browser (e.g., scale changes to any ancestors' transformations). A FALSE value sent to *set_bind* pops the NavigationInfo node from the stack, results in an *isBound* FALSE event, and pops to the next entry in the stack which shall be re-parented to the current Viewpoint node. 4.6.10, Bindable children nodes, has more details on binding stacks.

The *type* field specifies an ordered list of navigation paradigms that specify a combination of navigation types and the initial navigation type. The navigation type of the currently bound NavigationInfo node determines the user interface capabilities of the browser. For example, if the currently bound NavigationInfo node's *type* is "WALK", the browser shall present a WALK navigation user interface paradigm (see below for description of WALK). Browsers shall recognize and support at least the following navigation types: "ANY", "WALK", "EXAMINE", "FLY", and "NONE".

If "ANY" does not appear in the *type* field list of the currently bound NavigationInfo, the browser's navigation user interface shall be restricted to the recognized navigation types specified in the list. In this case, browsers shall not present a user interface that allows the navigation type to be changed to a type not specified in the list. However, if any one of the values in the *type* field are "ANY", the browser may provide any type of navigation interface, and allow the user to change the navigation type dynamically. Furthermore, the first recognized type in the list shall be the initial navigation type presented by the browser's user interface.

ANY navigation specifies that the browser may choose the navigation paradigm that best suits the content and provide a user interface to allow the user to change the navigation paradigm dynamically. The results are undefined if the currently bound NavigationInfo's *type* value is "ANY" and Viewpoint transitions (see 6.53, Viewpoint) are triggered by the Anchor node (see 6.2, Anchor) or the `loadURL()`scripting method (see 4.12.10, Browser script interface).

WALK navigation is used for exploring a virtual world on foot or in a vehicle that rests on or hovers above the ground. It is strongly recommended that WALK navigation define the up vector in the +Y direction and provide some form of terrain following and gravity in order to produce a walking or driving experience. If the bound NavigationInfo's *type* is "WALK", the browser shall strictly support collision detection (see 6.8, Collision).

FLY navigation is similar to WALK except that terrain following and gravity may be disabled or ignored. There shall still be some notion of "up" however. If the bound NavigationInfo's *type* is "FLY", the browser shall strictly support collision detection (see 6.8, Collision).

EXAMINE navigation is used for viewing individual objects and often includes (but does not require) the ability to spin around the object and move the viewer closer or further away.

NONE navigation disables and removes all browser-specific navigation user interface forcing the user to navigate using only mechanisms provided in the scene, such as Anchor nodes or scripts that include `loadURL()`.

If the NavigationInfo type is "WALK", "FLY", "EXAMINE", or "NONE" or a combination of these types (i.e., "ANY" is not in the list), Viewpoint transitions (see 6.53, Viewpoint) triggered by the Anchor node (see 6.2, Anchor) or the `loadURL()`scripting method (see 4.12.10, Browser script interface) shall be implemented as a jump cut from the old Viewpoint to the new Viewpoint with transition effects that shall not trigger events besides the exit and enter events caused by the jump.

Browsers may create browser-specific navigation type extensions. It is recommended that extended *type* names include a unique suffix (e.g., HELICOPTER_mydomain.com) to prevent conflicts. Viewpoint transitions (see 6.53, Viewpoint) triggered by the Anchor node (see 6.2, Anchor) or the `loadURL()`scripting method (see 4.12.10, Browser script interface) are undefined for extended navigation types. If none of the types are recognized by the browser, the default "ANY" is used. These strings values are case sensitive ("any" is not equal to "ANY").

The *speed* field specifies the rate at which the viewer travels through a scene in metres per second. Since browsers may provide mechanisms to travel faster or slower, this field specifies the default, average speed of the viewer when the NavigationInfo node is bound. If the NavigationInfo *type* is EXAMINE, *speed* shall not affect the viewer's rotational speed. Scaling in the transformation hierarchy of the currently bound Viewpoint node (see above) scales the *speed*; parent translation and rotation transformations have no effect on *speed*. Speed shall be non-negative. Zero speed indicates that the avatar's position is stationary, but its orientation and field of view may still change. If the navigation *type* is "NONE", the *speed* field has no effect.

The *avatarSize* field specifies the user's physical dimensions in the world for the purpose of collision detection and terrain following. It is a multi-value field allowing several dimensions to be specified. The first value shall be the allowable distance between the user's position and any collision geometry (as specified by a Collision node) before a collision is detected. The second shall be the height above the terrain at which the browser shall maintain the viewer. The third shall be the height of the tallest object over which the viewer can move. This allows staircases to be built with dimensions that can be ascended by viewers in all browsers. The transformation hierarchy of the currently bound Viewpoint node scales the *avatarSize*. Translations and rotations have no effect on *avatarSize*.

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

Geometry beyond the visibilityLimit may not be rendered. A value of 0.0 indicates an infinite visibility limit. The *visibilityLimit* field is restricted to be greater than or equal to zero.

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

The *headlight* field specifies whether a browser shall turn on a headlight. 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 turn the headlight off. The headlight shall have *intensity* = 1, *color* = (1 1 1), *ambientIntensity* = 0.0, and *direction* = (0 0 -1).

It is recommended that the near clipping plane be set to one-half of the collision radius as specified in the *avatarSize* field (setting the near plane to this value prevents excessive clipping of objects just above the collision volume, and also provides a region inside the collision volume for content authors to include geometry intended to remain fixed relative to the viewer). Such geometry shall not be occluded by geometry outside of the collision volume.

6.30 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 (e.g., IndexedFaceSet and ElevationGrid). This node contains one multiple-valued field that contains the normal vectors. Normals shall be of unit length.

6.31 NormalInterpolator

```
NormalInterpolator {
  eventIn      SFFloat set_fraction       # (-∞,∞)
  exposedField MFFloat key           []   # (-∞,∞)
  exposedField MFVec3f keyValue      []   # (-∞,∞)
  eventOut     MFVec3f value_changed
}
```

The NormalInterpolator node interpolates among a list of normal vector sets specified by the *keyValue* field. The output vector, *value_changed*, shall be a set of normalized vectors.

Values in the *keyValue* field shall be of unit length. The number of normals in the *keyValue* field shall 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 shall 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 the unit sphere shall lie along the shortest arc (on the unit sphere) connecting points P and Q. Also, equally spaced input fractions shall result in arcs of equal length. The results are undefined if P and Q are diagonally opposite.

A more detailed discussion of interpolators is provided in 4.6.8, Interpolator nodes.

6.32 OrientationInterpolator

```
OrientationInterpolator {
  eventIn      SFFloat    set_fraction      # (-∞,∞)
  exposedField MFFloat    key          []   # (-∞,∞)
  exposedField MFRotation keyValue     []   # [-1,1], (-∞,∞)
  eventOut     SFRotation value_changed
}
```

The OrientationInterpolator node interpolates among a list of rotation values specified in the *keyValue* field. These rotations are absolute in object space and therefore are not cumulative. The *keyValue* field shall contain exactly as many rotations as there are keyframes in the *key* field.

An orientation represents the final position of an object after a rotation has been applied. An OrientationInterpolator interpolates between two orientations by computing the shortest path on the unit sphere between the two orientations. The interpolation is linear in arc length along this path. The results are undefined if the two orientations are diagonally opposite.

If two consecutive *keyValue* values exist such that the arc length between them is greater than π, the interpolation will take place on the arc complement. For example, the interpolation between the orientations (0, 1, 0, 0) and (0, 1, 0, 5.0) is equivalent to the rotation between the orientations (0, 1, 0, 2π) and (0, 1, 0, 5.0).

A more detailed discussion of interpolators is contained in 4.6.8, Interpolator nodes.

6.33 PixelTexture

```
PixelTexture {
  exposedField SFImage  image      0 0 0    # see 5.5, SFImage
  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 (*image* field) and parameters controlling tiling repetition of the texture onto geometry.

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

See 4.6.11, Texture maps, for a general description of texture maps. Figure 6.13 depicts an example PixelTexture.

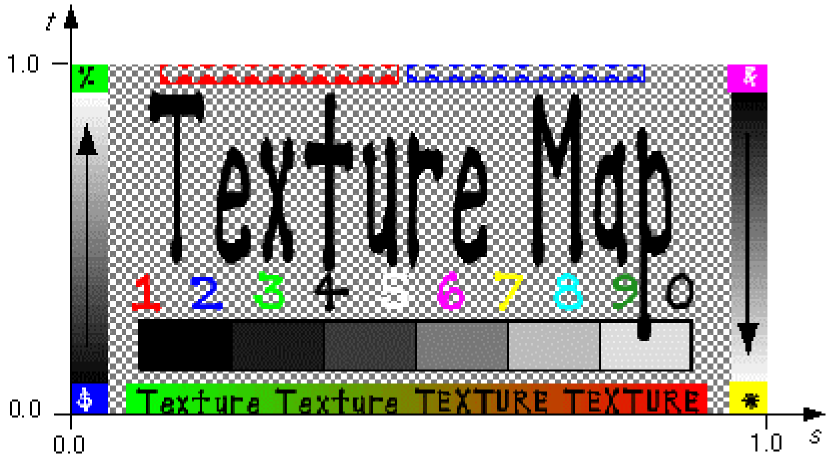


Figure 6.13 -- PixelTexture node

See 4.14 ,Lighting model, for a description of how the texture values interact with the appearance of the geometry. 5.5, SFImage, describes the specification of 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 0.0 to 1.0 range. The *repeatT* field is analogous to the *repeatS* field.

6.34 PlaneSensor

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

The PlaneSensor node maps pointing device motion into two-dimensional translation in a plane parallel to the Z=0 plane of the local coordinate system. The PlaneSensor node uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposedField enables and disables the PlaneSensor. If *enabled* is TRUE, the sensor reacts appropriately to user events. If *enabled* is FALSE, the sensor does not track user input or send 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 made ready for user activation.

The PlaneSensor node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for details on using the pointing device to activate the PlaneSensor.

Upon activation of the pointing device (e.g., mouse button down) while indicating the sensor's geometry, an *isActive* TRUE event is sent. Pointer motion is mapped into relative translation in the *tracking plane*, (a plane parallel to the sensor's local Z=0 plane and coincident with the initial point of intersection). For each subsequent movement of the bearing, a *translation_changed* event is output which corresponds to the sum of the relative translation from the original intersection point to the intersection point of the new bearing in the plane plus the *offset* value. The sign of the translation is defined by the Z=0 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_changed* value and an *offset_changed* event is generated. More details are provided in 4.6.7.4, Drag sensors.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is deactivated and generates an *isActive* FALSE event. Other pointing-device sensors shall 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 typically reflect the state of the primary button associated with the device (i.e., *isActive* is TRUE when the primary button is pressed, and is FALSE when it is released). If a 3D pointing device (e.g., wand) is in use, *isActive* events typically reflect whether the pointer is within or in contact with the sensor's geometry.

minPosition and *maxPosition* may be set to clamp *translation_changed* events to a range of values as measured from the origin of the Z=0 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 and moved, *trackPoint_changed* and *translation_changed* events are sent. *trackPoint_changed* events represent the unclamped intersection points on the surface of the tracking plane. If the pointing device is dragged off of the tracking plane while activated (e.g., above horizon line), browsers may interpret this in a variety 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.

Further information about this behaviour can be found in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

6.35 PointLight

```
PointLight {
  exposedField SFFloat ambientIntensity  0       # [0,1]
  exposedField SFVec3f attenuation       1 0 0   # [0,∞)
  exposedField SFColor color             1 1 1   # [0,1]
  exposedField SFFloat intensity         1       # [0,1]
  exposedField SFVec3f location          0 0 0   # (-∞,∞)
  exposedField SFBool  on                TRUE
  exposedField SFFloat radius            100     # [0,∞)
}
```

The PointLight node specifies a point light source at a 3D location in the local coordinate system. A point light source emits light equally in all directions; that is, it is omnidirectional. PointLight nodes are specified in the local coordinate system and are affected by ancestor transformations.

Subclause 4.6.6, Light sources, contains a detailed description of the *ambientIntensity*, *color*, and *intensity* fields.

A PointLight node illuminates geometry within *radius* metres of its *location*. Both radius and location are affected by ancestors' transformations (scales affect *radius* and transformations affect *location*). The *radius* field shall be greater than or equal to zero.

PointLight node's illumination falls off with distance as specified by three *attenuation* coefficients. The attenuation factor is $1/\max(attenuation[0] + attenuation[1] \times r + attenuation[2] \times r^2, 1)$, where r is the distance from the light to the surface being illuminated. The default is no attenuation. An *attenuation* value of (0, 0, 0) is identical to (1, 0, 0). Attenuation values shall be greater than or equal to zero. A detailed description of VRML's lighting equations is contained in 4.14, Lighting model.

6.36 PointSet

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

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

PointSet nodes are not lit, not texture-mapped, nor do they participate in collision detection. The size of each point is implementation-dependent.

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

If the *color* field is NULL and there is a Material node defined for the Appearance node affecting this PointSet node, the *emissiveColor* of the Material node shall be used to draw the points. More details on lighting equations can be found in 4.14, Lighting model.

6.37 PositionInterpolator

```
PositionInterpolator {
  eventIn      SFFloat set_fraction        # (-∞,∞)
  exposedField MFFloat key            []   # (-∞,∞)
  exposedField MFVec3f keyValue       []   # (-∞,∞)
  eventOut     SFVec3f value_changed
}
```

The PositionInterpolator node linearly interpolates among a list of 3D vectors. The *keyValue* field shall contain exactly as many values as in the *key* field.

4.6.8, Interpolator nodes, contains a more detailed discussion of interpolators.

6.38 ProximitySensor

```
ProximitySensor {
  exposedField SFVec3f    center      0 0 0    # (-∞,∞)
  exposedField SFVec3f    size        0 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 node generates events when the viewer enters, exits, and moves within a region in space (defined by a box). A proximity sensor is enabled or disabled by sending it an *enabled* event with a value of TRUE or FALSE. A disabled sensor does not send events.

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

(z) of the box bounding the region. The components of the *size* field shall be greater than or equal to zero. ProximitySensor nodes are affected by the hierarchical transformations of their parents.

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

The *position_changed* and *orientation_changed* eventOuts send events whenever the user is contained within the proximity region and the position and orientation of the viewer changes with respect to the ProximitySensor node's coordinate system including enter and exit times. The viewer movement may be a result of a variety of circumstances resulting from browser navigation, ProximitySensor node's coordinate system changes, or bound Viewpoint node's position or orientation changes.

Each ProximitySensor node behaves independently of all other ProximitySensor nodes. Every enabled ProximitySensor node that is affected by the viewer's movement receives and sends events, possibly resulting in multiple ProximitySensor nodes receiving and sending events simultaneously. Unlike TouchSensor nodes, there is no notion of a ProximitySensor node lower in the scene graph "grabbing" events.

Instanced (DEF/USE) ProximitySensor nodes use the union of all the boxes to check for enter and exit. A multiply instanced ProximitySensor node will detect enter and exit for all instances of the box and send enter/exit events appropriately. However, the results are undefined if the any of the boxes of a multiply instanced ProximitySensor node overlap.

A ProximitySensor node that surrounds the entire world has an *enterTime* equal to the time that the world was entered and can be used to start up animations or behaviours as soon as a world is loaded. A ProximitySensor node with a box containing zero volume (i.e., any *size* field element of 0.0) cannot generate events. This is equivalent to setting the *enabled* field to FALSE.

A ProximitySensor read from a VRML file shall generate *isActive* TRUE, *position_changed*, *orientation_changed* and *enterTime* events if the sensor is enabled and the viewer is inside the proximity region. A ProximitySensor inserted into the transformation hierarchy shall generate *isActive* TRUE, *position_changed*, *orientation_changed* and *enterTime* events if the sensor is enabled and the viewer is inside the proximity region. A ProximitySensor removed from the transformation hierarchy shall generate *isActive* FALSE, *position_changed*, *orientation_changed* and *exitTime* events if the sensor is enabled and the viewer is inside the proximity region.

6.39 ScalarInterpolator

```
ScalarInterpolator {
  eventIn      SFFloat set_fraction        # (-∞,∞)
  exposedField MFFloat key           []    # (-∞,∞)
  exposedField MFFloat keyValue      []    # (-∞,∞)
  eventOut     SFFloat value_changed
}
```

This node linearly interpolates among a list of SFFloat values. This interpolator is appropriate for any parameter defined using a single floating point value. Examples include width, radius, and intensity fields. The *keyValue* field shall contain exactly as many numbers as there are keyframes in the *key* field.

A more detailed discussion of interpolators is available in 4.6.8, Interpolator nodes.

6.40 Script

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

The Script node is used to program behaviour in a scene. Script nodes typically

a. signify a change or user action;

b. receive events from other nodes;

c. contain a program module that performs some computation;

d. effect change somewhere else in the scene by sending 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 is referred to as the "script" in the rest of this description. Details on the *url* field can be found in 4.5, VRML and the World Wide Web.

Browsers are not required to support any specific language. Detailed information on scripting languages is described in 4.12, Scripting. Browsers supporting a scripting language for which a language binding is specified shall adhere to that language binding.

Sometime before a script receives the first event it shall be initialized (any language-dependent or user-defined `initialize()` is performed). The script is able to receive and process events that are sent to it. Each event that can be received shall 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 (as defined in 5, Field and event reference). *Name* shall be an identifier that is unique for this Script node.

The Script node is able to generate events in response to the incoming events. Each event that may be generated shall be declared in the Script node using the following syntax:

```
eventOut type name
```

With the exception of the *url* field, exposedFields are not allowed in Script nodes.

If the Script node's *mustEvaluate* field is FALSE, the browser may delay sending input events to the script until its outputs are needed by the browser. If the *mustEvaluate* field is TRUE, the browser shall send input events to the script as soon as possible, regardless of whether the outputs are needed. The *mustEvaluate* field shall be set to TRUE only if the Script node 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 is able to read the contents of that node's exposed fields. 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), the script may only affect the rest of the world via events sent through its eventOuts. The results are undefined if *directOutput* is FALSE and the script sends events directly to a node to which it has access.

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

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

6.41 Shape

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

The Shape node has two fields, *appearance* and *geometry*, which are used to create rendered objects in the world. The *appearance* field contains an Appearance node that specifies the visual attributes (e.g., material and texture) to be applied to the geometry. The *geometry* field contains a geometry node. The specified geometry node is rendered with the specified appearance nodes applied. See 4.6.3, Shapes and geometry, and 6.3, Appearance, for more information.

4.14, Lighting model, contains details of the VRML lighting model and the interaction between Appearance nodes and geometry nodes.

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

6.42 Sound

```
Sound {
  exposedField SFVec3f  direction     0 0 1   # (-∞,∞)
  exposedField SFFloat  intensity     1       # [0,1]
  exposedField SFVec3f  location      0 0 0   # (-∞,∞)
  exposedField SFFloat  maxBack       10      # [0,∞)
  exposedField SFFloat  maxFront      10      # [0,∞)
  exposedField SFFloat  minBack       1       # [0,∞)
  exposedField SFFloat  minFront      1       # [0,∞)
  exposedField SFFloat  priority      0       # [0,1]
  exposedField SFNode   source        NULL
  field        SFBool   spatialize    TRUE
}
```

The Sound node specifies the spatial presentation of a sound in a VRML scene. The sound is located at a point in the local coordinate system and emits sound in an elliptical pattern (defined by two ellipsoids). The ellipsoids are oriented in a direction specified by the *direction* field. The shape of the ellipsoids may be modified to provide more or less directional focus from the location of the sound.

The *source* field specifies the sound source for the Sound node. If the *source* field is not specified, the Sound node will not emit audio. The *source* field shall specify either an AudioClip node or a MovieTexture node. If a MovieTexture node is specified as the sound source, the MovieTexture shall refer to a movie format that supports sound (e.g., MPEG1-Systems, see 2.[MPEG]).

The *intensity* field adjusts the loudness (decibels) of the sound emitted by the Sound node (note: this is different from the traditional definition of intensity with respect to sound; see E.[SNDA]). The *intensity* field has a value that ranges from 0.0 to 1.0 and specifies a factor which shall be used to scale the normalized sample data of the sound source during playback. A Sound node with an intensity of 1.0 shall emit audio at its maximum loudness (before attenuation), and a Sound node with an intensity of 0.0 shall emit no audio. Between these values, the loudness should increase linearly from a -20 dB change approaching an *intensity* of 0.0 to a 0 dB change at an *intensity* of 1.0.

The *priority* field provides a hint for the browser to choose which sounds to play when there are more active Sound nodes than can be played at once due to either limited system resources or system load. 7.3.4, Sound priority, attenuation, and spatialization, describes a recommended algorithm for determining which sounds to play under such circumstances. The *priority* field ranges from 0.0 to 1.0, with 1.0 being the highest priority and 0.0 the lowest priority.

The *location* field determines the location of the sound emitter in the local coordinate system. A Sound node's output is audible only if 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 are not audible unless they are traversed. If a Sound node is disabled by a Switch or LOD node, and later it becomes part of the traversal again, the sound shall resume where it would have been had it been playing continuously.

The Sound node has an inner ellipsoid that defines a volume of space in which the maximum level of the sound is audible. Within this ellipsoid, the normalized sample data is scaled by the *intensity* field and there is no attenuation. The inner ellipsoid is defined by extending the *direction* vector through the *location*. The *minBack* and *minFront* fields specify distances behind and in front of the *location* along the *direction* vector respectively. The inner ellipsoid has one of its foci at *location* (the second focus is implicit) and intersects the *direction* vector at *minBack* and *minFront*.

The Sound node has an outer ellipsoid that defines a volume of space that bounds the audibility of the sound. No sound can be heard outside of this outer ellipsoid. The outer ellipsoid is defined by extending the *direction* vector through the *location*. The *maxBack* and *maxFront* fields specify distances behind and in front of the *location* along the *direction* vector respectively. The outer ellipsoid has one of its foci at *location* (the second focus is implicit) and intersects the *direction* vector at *maxBack* and *maxFront*.

The *minFront*, *maxFront*, *minBack*, and *maxBack* fields are defined in local coordinates, and shall be greater than or equal to zero. The *minBack* field shall be less than or equal to *maxBack*, and *minFront* shall be less than or equal to *maxFront*. The ellipsoid parameters are specified in the local coordinate system but the ellipsoids' geometry is affected by ancestors' transformations.

Between the two ellipsoids, there shall be a linear attenuation ramp in loudness, from 0 dB at the minimum ellipsoid to -20 dB at the maximum ellipsoid:

```
    attenuation = -20 × (d′ / d")
```

where d' is the distance along the location-to-viewer vector, measured from the transformed minimum ellipsoid boundary to the viewer, and d" is the distance along the location-to-viewer vector from the transformed minimum ellipsoid boundary to the transformed maximum ellipsoid boundary (see Figure 6.14).

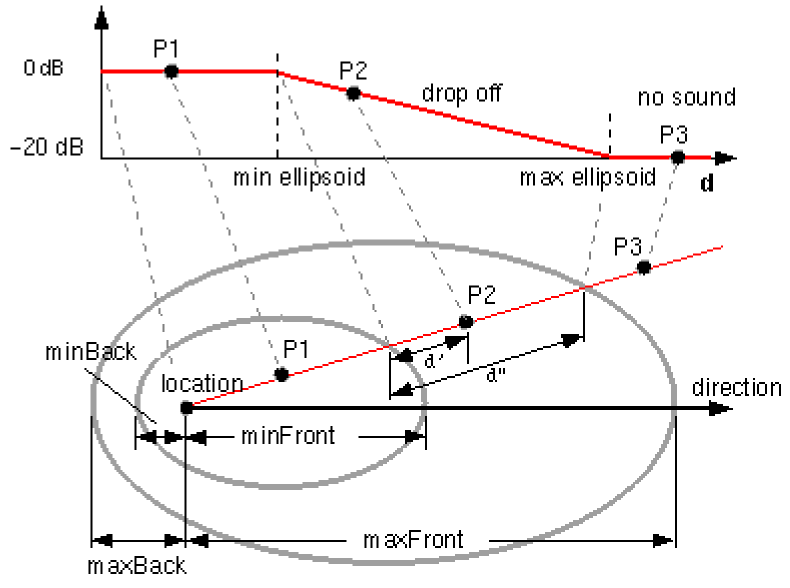


Figure 6.14 -- Sound node geometry

The *spatialize* field specifies if the sound is perceived as being directionally located relative to the viewer. If the *spatialize* field is TRUE and the viewer is located between the transformed inner and outer ellipsoids, the viewer's direction and the relative location of the Sound node should be taken into account during playback. Details outlining the minimum required spatialization functionality can be found in 7.3.4, Sound priority, attenuation, and spatialization. If the *spatialize* field is FALSE, then directional effects are ignored, but the ellipsoid dimensions and *intensity* will still affect the loudness of the sound. If the sound source is multi-channel (e.g., stereo), then the source should retain its channel separation during playback.

6.43 Sphere

```
Sphere {
  field SFFloat radius  1    # (0,∞)
}
```

The Sphere node specifies a sphere centred at (0, 0, 0) in the local coordinate system. The *radius* field specifies the radius of the sphere and shall be greater than zero. Figure 6.15 depicts the fields of the Sphere node.

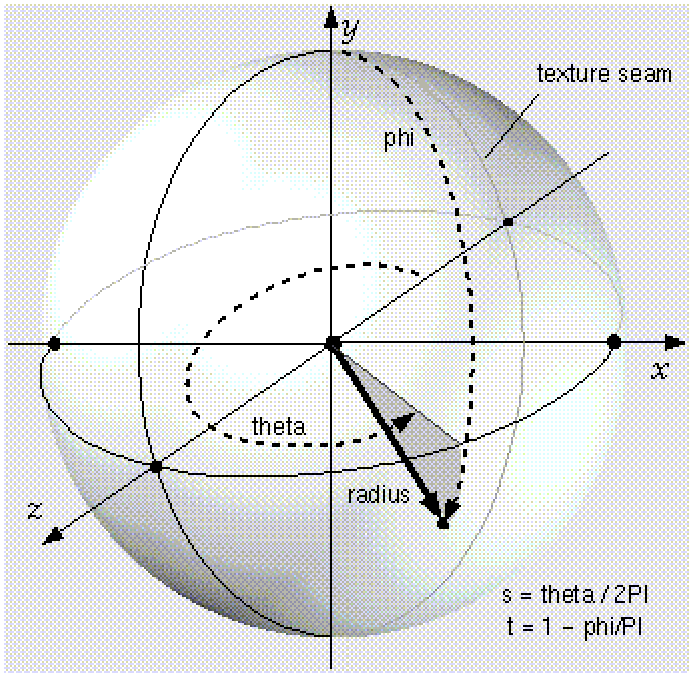


Figure 6.15 -- Sphere node

When a texture is applied to a sphere, the texture covers the entire surface, wrapping counterclockwise from the back of the sphere (i.e., longitudinal arc intersecting the -Z-axis) when viewed from the top of the sphere. The texture has a seam at the back where the X=0 plane intersects the sphere and Z values are negative. TextureTransform affects the texture coordinates of the Sphere.

The Sphere node's geometry requires outside faces only. When viewed from the inside the results are undefined.

6.44 SphereSensor

```
SphereSensor {
  exposedField SFBool     autoOffset        TRUE
  exposedField SFBool     enabled           TRUE
  exposedField SFRotation offset            0 1 0 0   # [-1,1], (-∞,∞)
  eventOut     SFBool     isActive
  eventOut     SFRotation rotation_changed
  eventOut     SFVec3f    trackPoint_changed
}
```

The SphereSensor node maps pointing device motion into spherical rotation about the origin of the local coordinate system. The SphereSensor node uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposed field enables and disables the SphereSensor node. If *enabled* is TRUE, the sensor reacts appropriately to user events. If *enabled* is FALSE, the sensor does not track user input or send 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 node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for details on using the pointing device to activate the SphereSensor.

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 that is used to map subsequent pointing device motion while dragging. The virtual sphere defined by this radius and the local origin at the time of activation is used to interpret subsequent pointing device motion and is not affected by any changes to the sensor's coordinate system while the sensor is active. For each position of the bearing, a *rotation_changed* event is sent which corresponds to the sum of the relative rotation from the original intersection point plus the *offset* value. *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_changed* value and an *offset_changed* event is generated. See 4.6.7.4, 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 is released and generates an *isActive* FALSE event (other pointing-device sensors shall not generate events during this time). Motion of the pointing device while *isActive* is TRUE is termed 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 it is 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 a variety of ways (e.g., clamp all values to the sphere or continue to rotate as the point is dragged away from the sphere). Each movement of the pointing device while *isActive* is TRUE generates *trackPoint_changed* and *rotation_changed* events.

Further information about this behaviour can be found in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

6.45 SpotLight

```
SpotLight {
  exposedField SFFloat ambientIntensity  0         # [0,1]
  exposedField SFVec3f attenuation       1 0 0     # [0,∞)
  exposedField SFFloat beamWidth         1.570796  # (0,π/2]
  exposedField SFColor color             1 1 1     # [0,1]
  exposedField SFFloat cutOffAngle       0.785398  # (0,π/2]
  exposedField SFVec3f direction         0 0 -1    # (-∞,∞)
  exposedField SFFloat intensity         1         # [0,1]
  exposedField SFVec3f location          0 0 0     # (-∞,∞)
  exposedField SFBool  on                TRUE
  exposedField SFFloat radius            100       # [0,∞)
}
```

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

A detailed description of *ambientIntensity*, *color*, *intensity*, and VRML's lighting equations is provided in 4.6.6, Light sources. More information on lighting concepts can be found in 4.14, Lighting model, including a detailed description of the VRML lighting equations.

The *location* field specifies a translation offset of the centre point of the light source from the light's local coordinate system origin. This point is the apex of the solid angle which bounds light emission from the given light source. The *direction* field specifies the direction vector of the light's central axis defined in the local coordinate system.

The *on* field specifies whether the light source emits light. If *on* is TRUE, the light source is emitting light and may illuminate geometry in the scene. If *on* is FALSE, the light source does not emit light and does not illuminate any geometry.

The *radius* field specifies the radial extent of the solid angle and the maximum distance from *location* that may be illuminated by the light source. The light source does not emit light outside this radius. The *radius* shall be greater than or equal to zero.

Both radius and location are affected by ancestors' transformations (scales affect *radius* and transformations affect *location*).

The *cutOffAngle* field specifies the outer bound of the solid angle. The light source does not emit light outside of this solid angle. The *beamWidth* field specifies an inner solid angle in which the light source emits light at uniform full intensity. The light source's emission intensity drops off from the inner solid angle (*beamWidth*) to the outer solid angle (*cutOffAngle*) as described in the following equations:

```
angle = the angle between the Spotlight’s direction vector
        and the vector from the Spotlight location to the point
        to be illuminated

if (angle >= cutOffAngle):
    multiplier = 0
else if (angle <= beamWidth):
    multiplier = 1
else:
    multiplier = (angle - cutOffAngle) / (beamWidth - cutOffAngle)

intensity(angle) = SpotLight.intensity × multiplier
```

If the *beamWidth* is greater than the *cutOffAngle*, *beamWidth* is defined to be equal to the *cutOffAngle* and the light source emits full intensity within the entire solid angle defined by *cutOffAngle*. Both *beamWidth* and *cutOffAngle* shall be greater than 0.0 and less than or equal to $\pi/2$. Figure 6.16 depicts the *beamWidth*, *cutOffAngle*, *direction*, *location*, and *radius* fields of the SpotLight node.

SpotLight illumination falls off with distance as specified by three *attenuation* coefficients. The attenuation factor is $1/max(attenuation[0] + attenuation[1] \times r + attenuation[2] \times r^2 , 1)$, where r is the distance from the light to the surface being illuminated. The default is no attenuation. An *attenuation* value of (0, 0, 0) is identical to (1, 0, 0). Attenuation values shall be greater than or equal to zero. A detailed description of VRML's lighting equations is contained in 4.14, Lighting model.

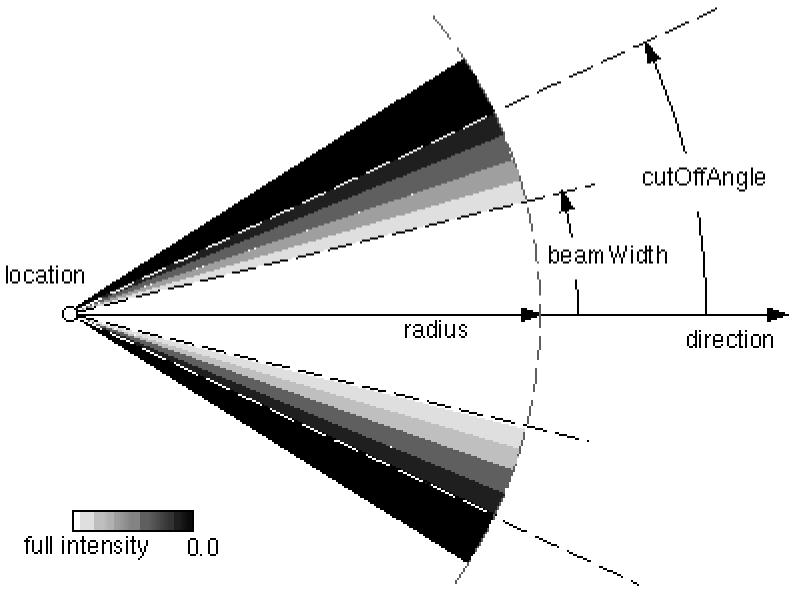


Figure 6.16 -- SpotLight node

6.46 Switch

```
Switch {
  exposedField    MFNode  choice      []
  exposedField    SFInt32 whichChoice -1    # [-1,∞)
}
```

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

4.6.5, Grouping and children nodes, describes details on the types of nodes that are legal values for *choice*.

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

All nodes under a Switch continue to receive and send events regardless of the value of *whichChoice*. For example, if an active TimeSensor is contained within an inactive choice of an Switch, the TimeSensor sends events regardless of the Switch's state.

6.47 Text

```
Text {
  exposedField  MFString string    []
  exposedField  SFNode   fontStyle NULL
  exposedField  MFFloat  length    []       # [0,∞)
  exposedField  SFFloat  maxExtent 0.0      # [0,∞)
}
```

6.47.1 Introduction

The Text node specifies a two-sided, flat text string object positioned in the Z=0 plane of the local coordinate system based on values defined in the fontStyle field (see 6.20, FontStyle). Text nodes may contain multiple text strings specified using the UTF-8 encoding as specified by ISO 10646-1:1993 (see 2.[UTF8]). The text strings are stored in the order in which the text mode characters are to be produced as defined by the parameters in the FontStyle node.

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 used for the text.

The *maxExtent* field limits and compresses all of the text strings if the length of the maximum string is longer than the maximum extent, as measured in the local coordinate system. If the text string with the maximum length is shorter than the *maxExtent*, then there is no compressing. The maximum extent is measured horizontally for horizontal text (FontStyle node: *horizontal*=TRUE) and vertically for vertical text (FontStyle node: *horizontal*=FALSE). The *maxExtent* field shall be greater than or equal to zero.

The *length* field contains an MFFloat value that specifies the length of each text string in the local coordinate system. 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. The *length* field shall be greater than or equal to zero.

Specifying a value of 0 for both the *maxExtent* and *length* fields indicates that the string may be any length.

6.47.2 ISO 10646-1:1993 Character Encodings

Characters in ISO 10646 (see 2.[UTF8]) are encoded in multiple octets. Code space is divided into four units, as follows:

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

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

a. UCS-2 (Universal Coded Character Set-2). This form is also known as the Basic Multilingual Plane (BMP). Characters are encoded in the lower two octets (row and cell).

b. UCS-4 (Universal Coded Character Set-4). Characters are encoded in the full four octets.

In addition, three transformation formats (UCS Transformation Format or UTF) are accepted: UTF-7, UTF-8, and UTF-16. Each represents the nature of the transformation: 7-bit, 8-bit, or 16-bit. UTF-7 and UTF-16 are referenced in 2.[UTF8].

UTF-8 maintains transparency for all 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, the remaining seven bits are interpreted as an ASCII character. Otherwise, the number of leading 1 bits indicates the number of bytes following. There is always a zero bit between the count bits and any data.

The first byte is 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 the format 10XXXXXX.

As a two byte example, the symbol for a register trade mark is &REG; or 174 in ISO Latin-1 (see 2.[I8859]). It is encoded as 0x00AE in UCS-2 of ISO 10646. In UTF-8, it has the following two byte encoding: 0xC2, 0xAE.

6.47.3 Appearance

Textures are applied to text as follows. The texture origin is at the origin of the first string, as determined by the justification. The texture is scaled equally in both S and T dimensions, with the font height representing 1 unit. S increases to the right, and T increases up.

4.14, Lighting model, has details on VRML lighting equations and how Appearance, Material and textures interact with lighting.

The Text node does not participate in collision detection.

6.48 TextureCoordinate

```
TextureCoordinate {
  exposedField MFVec2f point  []      # (-∞,∞)
}
```

The TextureCoordinate node specifies a set of 2D texture coordinates used by vertex-based geometry nodes (e.g., IndexedFaceSet and ElevationGrid) to map textures to vertices. Textures are two dimensional colour functions that, given an *(s, t)* coordinate, return a colour value *colour(s, t)*. Texture map values (ImageTexture, MovieTexture, and PixelTexture) range from [0.0, 1.0] along the S-axis and T-axis. However, TextureCoordinate values, specified by the *point* field, may be in the range (-∞,∞). Texture coordinates identify a location (and thus a colour value) in the texture map. The horizontal coordinate *s* is specified first, followed by the vertical coordinate *t*.

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

```
   Texture map location = (C - floor(C)) × N
```

If the texture map is not repeated, the texture coordinates are clamped to the 0.0 to 1.0 range as follows:

```
    Texture map location = N,     if C > 1.0,
                         = 0.0,   if C < 0.0,
                         = C × N, if 0.0 <= C <= 1.0.
```

Details on repeating textures are specific to texture map node types described in 6.22, ImageTexture, 6.28, MovieTexture, and 6.33, PixelTexture.

6.49 TextureTransform

```
TextureTransform {
  exposedField SFVec2f center      0 0     # (-∞,∞)
  exposedField SFFloat rotation    0       # (-∞,∞)
  exposedField SFVec2f scale       1 1     # (-∞,∞)
  exposedField SFVec2f translation 0 0     # (-∞,∞)
}
```

The TextureTransform node defines a 2D transformation that is applied to texture coordinates (see 6.48, TextureCoordinate). This node affects the way textures coordinates are applied to the geometric surface. The transformation consists of (in order):

a. a translation;

b. a rotation about the centre point;

c. a non-uniform scale about the centre point.

These parameters support changes to the size, orientation, and position of textures on shapes. Note that these operations appear reversed when viewed on the surface of geometry. For example, a *scale* value of (2 2) will scale the texture coordinates and have the net effect of shrinking the texture size by a factor of 2 (texture coordinates are twice as large and thus cause the texture to repeat). A translation of (0.5 0.0) translates the texture coordinates +.5 units along the S-axis and has the net effect of translating the texture -0.5 along the S-axis on the geometry's surface. A rotation of π/2 of the texture coordinates results in a -π/2 rotation of the texture on the geometry.

The *center* field specifies a translation offset in texture coordinate space about which the *rotation* and *scale* fields are applied. The *scale* field specifies a scaling factor in S and T of the texture coordinates about the *center* point. *scale* values shall be in the range (-∞,∞). The *rotation* field specifies a rotation in radians of the texture coordinates about the *center* point after the scale has been applied. A positive rotation value makes the texture coordinates rotate counterclockwise about the centre, thereby rotating the appearance of the texture itself clockwise. The *translation* field specifies a translation of the texture coordinates.

In matrix transformation notation, where *Tc* is the untransformed texture coordinate, *Tc'* is the transformed texture coordinate, *C* (*center*), *T* (*translation*), *R* (*rotation*), and *S* (*scale*) are the intermediate transformation matrices,

```
   Tc' = -C × S × R × C × T × Tc
```

Note that this transformation order is the reverse of the Transform node transformation order since the texture coordinates, not the texture, are being transformed (i.e., the texture coordinate system).

6.50 TimeSensor

```
TimeSensor {
  exposedField SFTime   cycleInterval 1       # (0,∞)
  exposedField SFBool   enabled       TRUE
  exposedField SFBool   loop          FALSE
  exposedField SFTime   startTime     0       # (-∞,∞)
  exposedField SFTime   stopTime      0       # (-∞,∞)
  eventOut     SFTime   cycleTime
  eventOut     SFFloat  fraction_changed      # [0, 1]
  eventOut     SFBool   isActive
  eventOut     SFTime   time
}
```

TimeSensor nodes generate events as time passes. TimeSensor nodes can be used for many purposes including:

a. driving continuous simulations and animations;

b. controlling periodic activities (*e.g.*, one per minute);

c. initiating single occurrence events such as an alarm clock.

The TimeSensor node contains two discrete eventOuts: *isActive* and *cycleTime*. The *isActive* eventOut sends TRUE when the TimeSensor node begins running, and FALSE when it stops running. The *cycleTime* eventOut sends a time event at *startTime* and at the beginning of each new cycle (useful for synchronization with other time-based objects). The remaining eventOuts generate continuous events. The *fraction_changed* eventOut, an SFFloat in the closed interval [0,1], sends the completed fraction of the current cycle. The *time* eventOut sends the absolute time for a given *simulation tick.*

If the *enabled* exposedField is TRUE, the TimeSensor node is enabled and may be running. If a *set_enabled* FALSE event is received while the TimeSensor node is running, the sensor performs the following actions:

d. evaluates and sends all relevant outputs;

e. sends a FALSE value for *isActive*;

f. disables itself.

Events on the exposedFields of the TimeSensor node (e.g., *set_startTime)* are processed and their corresponding eventOuts (e.g., *startTime_changed)* are sent regardless of the state of the *enabled* field. The remaining discussion assumes *enabled* is TRUE.

The *loop*, *startTime*, and *stopTime* exposedFields and the *isActive* eventOut and their effects on the TimeSensor node are discussed in detail in 4.6.9, Time-dependent nodes. The "cycle" of a TimeSensor node lasts for *cycleInterval* seconds. The value of *cycleInterval* shall be greater than zero.

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

When a TimeSensor node becomes active, it generates an *isActive* = TRUE event and begins generating *time*, *fraction_changed*, and *cycleTime* events which may be routed to other nodes to drive animation or simulated behaviours. The behaviour at read time is described below. The *time* event sends the absolute time for a given tick of the TimeSensor node (time fields and events represent the number of seconds since midnight GMT January 1, 1970).

fraction_changed events output a floating point value in the closed interval [0, 1]. At *startTime* the value of *fraction_changed* is 0. After *startTime*, the value of *fraction_changed* in any cycle will progress through the range (0.0, 1.0]. At *startTime* + N × *cycleInterval*, for N = 1, 2, ..., that is, at the end of every cycle, the value of *fraction_changed* is 1.

Let *now* represent the time at the current simulation tick. Then the *time* and *fraction_changed* eventOuts can then be computed as:

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

where `fractionalPart(x)` is a function that returns the fractional part, (that is, the digits to the right of the decimal point), of a nonnegative floating point number.

A TimeSensor node can be set up to be active at read time by specifying *loop* TRUE (not the default) and *stopTime* less than or equal to *startTime* (satisfied by the default values). The *time* events output absolute times for each tick of the TimeSensor node simulation. The *time* events shall start at the first simulation tick greater than or equal to *startTime*. *time* events end at *stopTime*, or at *startTime* + *N* × *cycleInterval* for some positive integer value of *N*, or loop forever depending on the values of the other fields. An active TimeSensor node shall stop at the first simulation tick when *now* >= *stopTime* > *startTime*.

No guarantees are made with respect to how often a TimeSensor node generates time events, but a TimeSensor node shall generate events at least at every simulation tick. TimeSensor nodes are guaranteed to generate final *time* and *fraction_changed* events. If loop is FALSE at the end of the *N*th cycleInterval and was TRUE at *startTime* + *M* × *cycleInterval* for all *0* < *M* < *N*, the final *time* event will be generated with a value of (*startTime* + *N* × *cycleInterval*) or *stopTime* (if *stopTime* > *startTime*), whichever value is less. If *loop* is TRUE at the completion of every cycle, the final event is generated as evaluated at *stopTime* (if *stopTime* > *startTime*) or never.

An active TimeSensor node ignores *set_cycleInterval* and *set_startTime* events. An active TimeSensor node also ignores *set_stopTime* events for *set_stopTime* less than or equal to *startTime*. For example, if a *set_startTime* event is received while a TimeSensor node is active, that *set_startTime* event is ignored (the *startTime* field is not changed, and a *startTime_changed* eventOut is not generated). If an active TimeSensor node receives a *set_stopTime* event that is less than the current time, and greater than *startTime*, it behaves as if the *stopTime* requested is the current time and sends the final events based on the current time (note that *stopTime* is set as specified in the eventIn).

A TimeSensor read from a VRML file shall generate *isActive* TRUE, *time* and *fraction_changed* events if the sensor is enabled and all conditions for a TimeSensor to be active are met.

6.51 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 node tracks the location and state of the pointing device and detects when the user points at geometry contained by the TouchSensor node's parent group. A TouchSensor node can be enabled or disabled by sending it an *enabled* event with a value of TRUE or FALSE. If the TouchSensor node is disabled, it does not track user input or send events.

The TouchSensor generates events when the pointing device points toward any geometry nodes that are descendants of the TouchSensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for more details on using the pointing device to activate the TouchSensor.

The *isOver* eventOut reflects the state of the pointing device with regard to whether it is pointing towards the TouchSensor node's geometry or not. When the pointing device changes state from a position such that its bearing does not intersect any of the TouchSensor node's geometry to one in which it does intersect geometry, an *isOver* TRUE event is generated. When the pointing device moves from a position such that its bearing intersects geometry to one in which it no longer intersects the geometry, or some other geometry is obstructing the TouchSensor node's geometry, an *isOver* FALSE event is generated. These events are generated only when the pointing device has moved and changed `over' state. Events are not generated if the geometry itself is animating and moving underneath the pointing device.

As the user moves the bearing over the TouchSensor node'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 node'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*. The values of *hitTexCoord_changed* and *hitNormal_changed* events are computed as appropriate for the associated shape.

If *isOver* is TRUE, the user may activate the pointing device to cause the TouchSensor node to generate *isActive* events (e.g., by pressing the primary mouse button). When the TouchSensor node generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is released 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 termed a "drag." If a 2D pointing device is in use, *isActive* events 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 it is released). If a 3D pointing device is in use, *isActive* events will typically reflect whether the pointing device is within (or in contact with) the TouchSensor node's geometry.

The eventOut field *touchTime* is generated when all three of the following conditions are true:

a. The pointing device was pointing towards the geometry when it was initially activated (*isActive* is TRUE).

b. The pointing device is currently pointing towards the geometry (*isOver* is TRUE).

c. The pointing device is deactivated (*isActive* FALSE event is also generated).

More information about this behaviour is described in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

6.52 Transform

```
Transform {
  eventIn      MFNode      addChildren
  eventIn      MFNode      removeChildren
  exposedField SFVec3f     center           0 0 0    # (-∞,∞)
  exposedField MFNode      children         []
  exposedField SFRotation  rotation         0 0 1 0  # [-1,1], (-∞,∞)
  exposedField SFVec3f     scale            1 1 1    # (0,∞)
  exposedField SFRotation  scaleOrientation 0 0 1 0  # [-1,1], (-∞,∞)
  exposedField SFVec3f     translation      0 0 0    # (-∞,∞)
  field        SFVec3f     bboxCenter       0 0 0    # (-∞,∞)
  field        SFVec3f     bboxSize         -1 -1 -1 # (0,∞) or -1,-1,-1
}
```

The Transform node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors. See 4.4.4, Transformation hierarchy, and 4.4.5, Standard units and coordinate system, for a description of coordinate systems and transformations.

4.6.5, Grouping and children nodes, provides a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the children of the Transform node. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and, if needed, shall be calculated by the browser. The bounding box shall be large enough at all times to enclose the union of the group's children's bounding boxes; it shall not include any transformations performed by the group itself (i.e., the bounding box is defined in the local coordinate system of the children). The results are undefined if the specified bounding box is smaller than the true bounding box of the group. A description of the *bboxCenter* and *bboxSize* fields is provided in 4.6.4, Bounding boxes.

The *translation*, *rotation*, *scale*, *scaleOrientation* and *center* fields define a geometric 3D transformation consisting of (in order):

a. a (possibly) non-uniform scale about an arbitrary point;

b. a rotation about an arbitrary point and axis;

c. a translation.

The *center* field specifies a translation offset from the origin of the local coordinate system (0,0,0). The *rotation* field specifies a rotation of the coordinate system. The *scale* field specifies a non-uniform scale of the coordinate system. *scale* values shall be greater than zero. The *scaleOrientation* specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The *scaleOrientation* applies only to the scale operation. The *translation* field specifies a translation to the coordinate system.

Given a 3-dimensional point **P** and Transform node, **P** is transformed into point **P'** in its parent's coordinate system by a series of intermediate transformations. In matrix transformation notation, where C (*center*), SR (*scaleOrientation*), T (*translation*), R (*rotation*), and S (*scale*) are the equivalent transformation matrices,

```
    P' = T × C × R × SR × S × -SR × -C × P
```

The following Transform node:

```
Transform {
    center           C
    rotation         R
    scale            S
    scaleOrientation SR
    translation      T
    children         [...]
}
```

is equivalent to the nested sequence of:

```
Transform {
  translation T
  children Transform {
    translation C
    children Transform {
      rotation R
      children Transform {
        rotation SR
        children Transform {
          scale S
          children Transform {
            rotation -SR
            children Transform {
              translation -C
              children [...]
}}}}}}}
```

6.53 Viewpoint

```
Viewpoint {
  eventIn      SFBool     set_bind
  exposedField SFFloat    fieldOfView    0.785398  # (0,π)
  exposedField SFBool     jump           TRUE
  exposedField SFRotation orientation    0 0 1 0   # [-1,1], (-∞,∞)
  exposedField SFVec3f    position       0 0 10    # (-∞,∞)
  field        SFString   description    ""
  eventOut     SFTime     bindTime
  eventOut     SFBool     isBound
}
```

The Viewpoint node defines a specific location in the local coordinate system from which the user may view the scene. Viewpoint nodes are bindable children nodes (see 4.6.10, Bindable children nodes) and thus there exists a Viewpoint node stack in the browser in which the top-most Viewpoint node on the stack is the currently active Viewpoint node. If a TRUE value is sent to the *set_bind* eventIn of a Viewpoint node, it is moved to the top of the Viewpoint node stack and activated. When a Viewpoint node is at the top of the stack, the user's view is conceptually re-parented as a child of the Viewpoint node. All subsequent changes to the Viewpoint node's coordinate system change the user's view (e.g., changes to any ancestor transformation nodes or to the Viewpoint node's *position* or *orientation* fields). Sending a *set_bind* FALSE event removes the Viewpoint node from the stack and produces *isBound* FALSE and *bindTime* events. If the popped Viewpoint node is at the top of the viewpoint stack, the user's view is re-parented to the next entry in the stack. More details on binding stacks can be found in 4.6.10, Bindable children nodes. When a Viewpoint node is moved to the top of the stack, the existing top of stack Viewpoint node sends an *isBound* FALSE event and is pushed down the stack.

An author can automatically move the user's view through the world by binding the user to a Viewpoint node and then animating either the Viewpoint node or the transformations above it. Browsers shall allow the user view to be navigated relative to the coordinate system defined by the Viewpoint node (and the transformations above it) even if the Viewpoint node or its ancestors' transformations are being animated.

The *bindTime* eventOut sends the time at which the Viewpoint node is bound or unbound. This can happen:

a. during loading;

b. when a *set_bind* event is sent to the Viewpoint node;

c. when the browser binds to the Viewpoint node through its user interface described 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. In the default position and orientation, the viewer is on the Z-axis looking down the -Z-axis toward the origin with +X to the right and +Y straight up. Viewpoint nodes are affected by the transformation hierarchy.

Navigation types (see 6.29, NavigationInfo) that require a definition of a down vector (e.g., terrain following) shall use the negative Y-axis of the coordinate system of the currently bound Viewpoint node. Likewise, navigation types that require a definition of an up vector shall use the positive Y-axis of the coordinate system of the currently bound Viewpoint node. The *orientation* field of the Viewpoint node does not affect the definition of the down or up vectors. This allows the author to separate the viewing direction from the gravity direction.

The *jump* field specifies whether the user's view "jumps" to the position and orientation of a bound Viewpoint node or remains unchanged. This jump is instantaneous and discontinuous in that no collisions are performed and no ProximitySensor nodes are checked in between the starting and ending jump points. If the user's position before the jump is inside a ProximitySensor the *exitTime* of that sensor shall send the same timestamp as the bind eventIn. Similarly, if the user's position after the jump is inside a ProximitySensor the *enterTime* of that sensor shall send the same timestamp as the bind eventIn. Regardless of the value of *jump* at bind time, the relative viewing transformation between the user's view and the current Viewpoint node shall be stored with the current Viewpoint node for later use when *un-jumping* (i.e., popping the Viewpoint node binding stack from a Viewpoint node with *jump* TRUE*)*. The following summarizes the bind stack rules (see 4.6.10, Bindable children nodes) with additional rules regarding Viewpoint nodes (displayed in boldface type):

d. During read, the first encountered Viewpoint node is bound by pushing it to the top of the Viewpoint node stack. If a Viewpoint node name is specified in the URL that is being read, this named Viewpoint node is considered to be the first encountered Viewpoint node. Nodes contained within Inline nodes, within the strings passed to the Browser.createVrmlFromString() method, or within files passed to the Browser.createVrmlFromURL() method (see 4.12.10, Browser script interface) are not candidates for the first encountered Viewpoint node. The first node within a prototype instance is a valid candidate for the first encountered Viewpoint node. The first encountered Viewpoint node sends an *isBound* TRUE event.

e. When a *set_bind* TRUE event is received by a Viewpoint node,

   1. If it is not on the top of the stack: **The relative transformation from the current top of stack Viewpoint node to the user's view is stored with the current top of stack Viewpoint node.** The current top of stack node sends an *isBound* FALSE event. The new node is moved to the top of the stack and becomes the currently bound Viewpoint node. The new Viewpoint node (top of stack) sends an *isBound* TRUE event. **If *jump* is TRUE for the new Viewpoint node, the user's view is**

instantaneously "jumped" to match the values in the *position* and *orientation* fields of the new Viewpoint node.

2. If the node is already at the top of the stack, this event has no affect.

f. When a *set_bind* FALSE event is received by a Viewpoint node in the stack, it is removed from the stack. If it was on the top of the stack,

   1. it sends an *isBound* FALSE event,

   2. the next node in the stack becomes the currently bound Viewpoint node (i.e., pop) and issues an *isBound* TRUE event,

   3. **if its *jump* field value is TRUE, the user's view is instantaneously "jumped" to the *position* and *orientation* of the next Viewpoint node in the stack with the stored relative transformation of this next Viewpoint node applied.**

g. If a *set_bind* FALSE event is received by a node not in the stack, the event is ignored and *isBound* events are not sent.

h. When a node replaces another node at the top of the stack, the *isBound* TRUE and FALSE events from the two nodes are sent simultaneously (i.e., with identical timestamps).

i. If a bound node is deleted, it behaves as if it received a *set_bind* FALSE event (see c.).

The *jump* field may change after a Viewpoint node is bound. The rules described above still apply. If *jump* was TRUE when the Viewpoint node is bound, but changed to FALSE before the *set_bind* FALSE is sent, the Viewpoint node does not *un-jump* during unbind. If *jump* was FALSE when the Viewpoint node is bound, but changed to TRUE before the *set_bind* FALSE is sent, the Viewpoint node does perform the *un-jump* during unbind.

Note that there are two other mechanisms that result in the binding of a new Viewpoint:

j. An Anchor node's *url* field specifies a "#ViewpointName".

k. A script invokes the `loadURL()` method and the URL argument specifies a "#ViewpointName".

Both of these mechanisms override the *jump* field value of the specified Viewpoint node (#ViewpointName) and assume that *jump* is TRUE when binding to the new Viewpoint. The behaviour of the viewer transition to the newly bound Viewpoint depends on the currently bound NavigationInfo node's *type* field value (see 6.29, NavigationInfo).

The *fieldOfView* field specifies a preferred minimum viewing angle from this viewpoint in radians. A small field of view roughly corresponds to a telephoto lens; a large field of view roughly corresponds to a wide-angle lens. The field of view shall be greater than zero and smaller than π. The value of *fieldOfView* represents the minimum viewing angle in any direction axis perpendicular to the view. For example, a browser with a rectangular viewing projection shall have the following relationship:

$$\frac{\text{display width}}{\text{display height}} = \frac{\tan(FOV_{horizontal}/2)}{\tan(FOV_{vertical}/2)}$$

where the smaller of display width or display height determines which angle equals the *fieldOfView* (the larger angle is computed using the relationship described above). The larger angle shall not exceed π and may force the smaller angle to be less than *fieldOfView* in order to sustain the aspect ratio.

The *description* field specifies a textual description of the Viewpoint node. This may be used by browser-specific user interfaces. If a Viewpoint's *description* field is empty it is recommended that the browser not present this Viewpoint in its browser-specific user interface.

The URL syntax "`.../scene.wrl#ViewpointName`" specifies the user's initial view when loading "scene.wrl" to be the first Viewpoint node in the VRML file that appears as **`DEF ViewpointName Viewpoint {...}`**. This overrides the first Viewpoint node in the VRML file as the initial user view, and a *set_bind* TRUE message is sent to the Viewpoint node named "ViewpointName". If the Viewpoint node named "ViewpointName" is not found, the browser shall use the first Viewpoint node in the VRML file (i.e. the normal default behaviour). The URL syntax "`#ViewpointName`" (i.e. no file name) specifies a viewpoint within the existing VRML file. If this URL is loaded (e.g. Anchor node's *url* field or `loadURL()` method is invoked by a Script node), the Viewpoint node named "ViewpointName" is bound (a *set_bind* TRUE event is sent to this Viewpoint node).

The results are undefined if a Viewpoint node is bound and is the child of an LOD, Switch, or any node or prototype that disables its children. If a Viewpoint node is bound that results in collision with geometry, the browser shall perform its self-defined navigation adjustments as if the user navigated to this point (see 6.8, Collision).

6.54 VisibilitySensor

```
VisibilitySensor {
  exposedField SFVec3f center   0 0 0      # (-∞,∞)
  exposedField SFBool  enabled  TRUE
  exposedField SFVec3f size     0 0 0      # [0,∞)
  eventOut     SFTime  enterTime
  eventOut     SFTime  exitTime
  eventOut     SFBool  isActive
}
```

The VisibilitySensor node 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 in order to activate or deactivate some behaviour or animation. The purpose is often to attract the attention of the user or to improve performance.

The *enabled* field enables and disables the VisibilitySensor node. If *enabled* is set to FALSE, the VisibilitySensor node does not send events. If *enabled* is TRUE, the VisibilitySensor node 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. A FALSE event is sent when the box has no effect on the view. Browsers shall guarantee that, if *isActive* is FALSE, the box has absolutely no effect on the rendered view. Browsers may err liberally when *isActive* is TRUE. For example, the box may affect the rendering.

The exposed fields *center* and *size* specify the object space location of the box centre and the extents of the box (i.e., width, height, and depth). The VisibilitySensor node's box is affected by hierarchical transformations of its parents. The components of the *size* field shall be greater than or equal to zero.

The *enterTime* event is generated whenever the *isActive* TRUE event is generated, and *exitTime* events are generated whenever *isActive* FALSE events are generated. A VisibilitySensor read from a VRML file shall generate *isActive* TRUE and *enterTime* events if the sensor is enabled and the visibility box is visible. A VisibilitySensor inserted into the transformation hierarchy shall generate *isActive* TRUE and *enterTime* events if the sensor is enabled and the visibility box is visible. A VisibilitySensor removed from the transformation hierarchy shall generate *isActive* FALSE and *exitTime* events if the sensor is enabled and the visibility box is visible.

Each VisibilitySensor node behaves independently of all other VisibilitySensor nodes. Every enabled VisibilitySensor node that is affected by the user's movement receives and sends events, possibly resulting in multiple VisibilitySensor nodes receiving and sending events simultaneously. Unlike TouchSensor nodes, there is no notion of a VisibilitySensor node lower in the scene graph "grabbing" events. Multiply instanced VisibilitySensor nodes (i.e., DEF/USE) use the union of all the boxes defined by their instances. An instanced VisibilitySensor node shall detect visibility changes for all instances of the box and send events appropriately.

6.55 WorldInfo

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

The WorldInfo node contains information about the world. This node is strictly for documentation purposes and has no effect on the visual appearance or behaviour of the world. The *title* field is intended to store the name or title of the world so that browsers can present this to the user (perhaps in the window border). Any other information about the world can be stored in the *info* field, such as author information, copyright, and usage instructions.

7 Conformance and minimum support requirements

7.1 Introduction

7.1.1 Table of contents

7.1.2 Objectives

This clause addresses conformance of VRML files, VRML generators and VRML browsers.

The primary objectives of the specifications in this clause are:

a. to promote interoperability by eliminating arbitrary subsets of, or extensions to, ISO/IEC 14772;

b. to promote uniformity in the development of conformance tests;

c. to promote consistent results across VRML browsers;

d. to facilitate automated test generation.

7.1.3 Scope

Conformance is defined for VRML files and for VRML browsers. For VRML generators, conformance guidelines are presented for enhancing the likelihood of successful interoperability.

A concept of *base profile conformance* is defined to ensure interoperability of VRML generators and VRML browsers. Base profile conformance is based on a set of limits and minimal requirements. Base profile conformance is intended to provide a functional level of reasonable utility for VRML generators while limiting the complexity and resource requirements of VRML browsers. Base profile conformance may not be adequate for all uses of VRML.

This clause addresses the VRML data stream and implementation requirements. Implementation requirements include the latitude allowed for VRML generators and VRML browsers. This clause does not directly address the environmental, performance, or resource requirements of the generator or browser.

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 VRML files

A VRML file is *syntactically correct* according to ISO/IEC 14772 if the following conditions are met:

a. The VRML file contains as its first element a VRML header comment (see 4.2.2, Header).

b. All entities contained therein match the functional specification of the corresponding entities of ISO/IEC 14772-1. The VRML file shall obey the relationships defined in the formal grammar and all other syntactic requirements.

c. The sequence of entities in the VRML file obeys the relationships specified in ISO/IEC 14772-1 producing the structure specified in ISO/IEC 14772-1.

d. All field values in the VRML file obey the relationships specified in ISO/IEC 14772-1 producing the structure specified in ISO/IEC 14772-1.

e. No nodes appear in the VRML file other than those specified in ISO/IEC 14772-1 unless required for the encoding technique or those defined by the PROTO or EXTERNPROTO entities.

f. The VRML file is encoded according to the rules of ISO/IEC 14772.

g. It does not contain behaviour described as undefined elsewhere in this specification.

A VRML file conforms to the *base profile* if:

h. It is syntactically correct.

i. It meets the restrictions of Table 7.1.

7.2.2 Conformance of VRML generators

A VRML generator is conforming to this part of ISO/IEC 14772 if all VRML files that are generated are syntactically correct.

A VRML generator conforms to the base profile if it can be configured such that all VRML files generated conform to the base profile.

7.2.3 Conformance of VRML browsers

A VRML browser conforms to the base profile if:

a. It is able to read any VRML file that conforms to the base profile.

b. It presents the graphical and audio characteristics of the VRML nodes in any VRML file that conforms to the base profile, within the latitude defined in this clause.

c. It correctly handles user interaction and generation of events as specified in ISO/IEC 14772, within the latitude defined in this clause.

d. It satisfies the requirements of 7.3.2, Minimum support requirements for browsers, as enumerated in Table 7.1.

7.3 Minimum support requirements

7.3.1 Minimum support requirements for generators

There is no minimum complexity which is required of (or appropriate for) VRML generators. Any compliant set of nodes of arbitrary complexity may be generated, as appropriate to represent application content.

7.3.2 Minimum support requirements for browsers

This subclause defines the minimum complexity which shall be supported by a VRML browser. Browser implementations may choose to support greater limits but may not reduce the limits described in Table 7.1. When the VRML file contains nodes which exceed the limits implemented by the browser, the results are undefined. Where latitude is specified in Table 7.1 for a particular node, full support is required for other aspects of that node.

7.3.3 VRML requirements for conforming to the base profile

In the following table, the first column defines the item for which conformance is being defined. In some cases, general limits are defined but are later overridden in specific cases by more restrictive limits. The second column defines the requirements for a VRML file conforming to the base profile; if a VRML file contains any items that exceed these limits, it may not be possible for a VRML browser conforming to the base profile to successfully parse that VRML file. The third column defines the minimum complexity for a VRML scene that a VRML browser conforming to the base profile shall be able to present to the user. The word "ignore" in the minimum browser support column refers only to the display of the item; in particular, *set_* events to ignored exposedFields must still generate corresponding *_changed* events.

Table 7.1 -- Specifications for VRML browsers conforming to the base profile

Item	**VRML File Limit**	**Minimum Browser Support**
All groups	500 children.	500 children. Ignore *bboxCenter* and *bboxSize*.
All interpolators	1000 key-value pairs.	1000 key-value pairs.
All lights	8 simultaneous lights.	8 simultaneous lights.
Names for DEF/PROTO/field	50 utf8 octets.	50 utf8 octets.
All *url* fields	10 URLs.	10 URLs. URN's ignored. Support `http', `file', and `ftp' protocols. Support relative URLs where relevant.
PROTO/ EXTERNPROTO	30 fields, 30 eventIns, 30 eventOuts, 30 exposedFields.	30 fields, 30 eventIns, 30 eventOuts, 30 exposedFields.
EXTERNPROTO	n/a	URL references VRML files conforming to the base profile
PROTO definition nesting depth	5 levels.	5 levels.
SFBool	No restrictions.	Full support.
SFColor	No restrictions.	Full support.
SFFloat	No restrictions.	Full support.
SFImage	256 width. 256 height.	256 width. 256 height.
SFInt32	No restrictions.	Full support.
SFNode	No restrictions.	Full support.
SFRotation	No restrictions.	Full support.
SFString	30,000 utf8 octets.	30,000 utf8 octets.
SFTime	No restrictions.	Full support.
SFVec2f	15,000 values.	15,000 values.
SFVec3f	15,000 values.	15,000 values.

MFColor	15,000 values.	15,000 values.
MFFloat	1,000 values.	1,000 values.
MFInt32	20,000 values.	20,000 values.
MFNode	500 values.	500 values.
MFRotation	1,000 values.	1,000 values.
MFString	30,000 utf8 octets per string, 10 strings.	30,000 utf8 octets per string, 10 strings.
MFTime	1,000 values.	1,000 values.
MFVec2f	15,000 values.	15,000 values.
MFVec3f	15,000 values.	15,000 values.
Anchor	No restrictions.	Ignore *parameter*. Ignore *description.*
Appearance	No restrictions.	Full support.
AudioClip	30 second uncompressed PCM WAV.	30 second uncompressed PCM WAV. Ignore *description*.
Background	No restrictions.	One *skyColor*, one *groundColor*, panorama images as per ImageTexture.
Billboard	Restrictions as for all groups.	Full support except as for all groups.
Box	No restrictions.	Full support.
Collision	Restrictions as for all groups.	Full support except as for all groups. Any navigation behaviour acceptable when collision occurs.
Color	15,000 colours.	15,000 colours.
ColorInterpolator	Restrictions as for all interpolators.	Full support except as for all interpolators.
Cone	No restrictions.	Full support.
Coordinate	15,000 points.	15,000 points.
CoordinateInterpolator	15,000 coordinates per	15,000 coordinates per *keyValue*. Support as for all

	keyValue. Restrictions as for all interpolators.	interpolators.
Cylinder	No restrictions.	Full support.
CylinderSensor	No restrictions.	Full support.
DirectionalLight	No restrictions.	Not scoped by parent Group or Transform.
ElevationGrid	16,000 heights.	16,000 heights.
Extrusion	(#*crossSection* points)*(#*spine* points) <= 2,500.	(#*crossSection* points)*(#*spine* points) <= 2,500.
Fog	No restrictions.	"EXPONENTIAL" treated as "LINEAR"
FontStyle	No restrictions.	If the values of the text aspects character set, *family*, *style* cannot be simultaneously supported, the order of precedence shall be: 1) character set 2) *family* 3) *style*. Browser must display all characters in ISO 8859-1 character set 2.[I8859].
Group	Restrictions as for all groups.	Full support except as for all groups.
ImageTexture	JPEG and PNG format. Restrictions as for PixelTexture.	JPEG and PNG format. Support as for PixelTexture.
IndexedFaceSet	10 vertices per face. 5000 faces. Less than 15,000 indices.	10 vertices per face. 5000 faces. 15,000 indices in any index field.
IndexedLineSet	15,000 total vertices. 15,000 indices in any index field.	15,000 total vertices. 15,000 indices in any index field.
Inline	No restrictions.	Full support except as for all groups. *url* references VRML files conforming to the base profile
LOD	Restrictions as for all groups.	At least first 4 *level*/*range* combinations interpreted, and support as for all groups. Implementations may disregard *level* distances.
Material	No restrictions.	Ignore ambient intensity. Ignore specular colour. Ignore emissive colour. One-bit transparency; transparency values >= 0.5 transparent.

MovieTexture	MPEG1-Systems and MPEG1-Video formats.	MPEG1-Systems and MPEG1-Video formats. Display one active movie texture. Ignore *speed* field.
NavigationInfo	No restrictions.	Ignore *avatarSize*. Ignore *visibilityLimit*.
Normal	15,000 normals	15,000 normals
NormalInterpolator	15,000 normals per *keyValue*. Restrictions as for all interpolators.	15,000 normals per *keyValue*. Support as for all interpolators.
OrientationInterpolator	Restrictions as for all interpolators.	Full support except as for all interpolators.
PixelTexture	256 width. 256 height.	256 width. 256 height. Display fully transparent and fully opaque pixels.
PlaneSensor	No restrictions.	Full support.
PointLight	No restrictions.	Ignore *radius*. Linear attenuation.
PointSet	5000 points.	5000 points.
PositionInterpolator	Restrictions as for all interpolators.	Full support except as for all interpolators.
ProximitySensor	No restrictions.	Full support.
ScalarInterpolator	Restrictions as for all interpolators.	Full support except as for all interpolators.
Script	25 eventIns. 25 eventOuts. 25 fields.	25 eventIns. 25 eventOuts. 25 fields. No scripting language support required.
Shape	No restrictions.	Full support.
Sound	No restrictions.	2 active sounds. Linear distance attenuation. No spatialization. See 7.3.4.
Sphere	No restrictions.	Full support.
SphereSensor	No restrictions.	Full support.
SpotLight	No restriction	Ignore *beamWidth*. Ignore *radius*. Linear attenuation.
Switch	Restrictions as for all groups.	Full support except as for all groups.

Text	100 characters per string. 100 strings.	100 characters per string. 100 strings.
TextureCoordinate	15,000 coordinates.	15,000 coordinates.
TextureTransform	No restrictions.	Full support.
TimeSensor	No restrictions.	Ignored if *cycleInterval* < 0.01 second.
TouchSensor	No restrictions.	Full support.
Transform	Restrictions as for all groups.	Full support except as for all groups.
Viewpoint	No restrictions.	Ignore *fieldOfView*. Ignore *description*.
VisibilitySensor	No restrictions.	Always visible.
WorldInfo	No restrictions.	Ignored.

7.3.4 Sound priority, attenuation, and spatialization

7.3.4.1 Sound priority

If the browser does not have the resources 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 in the order specified:

a. decreasing *priority;*

b. for sounds with *priority* > 0.5, increasing (now-*startTime*);

c. decreasing *intensity* at viewer location (*intensity* &times; intensity attenuation);

where *priority* is the *priority* field of the Sound node, now represents the current time, *startTime* is the *startTime* field of the audio source node specified in the *source* field, and intensity attenuation refers to the intensity multiplier derived from the linear decibel attenuation ramp between inner and outer ellipsoids.

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 browser is "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 at the location of the viewer).

The browser shall play as many sounds from the beginning of this sorted list as it can given available resources and allowable latency between rendering. On most systems, the resources available for MIDI streams are different from those for playing sampled sounds, thus it may be beneficial to maintain a separate list to handle MIDI data.

7.3.4.2 Sound attenuation and spatialization

In order to create a linear decrease in loudness as the viewer moves from the inner to the outer ellipsoid of the sound, the attenuation must be based on a linear decibel ramp. To make the falloff consistent across browsers, the decibel ramp is to vary from 0 dB at the minimum ellipsoid to -20 dB at the outer ellipsoid. Sound nodes with an outer ellipsoid that is ten times larger than the minimum will display the inverse square intensity dropoff that approximates sound attenuation in an anechoic environment.

Browsers may support spatial localization of sounds whose *spatialize* field is TRUE as well as their underlying sound libraries will allow. Browsers shall at least support stereo panning of non-MIDI sounds based on the angle between the viewer and the source. This angle is obtained by projecting the Sound *location* (in global space) onto the XZ plane of the viewer. Determine the angle between the Z-axis and the vector from the viewer to the transformed *location*, and assign a pan value in the range [0.0, 1.0] as depicted in Figure 7.1. Given this pan value, left and right channel levels can be obtained using the following equations:

$$\text{leftPanFactor} = 1 - \text{pan}^2$$

$$\text{rightPanFactor} = 1 - (1 - \text{pan})^2$$

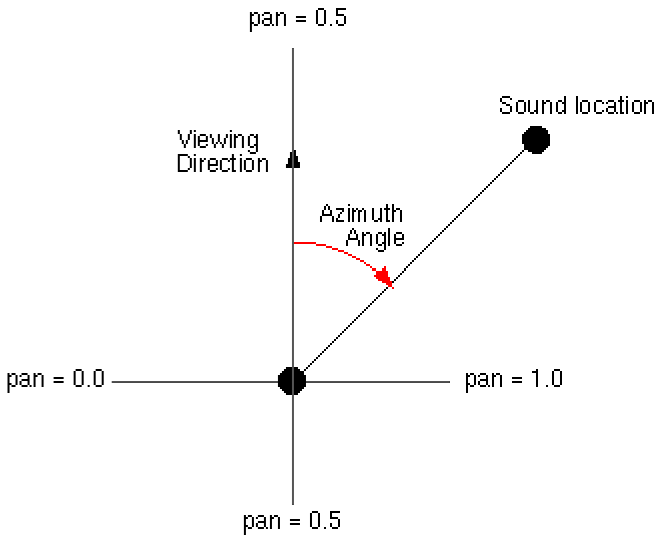


Figure 7.1: Stereo Panning

Using this technique, the loudness of the sound is modified by the *intensity* field value, then distance attenuation to obtain the unspatialized audio output. The values in the unspatialized audio output are then scaled by leftPanFactor and rightPanFactor to determine the final left and right output signals. The use of more sophisticated localization techniques is encouraged, but not required (see E.[SNDB]).

Annex A
(normative)

Grammar definition

A.1 Table of contents and introduction

A.1.1 Table of contents

This annex provides a detailed description of the grammar for each syntactic element in this part of ISO/IEC 14772. The following table of contents lists the topics in this clause:

A.1.2 Introduction

It is not possible to parse VRML files using a context-free grammar. Semantic knowledge of the names and types of fields, eventIns, and eventOuts for each node type (either built-in or user-defined using **PROTO** or **EXTERNPROTO**) shall 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 the first line of the VRML file or quoted SFString or MFString fields. The '#' character and all characters until the next line terminator comprise the comment and are treated as whitespace.

The carriage return (0x0d), linefeed (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. All reserved keywords are displayed in boldface type.

Any characters (including linefeed and '#') may appear within the quotes of SFString and MFString fields. A double quote character within a string shall be preceded with a backslash (e.g, "Each double quotes character \" shall have a backslash."). A backslash character within a string shall be preceded with a backslash forming two backslashes (e.g., "One backslash \\ character").

Clause 6, Nodes reference, contains a description of the allowed fields, eventIns and eventOuts for all pre-defined node types. The *double*, *float*, and *int32* symbols are expressed using Perl regular expression syntax; see E.[PERL] for details. The *IdFirstChar*, *IdRestChars*, and *string* symbols have not been formally specified; Clause 5, Fields and events reference, contains a more complete description of their syntax.

The following conventions are used in the semi-formal grammar specified in this clause:

a. Keywords and terminal symbols which appear literally in the VRML file, are specified in **bold**.

b. Nonterminal symbols used in the grammar are specified in *italic*.

c. Production rules begin with a nonterminal symbol and the sequence of characters "::=", and end with a semi-colon (";").

d. Alternation for production rules is specified using the vertical-bar symbol ("|").

Table A.1 contains the complete list of lexical elements for the grammar in this part of ISO/IEC 14772.

Table A.1 -- VRML lexical elements

Keywords	Terminal symbols	Other symbols
DEF **EXTERNPROTO** **FALSE** **IS** **NULL** **PROTO** **ROUTE** **TO** **TRUE** **USE** **eventIn** **eventOut** **exposedField** **field**	period (**.**) open brace (**{**) close brace (**}**) open bracket (**[**) close bracket (**]**)	*Id* *double* *fieldType* *float* *int32* *string*

Terminal symbols and the string symbol may be separated by one or more whitespace characters. Keywords and the *Id*, *fieldType*, *float*, *int32*, and *double* symbols shall be separated by one or more whitespace characters.

A.2 General

vrmlScene ::=
    statements ;

statements ::=
    statement |
    statement statements |
    empty ;

statement ::=
    nodeStatement |
    protoStatement |
    routeStatement ;

nodeStatement ::=
    node |
    DEF *nodeNameId node |*
    USE *nodeNameId ;*

rootNodeStatement ::=
    node | **DEF** *nodeNameId node ;*

protoStatement ::=
    proto |
    externproto ;

protoStatements ::=
    protoStatement |
    protoStatement protoStatements |
    empty ;

proto ::=
    PROTO *nodeTypeId* **[** *interfaceDeclarations* **] {** *protoBody* **}** *;*

protoBody ::=
    protoStatements rootNodeStatement statements ;

interfaceDeclarations ::=
    interfaceDeclaration |
    interfaceDeclaration interfaceDeclarations |
    empty ;

restrictedInterfaceDeclaration ::=
    eventIn *fieldType eventInId* |
    eventOut *fieldType eventOutId* |
    field *fieldType fieldId fieldValue ;*

interfaceDeclaration ::=
    restrictedInterfaceDeclaration |
    exposedField *fieldType fieldId fieldValue ;*

externproto ::=
    EXTERNPROTO *nodeTypeId* **[** *externInterfaceDeclarations* **]** *URLList ;*

externInterfaceDeclarations ::=
    externInterfaceDeclaration |
    externInterfaceDeclaration externInterfaceDeclarations |
    empty ;

externInterfaceDeclaration ::=
    eventIn *fieldType eventInId* |
    eventOut *fieldType eventOutId* |
    field *fieldType fieldId* |
    exposedField *fieldType fieldId ;*

routeStatement ::=
    ROUTE *nodeNameId* **.** *eventOutId* **TO** *nodeNameId* **.** *eventInId ;*

URLList ::=
    mfstringValue ;

empty ::=
    ;

A.3 Nodes

node ::=
  nodeTypeId **{** *nodeBody* **}** |
  Script { *scriptBody* **}** *;*

nodeBody ::=
  nodeBodyElement |
  nodeBodyElement nodeBody |
  empty ;

scriptBody ::=
  scriptBodyElement |
  scriptBodyElement scriptBody |
  empty ;

scriptBodyElement ::=
  nodeBodyElement |
  restrictedInterfaceDeclaration |
  eventIn *fieldType eventInId* **IS** *eventInId* |
  eventOut *fieldType eventOutId* **IS** *eventOutId* |
  field *fieldType fieldId* **IS** *fieldId ;*

nodeBodyElement ::=
  fieldId fieldValue |
  fieldId **IS** *fieldId* |
  eventInId **IS** *eventInId* |
  eventOutId **IS** *eventOutId* |
  routeStatement |
  protoStatement ;

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, 0x2b, 0x2c, 0x2d, 0x2e, 0x5b, 0x5c, 0x5d, 0x7b, 0x7d, 0x7f ;

IdRestChars ::=
  Any number of ISO-10646 characters except: 0x0-0x20, 0x22, 0x23, 0x27, 0x2c, 0x2e, 0x5b, 0x5c, 0x5d, 0x7b, 0x7d, 0x7f ;

A.4 Fields

fieldType ::=
MFColor |
MFFloat |
MFInt32 |
MFNode |
MFRotation |
MFString |
MFTime |
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 |
mftimeValue |
mfvec2fValue |
mfvec3fValue ;

sfboolValue ::=
TRUE |
FALSE ;

sfcolorValue ::=
float float float ;

sffloatValue ::=
float ;

float ::=
  ([+/-]?((([0-9]+(\.)?)|([0-9]*\.[0-9]+))([eE][+\-]?[0-9]+)?)).

sfimageValue ::=
  int32 int32 int32 ...

sfint32Value ::=
  int32 ;

int32 ::=
  ([+\-]?(([0-9]+)|(0[xX][0-9a-fA-F]+)))

sfnodeValue ::=
  nodeStatement |
  NULL *;*

sfrotationValue ::=
  float float float float ;

sfstringValue ::=
  string ;

string ::=
  ".*" ... double-quotes must be \", backslashes must be \\...

sftimeValue ::=
  double ;

double ::=
  ([+/-]?((([0-9]+(\.)?)|([0-9]*\.[0-9]+))([eE][+\-]?[0-9]+)?))

mftimeValue ::=
  sftimeValue |
  [ ] |
  [*sftimeValues* **]** *;*

sftimeValues ::=
  sftimeValue |
  sftimeValue sftimeValues ;

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 ::=
    nodeStatement |
    [ ] |
    [*nodeStatements* **]** *;*

nodeStatements ::=
    nodeStatement |
    nodeStatement nodeStatements ;

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 *;*

Annex B
(normative)

Java platform scripting reference

B.1 Introduction

This annex describes the Java platform classes and methods that enable Script nodes (see 6.40, Script) to interact with VRML scenes. See 4.12, Scripting, for a general description of scripting languages in ISO/IEC 14772. Note that support for the Java platform is not required by ISO/IEC 14772, but any access of the Java platform from within VRML Script nodes shall conform with the requirements specified in this annex.

B.2 Platform

The Javatm platform is an object-oriented, hardware and operating system independent, multi-threaded, general-purpose application environment developed by Sun Microsystems, Inc. The Java platform consists of the language, the virtual machine, and a set of core class libraries. A conforming Java platform implements all three components according to their specifications. See 2.[JAVA] for a description of the language, the virtual machine, and the three core classes java.lang, java.util, and java.io. The other core class libraries, which are not used in this annex, are described in E.[JAPI].

B.3 Supported protocol in the script node's *url* field

B.3.1 *url* field

The *url* field of the Script node may contain URL references to Java bytecode as illustrated below:

```
Script {
  url "http://foo.co.jp/Example.class"
  eventIn SFBool start
}
```

B.3.2 File extension

The file extension for Java bytecode is `.class`.

B.3.3 MIME type

The MIME type for Java bytecode is defined as follows:

```
application/x-java
```

B.4 EventIn handling

B.4.1 Description

Events to the Script node are passed to the corresponding Java platform method (processEvents() or processEvent()) in the script. The script is specified in the *url* field of the Script node.

For a Java bytecode file specified in the *url* field, the following three conditions hold:

a. it shall contain the class definition whose name is exactly the same as the body of the file name

b. it shall be a subclass of the Script class (see B.9.2.3, vrml.node package)

c. it shall be declared as a "public" class

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

```
Script {
  url "http://foo.co.jp/Example1.class"
  eventIn SFBool start
}
```

This node points to the script file Example1.class. Its source (Example1.java) looks like this:

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

public class Example1 extends Script {
    ...
    // This method is called when any event is received
    public void processEvent(Event e){
       // ... perform some operation ...
    }
}
```

In the above example, when the *start* eventIn is sent the processEvent() method receives the eventIn and is executed.

B.4.2 Parameter passing with Event objects

When a Script node receives an eventIn, a processEvent() or processEvents() method in the file specified in the url field of the Script node is called, which receives the eventIn as a Java platform object (Event object, see B.4.3, processEvents() and processEvent() methods).

The Event object has three fields of information associated with it: name, value, and timestamp, whose values are passed by the eventIn. These can be retrieved using the corresponding method on the Event object.

```
public class Event implements Cloneable {
   public String getName();
   public ConstField getValue();
   public double getTimeStamp();
   // other methods ...
}
```

Suppose that the eventIn type is SFXXX and eventIn name is eventInYYY, then

a. getName() shall return the string "eventInYYY "

b. getValue() shall return ConstField containing the value of the eventIn

c. getTimeStamp() shall return a double (in seconds) containing the timestamp when the eventIn occurred (see 4.11, Time)

In the example below, the eventIn name is *start* and the eventIn value is cast to ConstSFBool. Also, the timestamp for the time when the eventIn occurred is available as a double. These are passed as an Event object to the processEvent() method:

```
public void processEvent(Event e){
    if(e.getName().equals("start")){
          ConstSFBool v = (ConstSFBool)e.getValue();
          if(v.getValue()==true){
               // ... perform some operation with e.getTimeStamp()...
          }
     }
}
```

B.4.3 processEvents() and processEvent() methods

B.4.3.1 processEvents() method

Authors can define a processEvents() method within a class that is called when the script receives some set of events. The prototype of the processEvents() method is **`public void processEvents(int count, Event events[]);`**

count indicates the number of events delivered. *events* is the array of events delivered. Its default behaviour is to iterate over each event, calling processEvent() on each one as follows:

```
public void processEvents(int count, Event events[])
{
    for (int i = 0; i < count; i++){
        processEvent(events[i]);
    }
}
```

Although authors might change this operation by giving a user-defined processEvents() method, in most cases, they only change the processEvent() method and the eventsProcessed() method as described below.

When multiple eventIns are routed from a single node to a single Script node and eventIns which have the same timestamp are received, processEvents() receives multiple events as an event array. Otherwise, each incoming event invokes separate processEvents().

For example, the processEvents() method receives two events in the following case, when the TouchSensor is activated:

```
Transform {
  children [
    DEF TS TouchSensor {}
    Shape { geometry Cone {} }
  ]
}
DEF SC Script {
  url "Example.class"
  eventIn SFBool isActive
  eventIn SFTime touchTime
```

```
    }
    ROUTE TS.isActive  TO SC.isActive
    ROUTE TS.touchTime TO SC.touchTime
```

B.4.3.2 processEvent() method

Authors can define a processEvent() method within a class. The prototype of the processEvent() is **`public void processEvent(Event event);`**

Its default behaviour is no operation.

B.4.4 eventsProcessed() method

Authors may define an eventsProcessed() method within a class that is called after some set of events has been received. This allows Script nodes that do not rely on the ordering of events received to generate fewer events than an equivalent Script node that generates events whenever events are received (see B.4.3.1, processEvents() method).

The prototype of the eventsProcessed() method is **`public void eventsProcessed();`**

Its default behaviour is no operation.

B.4.5 shutdown() method

Authors may define a shutdown() method within the Script class that is called when the corresponding Script node is deleted or the world containing the Script node is unloaded or replaced by another world (see 4.12.3, Initialize() and shutdown()).

The prototype of the shutdown() method is **`public void shutdown();`**

Its default behaviour is no operation.

B.4.6 initialize() method

Authors may define an initialize() method within the Script class that is called before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script (see 4.12.3, Initialize() and shutdown()). The various methods of the Script class such as getEventIn(), getEventOut(), getExposedField(), and getField() are not guaranteed to return correct values before the initialize() method has been executed. The initialize() method is called once during the life of the Script object.

The prototype of the initialize() method is **`public void initialize();`**

Its default behaviour is no operation. See Example2.java in B.5.1 for an example of a user-specified initialize() method.

B.5 Accessing fields and events

B.5.1 Accessing fields, eventIns and eventOuts of the script

The fields, eventIns, and eventOuts of a Script node are accessible from its corresponding Script class. Each field defined in the Script node is available to the Script class by using its name. Its value can be read-from or written-into. This value is persistent across function calls. EventOuts defined in the Script node can be read. EventIns defined in the Script node can be written to.

Accessing the fields of the Script node can be done by using the following three types of Script class methods:

a. **`Field getField(String fieldName)`**
is the method to get the reference to the Script node's field whose name is *fieldName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

b. **`Field getEventOut(String eventOutName)`**
is the method to get the reference to the Script node's eventOut whose name is *eventOutName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

c. **`Field getEventIn(String eventInName)`**
is the method to get the reference to the Script node's eventIn whose name is *eventInName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class). EventIn is a write-only field. When the getValue() method is invoked on a Field object obtained by the getEventIn() method, the return value is unspecified.

When the setValue(), set1Value(), addValue(), insertValue(), delete() or clear() methods are invoked on a Field object obtained by the **getField()** method, the new value is stored in the corresponding VRML node's field (see also B.6.2, Field class and ConstField class, and B.6.3, Array handling). In the case of the set1Value(), addValue(), insertValue() or delete() methods, all elements of the VRML node's field are retrieved, then the value specified as an argument is set, added, inserted, deleted (as appropriate) to/from the elements, and then stored as the elements in the corresponding VRML node's field. In the case of the clear() method, all elements of a VRML node's field are cleared (see the definition of the clear() method).

When the setValue(), set1Value(), addValue(), insertValue(), delete() or clear() methods are invoked on a Field object obtained by the **getEventOut()** method, the call generates an eventOut in the VRML scene (see also B.6.2, Field class and ConstField class, and B.6.3, Array handling). The effect of this eventOut is specified by the associated Route(s) in the VRML scene. In the case of the set1Value(), addValue(), insertValue() or delete() methods, all elements of the VRML node's eventOut are retrieved, then the value specified as an argument is set, added, inserted or deleted (as appropriate) to/from the elements, then stored as the elements in the corresponding VRML node's eventOut, and then the eventOut is sent. In the case of the clear() method, all elements of VRML node's eventOut are cleared and an eventOut with zero elements is sent (see the definition of the clear() method).

When the setValue() or clear() methods are invoked on a Field object obtained by the **getEventIn()** method, the call generates an eventIn to the Script node. When the set1Value(), addValue(), insertValue() or delete() methods are invoked on a Field object obtained by the getEventIn() method, the exception (InvalidFieldChangeException) is thrown.

For example, the following Script node (Example2) defines an eventIn *start*, a field *state*, and an eventOut *on*. The method initialize() is invoked before any events are received, and the method processEvent() is invoked when *start* receives an event:

```
    Script {
```

```
    url      "Example2.class"
    eventIn  SFBool start
    field    SFBool state TRUE
    eventOut SFBool on
  }
```

Example2.java:

```
// Example2 toggles a persistent field variable "state" in the VRML
// Script node each time an eventIn "start" is received, then sets
// eventOut "on" equal to the value of "state"
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Example2 extends Script {
    private SFBool state; // field
    private SFBool on;    // eventOut

    public void initialize(){
        state = (SFBool) getField("state");
        on = (SFBool) getEventOut("on");
    }

    public void processEvent(Event e){
        if(state.getValue()==true){
            on.setValue(false); // set false to eventOut 'on'
            state.setValue(false);
        }
        else {
            on.setValue(true);  // set true to eventOut 'on'
            state.setValue(true);
        }
    }
}
```

B.5.2 Accessing fields, eventIns and eventOuts of other nodes

If a script program has an access to a node, any eventIn, eventOut or exposedField of that node is accessible by using the getEventIn(), getEventOut() or getExposedField() method defined in the node's class (see B.6.4, Node class).

The typical way for a Script node to have an access to another VRML node is to have an SFNode field which provides a reference to the other node. The following Example3 shows how this is done:

```
DEF SomeNode Transform {}
Script {
  field SFNode node USE SomeNode # SomeNode is a Transform node
  eventIn SFVec3f pos            # new value to be inserted in
                                 #   SomeNode's translation field
  url "Example3.class"
}
```

Example3.java:

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

public class Example3 extends Script {
    private SFNode node;   // field
    private SFVec3f trans; // translation field captured from remote
                           // Transform node

    public void initialize(){
        node = (SFNode) getField("node");
    }

    public void processEvent(Event e){
        // get the reference to the 'translation' field of the Transform node
        trans = (SFVec3f)((Node) node.getValue()).getExposedField("translation");
        // reset translation to value given in Event e, which is eventIn pos
        // in the VRML Script node.
        trans.setValue((ConstSFVec3f)e.getValue());
    }
}
```

B.5.3 Sending eventOuts or eventIns

Assume that the thread which executes processEvent() (or processEvents()) is called 'main' thread and any other thread spawned by the Script, except for the 'main' thread, is called 'sub' thread. Sending eventOuts/eventIns in the 'main' thread follows the model described in 4.10.3, Execution model, and sending eventOuts/eventIns in any 'sub' thread follows the model described in 4.12.6, Asynchronous scripts.

In the 'main' thread: Calling one of the setValue(), set1Value, addValue(), insertValue(), clear() or delete() methods on an eventOut/eventIn sends that event at that time. Calling the methods multiple times during one execution of the thread still only sends one event which corresponds to the first call of the method. All other calls are ignored. The event is assigned the same timestamp as the initial event which caused the main thread to execute.

In the 'sub' thread: Calling one of the setValue(), set1Value, addValue(), insertValue(), clear() or delete() method on an eventOut/eventIn sends that event at that time. Calling the methods multiple times during one execution of the thread sends one event per call of the method. The browser assigns the timestamp to the event.

Note: sending eventIns is ordinarily performed by the VRML scene, not by Java platform scripts. Exceptions are possible as specified in B.5.1, Accessing fields, eventIns and eventOuts of the script.

B.6 Exposed classes and methods for nodes and fields

B.6.1 Introduction

Java platform classes for VRML are defined in the packages: *vrml*, *vrml.node* and *vrml.field*.

The Field class extends the Java platform'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 platform type value. A setValue() method converts a Java platform type value into a VRML type value and sets it to the VRML field.

Some methods are listed as "throws exception," meaning that errors are possible. It may be necessary to write exception handlers (using the Java platform's **catch()** method) when those methods are used. Any method not listed as "throws exception" is guaranteed to generate no exceptions. Each method that throws an exception includes a prototype showing which exception(s) can be thrown.

B.6.2 Field class and ConstField class

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

```
    public abstract class Field implements Cloneable {
        // methods
    }
```

This class has two types of subclasses: read-only classes and read/write classes

a. **Read-only classes**
   These classes support the getValue() method. Some classes support additional convenience methods to get value(s) from the object.

   ConstSFBool, ConstSFColor, ConstMFColor, ConstSFFloat, ConstMFFloat, ConstSFImage, ConstSFInt32, ConstMFInt32, ConstSFNode, ConstMFNode, ConstSFRotation, ConstMFRotation, ConstSFString, ConstMFString, ConstSFVec2f, ConstMFVec2f, ConstSFVec3f, ConstMFVec3f, ConstSFTime, ConstMFTime

b. **Read/write classes**
   These classes support both getValue() and setValue() methods. If the class name is prefixed with **MF** (meaning that it is a multiple valued field class), the class also supports the set1Value(), addValue() and insertValue() methods. Some classes support additional convenience methods to get and set value(s) from the object.

   SFBool, SFColor, MFColor, SFFloat, MFFloat, SFImage, SFInt32, MFInt32, SFNode, MFNode, SFRotation, MFRotation, SFString, MFString, SFVec2f, MFVec2f, SFVec3f, MFVec3f, SFTime, MFTime

The VRML Field class and its subclasses have several methods to get and set value(s): getSize(), getValue(), get1Value(), setValue(), set1Value(), addValue(), insertValue(), clear(), delete() and toString(). In these methods, getSize(), get1Value(), set1Value(), addValue(), insertValue(), clear() and delete() are only available for multiple value field classes (MF classes).

c. **`getSize()`**
   is the method to return the number of elements of each multiple value field class (MF class).

d. **`getValue()`**
   is the method to convert a VRML type value into a Java platform type value and return it.

e. **`get1Value(int index)`**
   is the method to convert a single VRML type value (*index*-th element of an array) and return it as a single Java platform type value. The index of the first element is 0. Attempting to get an element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

f. **`setValue(value)`**
   is the method to convert a Java platform type *value* into a VRML type value and copy it to the target object.

g. **`set1Value(int index, value)`**
   is the method to convert from a Java platform type *value* to a VRML type value and copy it to the *index*-th element of the target object. The index of the first element is 0. Attempting to set an element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

h. **`addValue(value)`**
   is the method to convert from a Java platform type *value* to a VRML type value and append it to the target object, thus adding an element.

i. **`insertValue(int index, value)`**
   is the method to convert from a Java platform type *value* to a VRML type value and insert it as a new element at the *index*-th position, thus adding an element. The index of the first element is 0. Attempting to insert the element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

j. **`clear()`**
is the method to clear all elements in the target object so that it has no more elements in it.

k. **`delete(int index)`**
is the method to delete the *index*-th element from the target object, thus decreasing the length of the element array by one. The index of the first element is 0. Attempting to delete the element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

l. **`toString()`**
is the method to return a String containing the VRML utf8 encoded value (or values) of the equivalent of the field. In the case of the SFNode(ConstSFNode) and MFNode (ConstMFNode),

- **SFNode(ConstSFNode)**: the method returns the VRML utf8 string that, if parsed as the value of an SFNode field, would produce this node. If the browser is unable to reproduce this node, the name of the node followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings.

- **MFNode(ConstMFNode)**: the method returns the VRML utf8 string that, if parsed as the value of a MFNode field, would produce this array of nodes. If the browser is unable to reproduce this node, the name of the nodes followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings

See also B.5.1, Accessing fields, eventIns and eventOuts of the Script, B.6.3, Array handling, B.6.4, Node class, and B.9.2.1, vrml package, for each class' methods definition.

B.6.3 Array handling

B.6.3.1 Format

Some constructors and other methods of the field classes take an array as an argument.

a. **A single-dimensional array**

Some constructors and other methods of the following classes take a single-dimensional array as an argument. The array is treated as follows:

1. **ConstSFColor, ConstMFColor, SFColor and MFColor**
```
float colors[]
```
colors[] consists of a set of three float-values (representing red, green and blue).

2. **ConstSFRotation, ConstMFRotation, SFRotation and MFRotation**
```
float rotations[]
```
rotations[] consists of a set of four float-values (representing axisX, axisY, axisZ and angle).

3. **ConstSFVec2f, ConstMFVec2f, SFVec2f and MFVec2f**
```
float vec2s[]
```
vec2s[] consists of a set of two float-values (representing x and y).

4. **ConstSFVec3f, ConstMFVec3f, SFVec3f and MFVec3f**
```
float vec3s[]
```

vec3s[] consists of a set of three float-values (representing x, y and z).

5. **ConstSFImage and SFImage**

   ```
   byte pixels[]
   ```

   pixels[] consists of 2-dimensional pixel image. The ordering of the individual components for an individual pixel within the array of bytes will be as follows:

# Comp.	byte[i]	byte[i + 1]	byte[i + 2]	byte[i + 3]
1	intensity1	intensity2	intensity3	intensity4
2	intensity1	alpha1	intensity2	alpha2
3	red1	green1	blue1	red2
4	red1	green1	blue1	alpha1

   The order of pixels in the array are to follow that defined in 5.5, SFImage. byte 0 is pixel 0, starting from the bottom left corner.

b. **A single integer and a single-dimensional array**

Some constructors and other methods take a single integer value (called *size*) and a single-dimensional array as arguments: for example, MFFloat(int size, float values[]). The *size* parameter specifies the number of valid elements in the array, from 0-th element to (*size* - 1)-th element, all other values are ignored. This means that the method may be passed an array of length *size* or larger. The same rule for a single-dimensional array is applied to the valid elements.

c. **An array of arrays**

Some constructors and other methods alternatively take an array of arrays as an argument. The array is treated as follows:

1. **ConstMFColor and MFColor**

   ```
   float colors[][]
   ```

   colors[][] consists of an array of sets of three float-values (representing red, green and blue).

2. **ConstMFRotation and MFRotation**

   ```
   float rotations[][]
   ```

   rotations[][] consists of an array of sets of four float-values (representing axisX, axisY, axisZ and angle).

3. **ConstMFVec2f and MFVec2f**

   ```
   float vec2s[][]
   ```

   vec2s[][] consists of an array of sets of two float-values (representing x and y).

4. **ConstMFVec3f and MFVec3f**

   ```
   float vec3s[][]
   ```

   vec3s[][] consists of an array of sets of three float-values (representing x, y and z).

B.6.3.2 Constructors and methods

The following describes how arrays are interpreted in detail for each constructor and method.

Suppose *NA* represents the number of elements in the array specified as an argument of some constructors and other methods, and *NT* represents the number of elements which the target object requires or has. For example, if the target object is SFColor, it requires exactly 3 float values.

In the following description, suppose SF* represents subclasses of Field class, ConstSF* represents subclasses of ConstField class, MF* represents subclasses of MField class and ConstMF* represents subclasses of ConstMField class.

a. **A single-dimensional array**

In the following description, if the target object is:

- ConstSFColor and SFColor, *NT* is exactly 3
- ConstMFColor and MFColor, *NT* is a multiple of 3, and *NA* is rounded down to a multiple of 3
- ConstSFRotation and SFRotation, *NT* is exactly 4
- ConstMFRotation and MFRotation, *NT* is a multiple of 4, and *NA* is rounded down to a multiple of 4
- ConstSFVec2f and SFVec2f, *NT* is exactly 2
- ConstMFVec2f and MFVec2f, *NT* is a multiple of 2, and *NA* is rounded down to a multiple of 2
- ConstSFVec3f and SFVec3f, *NT* is exactly 3
- ConstMFVec3f and MFVec3f, *NT* is a multiple of 3, and *NA* is rounded down to a multiple of 3
- ConstSFImage and SFImage, *NT* is exactly *width***height***components* (*width*, *height* and number of *components* in the image, see 5.5, SFImage)

1. **For ConstSF* objects and SF* objects**

   For all constructors and methods which take a single-dimensional array as an argument, the following rules are applied. *NA* shall be larger than or equal to *NT*. If *NA* is larger than *NT*, the elements from the 0-th to the (*NT* - 1)-th element are used and remaining elements are ignored. Otherwise, an exception(ArrayIndexOutOfBoundsException) is thrown.

   For example, when the array is used as an argument of the setValue() for SFColor, the array shall contain at least 3 float values. If the array contains more than 3 float values, the first 3 values are used.

2. **For ConstMF* objects and MF* objects**

- **For constructor.**

  The same rule for ConstSF* and SF* objects is applied.

  For example, when the array is used as an argument of the constructor for MFColor, the array shall contain at least 3 float values. If the array contains 3N, 3N +1 or 3N + 2 float values, the first 3N values are used.

- **For setValue() method.**

  If *NT* is smaller than or equal to *NA*, *NT* is increased to *NA* and then all elements of the array are copied into the target object. If *NT* is larger than *NA*, *NT* is decreased to *NA* and then all elements of the array are copied into the target object.

- **For getValue() method.**

  If *NT* is smaller than or equal to *NA*, all elements of the target object are copied into the first *NT* elemets of the array. If *NT* is larger than *NA*, an exception (ArrayIndexOutOfBoundsException) is thrown.

- F**or set1Value() method.**

  The target element (the *index*-th element) is treated as an SF* object. So the same rule for ConstSF* and SF* objects is applied.

- **For get1Value() method.**

  The target element (the *index*-th element) is treated as an SF* (or ConstSF*) object. So the same rule for ConstSF* and SF* objects is applied.

- **For addValue() and insertValue() method.**

  The corresponding SF* object is created using the argument, and then added to the target object or inserted into the target object.

b. **A single integer and a single-dimensional array**

For all constructors and methods which take a single integer value (called *size*) and a single-dimensional array as arguments, for example, MFFloat(int size, float values[]), the following rule is applied.

The *size* parameter specifies the number of valid elements in the array from the 0-th element to the (*size* - 1)-th element; all other values are ignored. This means that the method may be passed an array of length *size* or larger.

The valid elements are copied to a new array and the rules for a single-dimensional array are applied to the new array for all methods.

c. **An array of arrays**

This argument is used only for MF* objects and ConstMF* objects. In the following case, suppose *NA* is the number of arrays (for example float f[4][3], *NA* is 4) specified as an argument of some constructors and other methods and *NT* is the return value of getSize() method of each object.

- **For constructor.**
  The object which has *NA* elements is created.

- **For setValue() method.**
  If *NT* is smaller than or equal to *NA*, *NT* is increased to *NA* and then all elements of the array are copied into the target object. If *NT* is larger than *NA*, *NT* is decreased to *NA* and then all elements of the array are copied into the target object.

- **For getValue() method.**
  If *NT* is smaller than or equal to *NA*, all elements of the target object are copied into the array. If *NT* is larger than *NA*, an exception(ArrayIndexOutOfBoundsException) is thrown.

B.6.4 Node class

The *Node* class has several methods:

a. **`String getType()`**
is the method to return the type of the node.

b. **`ConstField getEventOut(String eventOutName)`**
is the method to get the reference to the node's eventOut whose name is *eventOutName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

c. **`Field getEventIn(String eventInName)`**
is the method to get the reference to the node's eventIn whose name is *eventInName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class). EventIn is a write-only field. When the getValue() method is invoked on a Field object obtained by the getEventIn() method, the return value is unspecified.

d. **`Field getExposedField(String exposedFieldName)`**
is the method to get the reference to the node's exposedField whose name is *exposedFieldName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

e. **`Browser getBrowser()`**
is the method to get the browser object that this node is contained in (see B.6.5, Browser class).

f. **`String toString()`**
is the same as the toString() method of SFNode (ConstSFNode).

When the setValue(), set1Value(), addValue(), insertValue(), delete() or clear() methods are invoked on a Field object obtained by the **getExposedField()** method, the call generates an eventOut in the VRML scene (see also B.6.2, Field class and ConstField class, and B.6.3, Array handling). The effect of this eventOut is specified by the associated Route(s) in the VRML scene. In the case of the set1Value(), addValue(), insertValue() or delete() methods, all elements of the VRML node's exposedField are retrieved, then the value specified as an argument is set, added, inserted or deleted (as appropriate) to/from the elements, then stored as the elements in the corresponding VRML node's exposedField, and then the eventOut is sent. In the case of the clear() method, all elements of VRML node's exposedField are cleared and an eventOut with zero elements is sent (see the definition of the clear() method).

When the setValue() or clear() methods are invoked on a Field object obtained by the **getEventIn()** method, the call generates an eventIn in the VRML scene. When the set1Value(), addValue(), insertValue() or delete() methods are invoked on the Field object, an exception (InvalidFieldChangeException) is thrown.

B.6.5 Browser class

This section lists the public Java platform interfaces to the *Browser* class, which allows scripts to get and set browser information. For descriptions of the following methods, see 4.12.10, Browser script interface. Table B.1 lists the Browser class methods.

Table B.1 -- Browser class methods

Return value	Method name
`String`	`getName()`
`String`	`getVersion()`
`float`	`getCurrentSpeed()`
`float`	`getCurrentFrameRate()`
`String`	`getWorldURL()`
`void`	`replaceWorld(BaseNode[] nodes)`
`BaseNode[]`	`createVrmlFromString(String vrmlSyntax)`
`void`	`createVrmlFromURL(String[] url, BaseNode node, String event)`
`void`	`addRoute(BaseNode fromNode, String fromEventOut, BaseNode toNode, String toEventIn)`
`void`	`deleteRoute(BaseNode fromNode, String fromEventOut, BaseNode toNode, String toEventIn)`
`void`	`loadURL(String[] url, String[] parameter)`
`void`	`setDescription(String description)`

See B.9.2.1, vrml package, for each method's definition.

Table B.2 contains conversions from the types used in Browser class to Java platform types.

Table B.2 -- VRML and Java platform types

VRML type	Java platform type
SFString	String
SFFloat	float
MFString	String[]
MFNode	BaseNode[]

When a relative URL is specified as an argument of the loadURL() and createVrmlFromURL() method, the path is relative to the script file containing these methods (see 4.5.3, Relative URLs).

B.6.6 User-defined classes and packages

The Java platform classes defined by a user can be used in the Java program. They are first searched from the directories specified in the CLASSPATH environment variable and then the directory where the Java program's class file is placed.

If the Java platform class is in a package, this package is searched from the directories specified in the CLASSPATH environment variable and then the directory where the Java program's class file is placed.

B.6.7 Standard Java platform packages

Java programs have access to the full set of classes available in `java.*`. All parts of the Java platform are required to work as "normal" for the Java platform. So all methods specified in this annex are required to be thread-safe. The security model is browser specific.

B.7 Exceptions

Java platform methods may throw the following exceptions:

a. **InvalidFieldException**
   is thrown at the time getField() is executed and the field name is invalid.

b. **InvalidEventInException**
   is thrown at the time getEventIn() is executed and the eventIn name is invalid.

c. **InvalidEventOutException**
   is thrown at the time getEventOut() is executed and the eventOut name is invalid.

d. **InvalidExposedFieldException**
   is thrown at the time getExposedField() is executed and the exposedField name is invalid.

e. **InvalidVRMLSyntaxException**
   is thrown at the time createVrmlFromString(), createVrmlFromURL() or loadURL() is executed and the vrml syntax is invalid.

f. **InvalidRouteException**
   is thrown at the time addRoute() or deleteRoute() is executed and one or more of the arguments is invalid.

g. **InvalidFieldChangeException**
   may be thrown as a result of all sorts of illegal field changes, for example:

   1. Adding a node from one World as the child of a node in another World.

   2. Creating a circularity in a scene graph.

   3. Setting an invalid string on enumerated fields, such as the fogType field of the Fog node.

4. Calling the set1Value(), addValue() or delete() on a Field object obtained by the getEventIn() method.

h. **ArrayIndexOutOfBoundsException**
is generated at the time getValue(), set1Value(), insertValue() or delete() is executed and the index is out of bound (see B.6.2, Field class and ConstField class). This is the standard exception defined in the Java platform Array class.

i. **IllegalArgumentException**
is generated at the time loadURL() or createVrmlFromURL() is executed and an error is occurred before retrieving the content of the url (see B.6.5, Browser class). This is the standard exception defined by the Java platform.

If exceptions are not caught by authors, a browser's behaviour is unspecified (see B.10, Example of exception class).

B.8 Examples

The following is an example of a Script node which determines whether a given color contains a lot of red. The Script node exposes a field, an eventIn, and an eventOut:

```
Script {
  field    SFColor currentColor 0 0 0
  eventIn  SFColor colorIn
  eventOut SFBool  isRed
  url      "Example4.class"
}
```

The following is the source code for the Example4.java file that gets called every time an eventIn is routed to the above Script node:

Example4.java:

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

public class Example4 extends Script {
    // Declare field(s)
    private SFColor currentColor;

    // Declare eventOut
    private SFBool isRed;

    // buffer for  SFColor.getValue().
    private float colorBuff[] = new float[3];

    public void initialize(){
       currentColor = (SFColor) getField("currentColor");
       isRed = (SFBool) getEventOut("isRed");
    }

    public void processEvent(Event e){
        // This method is called when a colorIn event is received
        currentColor.setValue((ConstSFColor)e.getValue());
    }
```

```
        public void eventsProcessed(){
            currentColor.getValue(colorBuff);
            if (colorBuff[0] >= 0.5) // if red is at or above 50%
                isRed.setValue(true);
        }
    }
```

Details on when the methods defined in Example4.java are called may be found in 4.10.3, Execution model.

Example5: createVrmlFromUrl()

```
    Script {
      url "Example5.class"
      field   MFString target_url "foo.wrl"
      eventIn MFNode   nodesLoaded
      eventIn SFBool   trigger_event
    }
```

Example5.java:

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

  public class Example5 extends Script {
      private MFString target_url; // field
      private Browser browser;

      public void initialize(){
          target_url = (MFString)getField("target_url");
          browser = this.getBrowser();
      }

      public void processEvent(Event e){
          if(e.getName().equals("trigger_event")){
               // do something and then fetch values
               String[] urls;
               urls = new String[target_url.getSize()];
               target_url.getValue(urls);
               browser.createVrmlFromURL(urls, this, "nodesLoaded");
          }
          if(e.getName().equals("nodesLoaded")){
              // do something
          }
      }
  }
```

Example6: addRoute()

```
    DEF TS TouchSensor {}
    Script {
      url     "Example6.class"
      field   SFNode fromNode USE TS
      eventIn SFBool clicked
      eventIn SFBool trigger_event
    }
```

Example6.java:

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

public class Example6 extends Script {

    private SFNode fromNode;
    private Browser browser;

    public void initialize(){
        fromNode = (SFNode) getField("fromNode");
        browser = this.getBrowser();
    }

    public void processEvent(Event e){
        if(e.getName().equals("trigger_event")){
            // do something and then add routing
            browser.addRoute(fromNode.getValue(), "isActive", this, "clicked");
        }
        if(e.getName().equals("clicked")){
            // do something
        }
    }
}
```

B.9 Class definitions

B.9.1 Class hierarchy

The classes are divided into three packages: vrml, vrml.field and vrml.node.

```
java.lang.Object
      |
      +- vrml.Event
      +- vrml.Browser
      +- vrml.Field
      |       +- vrml.field.SFBool
      |       +- vrml.field.SFColor
      |       +- vrml.field.SFFloat
      |       +- vrml.field.SFImage
      |       +- vrml.field.SFInt32
      |       +- vrml.field.SFNode
      |       +- vrml.field.SFRotation
      |       +- vrml.field.SFString
      |       +- vrml.field.SFTime
      |       +- vrml.field.SFVec2f
      |       +- vrml.field.SFVec3f
      |       |
      |       +- vrml.MField
      |       |       +- vrml.field.MFColor
      |       |       +- vrml.field.MFFloat
      |       |       +- vrml.field.MFInt32
      |       |       +- vrml.field.MFNode
```

```
     |       |       +- vrml.field.MFRotation
     |       |       +- vrml.field.MFString
     |       |       +- vrml.field.MFTime
     |       |       +- vrml.field.MFVec2f
     |       |       +- vrml.field.MFVec3f
     |       |
     |       +- vrml.ConstField
     |               +- vrml.field.ConstSFBool
     |               +- vrml.field.ConstSFColor
     |               +- vrml.field.ConstSFFloat
     |               +- vrml.field.ConstSFImage
     |               +- vrml.field.ConstSFInt32
     |               +- vrml.field.ConstSFNode
     |               +- vrml.field.ConstSFRotation
     |               +- vrml.field.ConstSFString
     |               +- vrml.field.ConstSFTime
     |               +- vrml.field.ConstSFVec2f
     |               +- vrml.field.ConstSFVec3f
     |               |
     |               +- vrml.ConstMField
     |                       +- vrml.field.ConstMFColor
     |                       +- vrml.field.ConstMFFloat
     |                       +- vrml.field.ConstMFInt32
     |                       +- vrml.field.ConstMFNode
     |                       +- vrml.field.ConstMFRotation
     |                       +- vrml.field.ConstMFString
     |                       +- vrml.field.ConstMFTime
     |                       +- vrml.field.ConstMFVec2f
     |                       +- vrml.field.ConstMFVec3f
     |
     +- vrml.BaseNode
             +- vrml.node.Node
             +- vrml.node.Script

java.lang.Exception
     |
     +- java.lang.RuntimeException
     |       +- java.lang.IllegalArgumentException
     |               +- vrml.InvalidEventInException
     |               +- vrml.InvalidEventOutException
     |               +- vrml.InvalidExposedFieldException
     |               +- vrml.InvalidFieldChangeException
     |               +- vrml.InvalidFieldException
     |               +- vrml.InvalidRouteException
     |
     +- vrml.InvalidVRMLSyntaxException
```

B.9.2 VRML packages

B.9.2.1 vrml package

```
package vrml;

public class Event implements Cloneable
{
   public String getName();
   public double getTimeStamp();
   public ConstField getValue();
   public Object clone();

   public String toString();   // This overrides a method in Object
}

public class Browser
{
   private Browser();
   public String toString();   // This overrides a method in Object

   // Browser interface
   public String getName();
   public String getVersion();

   public float getCurrentSpeed();

   public float getCurrentFrameRate();

   public String getWorldURL();
   public void replaceWorld(BaseNode[] nodes);

   public BaseNode[] createVrmlFromString(String vrmlSyntax)
     throws InvalidVRMLSyntaxException;

   public void createVrmlFromURL(String[] url, BaseNode node, String event)
     throws InvalidVRMLSyntaxException;

   public void addRoute(BaseNode fromNode, String fromEventOut,
                        BaseNode toNode, String toEventIn);

   public void deleteRoute(BaseNode fromNode, String fromEventOut,
                           BaseNode toNode, String toEventIn);

   public void loadURL(String[] url, String[] parameter)
     throws InvalidVRMLSyntaxException;

   public void setDescription(String description);
}

public abstract class Field implements Cloneable
{
   public Object clone();
}
```

```
public abstract class MField extends Field
{
   public abstract int getSize();
   public abstract void clear();
   public abstract void delete(int index);
}

public abstract class ConstField extends Field
{
}

public abstract class ConstMField extends ConstField
{
   public abstract int getSize();
}

//
// This is the general BaseNode class
//
public abstract class BaseNode
{
   // Returns the type of the node.  If the node is a prototype
   // it returns the name of the prototype.
   public String getType();

   // Get the Browser that this node is contained in.
   public Browser getBrowser();
}
```

B.9.2.2 vrml.field package

```
package vrml.field;

public class SFBool extends Field
{
   public SFBool();
   public SFBool(boolean value);

   public boolean getValue();

   public void setValue(boolean b);
   public void setValue(ConstSFBool b);
   public void setValue(SFBool b);

   public String toString();   // This overrides a method in Object
}

public class SFColor extends Field
{
   public SFColor();
   public SFColor(float red, float green, float blue);

   public void getValue(float colors[]);
   public float getRed();
   public float getGreen();
   public float getBlue();

   public void setValue(float colors[]);
   public void setValue(float red, float green, float blue);
   public void setValue(ConstSFColor color);
   public void setValue(SFColor color);

   public String toString();   // This overrides a method in Object
}

public class SFFloat extends Field
{
   public SFFloat();
   public SFFloat(float f);

   public float getValue();

   public void setValue(float f);
   public void setValue(ConstSFFloat f);
   public void setValue(SFFloat f);

   public String toString();   // This overrides a method in Object
}

public class SFImage extends Field
{
   public SFImage();
   public SFImage(int width, int height, int components, byte pixels[]);

   public int getWidth();
   public int getHeight();
   public int getComponents();
   public void getPixels(byte pixels[]);

   public void setValue(int width, int height, int components,
                        byte pixels[]);
   public void setValue(ConstSFImage image);
   public void setValue(SFImage image);

   public String toString();   // This overrides a method in Object
}

public class SFInt32 extends Field
{
   public SFInt32();
   public SFInt32(int value);

   public int getValue();

   public void setValue(int i);
   public void setValue(ConstSFInt32 i);
   public void setValue(SFInt32 i);

   public String toString();   // This overrides a method in Object
}
```

```
public class SFNode extends Field
{
   public SFNode();
   public SFNode(BaseNode node);

   public BaseNode getValue();

   public void setValue(BaseNode node);
   public void setValue(ConstSFNode node);
   public void setValue(SFNode node);

   public String toString();   // This overrides a method in Object
}

public class SFRotation extends Field
{
   public SFRotation();
   public SFRotation(float axisX, float axisY, float axisZ, float angle);

   public void getValue(float rotations[]);

   public void setValue(float rotations[]);
   public void setValue(float axisX, float axisY, float axisZ,
                        float angle);
   public void setValue(ConstSFRotation rotation);
   public void setValue(SFRotation rotation);

   public String toString();   // This overrides a method in Object
}

public class SFString extends Field
{
   public SFString();
   public SFString(String s);

   public String getValue();

   public void setValue(String s);
   public void setValue(ConstSFString s);
   public void setValue(SFString s);

   public String toString();   // This overrides a method in Object
}

public class SFTime extends Field
{
   public SFTime();
   public SFTime(double time);

   public double getValue();

   public void setValue(double time);
   public void setValue(ConstSFTime time);
   public void setValue(SFTime time);

   public String toString();   // This overrides a method in Object
}
```

```
public class SFVec2f extends Field
{
   public SFVec2f();
   public SFVec2f(float x, float y);

   public void getValue(float vec2s[]);
   public float getX();
   public float getY();

   public void setValue(float vec2s[]);
   public void setValue(float x, float y);
   public void setValue(ConstSFVec2f vec);
   public void setValue(SFVec2f vec);

   public String toString();   // This overrides a method in Object
}

public class SFVec3f extends Field
{
   public SFVec3f();
   public SFVec3f(float x, float y, float z);

   public void getValue(float vec3s[]);
   public float getX();
   public float getY();
   public float getZ();

   public void setValue(float vec3s[]);
   public void setValue(float x, float y, float z);
   public void setValue(ConstSFVec3f vec);
   public void setValue(SFVec3f vec);

   public String toString();   // This overrides a method in Object
}

public class MFColor extends MField
{
   public MFColor();
   public MFColor(float colors[][]);
   public MFColor(float colors[]);
   public MFColor(int size, float colors[]);

   public void getValue(float colors[][]);
   public void getValue(float colors[]);

   public void get1Value(int index, float colors[]);
   public void get1Value(int index, SFColor color);

   public void setValue(float colors[][]);
   public void setValue(float colors[]);
   public void setValue(int size, float colors[]);
   /****************************************************
    color[0] ... color[size - 1] are used as color data
    in the way that color[0], color[1], and color[2]
    represent the first color. The number of colors
    is defined as "size / 3".
    ***************************************************/
   public void setValue(MFColor colors);
   public void setValue(ConstMFColor colors);

   public void set1Value(int index, ConstSFColor color);
   public void set1Value(int index, SFColor color);
   public void set1Value(int index, float red, float green, float blue);

   public void addValue(ConstSFColor color);
   public void addValue(SFColor color);
   public void addValue(float red, float green, float blue);

   public void insertValue(int index, ConstSFColor color);
   public void insertValue(int index, SFColor color);
   public void insertValue(int index, float red, float green, float blue);

   public String toString();   // This overrides a method in Object
}


public class MFFloat extends MField
{
   public MFFloat();
   public MFFloat(int size, float values[]);
   public MFFloat(float values[]);

   public void getValue(float values[]);

   public float get1Value(int index);

   public void setValue(float values[]);
   public void setValue(int size, float values[]);
   public void setValue(MFFloat value);
   public void setValue(ConstMFFloat value);

   public void set1Value(int index, float f);
   public void set1Value(int index, ConstSFFloat f);
   public void set1Value(int index, SFFloat f);

   public void addValue(float f);
   public void addValue(ConstSFFloat f);
   public void addValue(SFFloat f);

   public void insertValue(int index, float f);
   public void insertValue(int index, ConstSFFloat f);
   public void insertValue(int index, SFFloat f);

   public String toString();   // This overrides a method in Object
}

public class MFInt32 extends MField
{
   public MFInt32();
   public MFInt32(int size, int values[]);
   public MFInt32(int values[]);

   public void getValue(int values[]);
   public int get1Value(int index);

   public void setValue(int values[]);
   public void setValue(int size, int values[]);
   public void setValue(MFInt32 value);
   public void setValue(ConstMFInt32 value);

   public void set1Value(int index, int i);
   public void set1Value(int index, ConstSFInt32 i);
   public void set1Value(int index, SFInt32 i);

   public void addValue(int i);
   public void addValue(ConstSFInt32 i);
   public void addValue(SFInt32 i);

   public void insertValue(int index, int i);
   public void insertValue(int index, ConstSFInt32 i);
   public void insertValue(int index, SFInt32 i);

   public String toString();   // This overrides a method in Object
}

public class MFNode extends MField
{
   public MFNode();
   public MFNode(int size, BaseNode node[]);
   public MFNode(BaseNode node[]);

   public void getValue(BaseNode node[]);

   public BaseNode get1Value(int index);

   public void setValue(BaseNode node[]);
   public void setValue(int size, BaseNode node[]);
   public void setValue(MFNode node);
   public void setValue(ConstMFNode node);

   public void set1Value(int index, BaseNode node);
   public void set1Value(int index, ConstSFNode node);
   public void set1Value(int index, SFNode node);

   public void addValue(BaseNode node);
   public void addValue(ConstSFNode node);
   public void addValue(SFNode node);

   public void insertValue(int index, BaseNode node);
   public void insertValue(int index, ConstSFNode node);
   public void insertValue(int index, SFNode node);

   public String toString();   // This overrides a method in Object
}

public class MFRotation extends MField
{
   public MFRotation();
   public MFRotation(float rotations[][]);
   public MFRotation(float rotations[]);
   public MFRotation(int size, float rotations[]);

   public void getValue(float rotations[][]);
   public void getValue(float rotations[]);

   public void get1Value(int index, float rotations[]);
   public void get1Value(int index, SFRotation rotation);

   public void setValue(float rotations[][]);
   public void setValue(float rotations[]);
   public void setValue(int size, float rotations[]);
   public void setValue(MFRotation rotations);
   public void setValue(ConstMFRotation rotations);

   public void set1Value(int index, ConstSFRotation rotation);
   public void set1Value(int index, SFRotation rotation);
   public void set1Value(int index, float axisX, float axisY, float axisZ,
float angle);

   public void addValue(ConstSFRotation rotation);
   public void addValue(SFRotation rotation);
   public void addValue(float axisX, float axisY, float axisZ, float angle);

   public void insertValue(int index, ConstSFRotation rotation);
   public void insertValue(int index, SFRotation rotation);
   public void insertValue(int index, float axisX, float axisY, float axisZ,
                           float angle);

   public String toString();   // This overrides a method in Object
}

public class MFString extends MField
{
   public MFString();
   public MFString(int size, String s[]);
   public MFString(String s[]);

   public void getValue(String s[]);

   public String get1Value(int index);

   public void setValue(String s[]);
   public void setValue(int size, String s[]);
   public void setValue(MFString s);
   public void setValue(ConstMFString s);

   public void set1Value(int index, String s);
   public void set1Value(int index, ConstSFString s);
   public void set1Value(int index, SFString s);

   public void addValue(String s);
   public void addValue(ConstSFString s);
   public void addValue(SFString s);

   public void insertValue(int index, String s);
   public void insertValue(int index, ConstSFString s);
   public void insertValue(int index, SFString s);

   public String toString();   // This overrides a method in Object
}

public class MFTime extends MField
{
   public MFTime();
   public MFTime(int size, double times[]);
   public MFTime(double times[]);

   public void getValue(double times[]);

   public double get1Value(int index);

   public void setValue(double times[]);
   public void setValue(int size, double times[]);
   public void setValue(MFTime times);
   public void setValue(ConstMFTime times);

   public void set1Value(int index, double time);
   public void set1Value(int index, ConstSFTime time);
   public void set1Value(int index, SFTime time);

   public void addValue(double time);
   public void addValue(ConstSFTime time);
   public void addValue(SFTime time);

   public void insertValue(int index, double time);
   public void insertValue(int index, ConstSFTime time);
   public void insertValue(int index, SFTime time);

   public String toString();   // This overrides a method in Object
}

public class MFVec2f extends MField
{
   public MFVec2f();
   public MFVec2f(float vec2s[][]);
   public MFVec2f(float vec2s[]);
   public MFVec2f(int size, float vec2s[]);

   public void getValue(float vec2s[][]);
   public void getValue(float vec2s[]);

   public void get1Value(int index, float vec2s[]);
   public void get1Value(int index, SFVec2f vec);

   public void setValue(float vec2s[][]);
   public void setValue(float vec2s[]);
   public void setValue(int size, float vec2s[]);
   public void setValue(MFVec2f vecs);
   public void setValue(ConstMFVec2f vecs);

   public void set1Value(int index, float x, float y);
   public void set1Value(int index, ConstSFVec2f vec);
   public void set1Value(int index, SFVec2f vec);
   public void addValue(float x, float y);
   public void addValue(ConstSFVec2f vec);
   public void addValue(SFVec2f vec);

   public void insertValue(int index, float x, float y);
   public void insertValue(int index, ConstSFVec2f vec);
   public void insertValue(int index, SFVec2f vec);

   public String toString();   // This overrides a method in Object
}

public class MFVec3f extends MField
{
   public MFVec3f();
   public MFVec3f(float vec3s[][]);
   public MFVec3f(float vec3s[]);
   public MFVec3f(int size, float vec3s[]);

   public void getValue(float vec3s[][]);
   public void getValue(float vec3s[]);

   public void get1Value(int index, float vec3s[]);
   public void get1Value(int index, SFVec3f vec);

   public void setValue(float vec3s[][]);
   public void setValue(float vec3s[]);
   public void setValue(int size, float vec3s[]);
   public void setValue(MFVec3f vecs);
   public void setValue(ConstMFVec3f vecs);

   public void set1Value(int index, float x, float y, float z);
   public void set1Value(int index, ConstSFVec3f vec);
   public void set1Value(int index, SFVec3f vec);

   public void addValue(float x, float y, float z);
   public void addValue(ConstSFVec3f vec);
   public void addValue(SFVec3f vec);

   public void insertValue(int index, float x, float y, float z);
   public void insertValue(int index, ConstSFVec3f vec);
   public void insertValue(int index, SFVec3f vec);

   public String toString();   // This overrides a method in Object
}

public class ConstSFBool extends ConstField
{
   public ConstSFBool(boolean value);

   public boolean getValue();

   public String toString();   // This overrides a method in Object
}
```

```
public class ConstSFColor extends ConstField
{
   public ConstSFColor(float red, float green, float blue);

   public void getValue(float colors[]);
   public float getRed();
   public float getGreen();
   public float getBlue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFFloat extends ConstField
{
   public ConstSFFloat(float value);

   public float getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFImage extends ConstField
{
   public ConstSFImage(int width, int height, int components, byte pixels[]);

   public int getWidth();
   public int getHeight();
   public int getComponents();
   public void getPixels(byte pixels[]);

   public String toString();   // This overrides a method in Object
}

public class ConstSFInt32 extends ConstField
{
   public ConstSFInt32(int value);

   public int getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFNode extends ConstField
{
   public ConstSFNode(BaseNode node);

   public BaseNode getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFRotation extends ConstField
{
   public ConstSFRotation(float axisX, float axisY, float axisZ, float angle);

   public void getValue(float rotations[]);
   public String toString();   // This overrides a method in Object
}

public class ConstSFString extends ConstField
{
   public ConstSFString(String value);

   public String getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFTime extends ConstField
{
   public ConstSFTime(double time);

   public double getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFVec2f extends ConstField
{
   public ConstSFVec2f(float x, float y);

   public void getValue(float vec2s[]);
   public float getX();
   public float getY();

   public String toString();   // This overrides a method in Object
}

public class ConstSFVec3f extends ConstField
{
   public ConstSFVec3f(float x, float y, float z);

   public void getValue(float vec3s[]);
   public float getX();
   public float getY();
   public float getZ();

   public String toString();   // This overrides a method in Object
}

public class ConstMFColor extends ConstMField
{
   public ConstMFColor(float colors[][]);
   public ConstMFColor(float colors[]);
   public ConstMFColor(int size, float colors[]);

   public void getValue(float colors[][]);
   public void getValue(float colors[]);

   public void get1Value(int index, float colors[]);
   public void get1Value(int index, SFColor color);

   public String toString();   // This overrides a method in Object
```

```
}

public class ConstMFFloat extends ConstMField
{
   public ConstMFFloat(int size, float values[]);
   public ConstMFFloat(float values[]);

   public void getValue(float values[]);

   public float get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFInt32 extends ConstMField
{
   public ConstMFInt32(int size, int values[]);
   public ConstMFInt32(int values[]);

   public void getValue(int values[]);

   public int get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFNode extends ConstMField
{
   public ConstMFNode(int size, BaseNode node[]);
   public ConstMFNode(BaseNode node[]);

   public void getValue(BaseNode node[]);

   public BaseNode get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFRotation extends ConstMField
{
   public ConstMFRotation(float rotations[][]);
   public ConstMFRotation(float rotations[]);
   public ConstMFRotation(int size, float rotations[]);

   public void getValue(float rotations[][]);
   public void getValue(float rotations[]);

   public void get1Value(int index, float rotations[]);
   public void get1Value(int index, SFRotation rotation);

   public String toString();   // This overrides a method in Object
}

public class ConstMFString extends ConstMField
{
   public ConstMFString(int size, String s[]);
   public ConstMFString(String s[]);
   public void getValue(String values[]);

   public String get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFTime extends ConstMField
{
   public ConstMFTime(int size, double times[]);
   public ConstMFTime(double times[]);

   public void getValue(double times[]);

   public double get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFVec2f extends ConstMField
{
   public ConstMFVec2f(float vec2s[][]);
   public ConstMFVec2f(float vec2s[]);
   public ConstMFVec2f(int size, float vec2s[]);

   public void getValue(float vec2s[][]);
   public void getValue(float vec2s[]);

   public void get1Value(int index, float vec2s[]);
   public void get1Value(int index, SFVec2f vec);

   public String toString();   // This overrides a method in Object
}

public class ConstMFVec3f extends ConstMField
{
   public ConstMFVec3f(float vec3s[][]);
   public ConstMFVec3f(float vec3s[]);
   public ConstMFVec3f(int size, float vec3s[]);

   public void getValue(float vec3s[][]);
   public void getValue(float vec3s[]);

   public void get1Value(int index, float vec3s[]);
   public void get1Value(int index, SFVec3f vec);

   public String toString();   // This overrides a method in Object
}
```

B.9.2.3 vrml.node package

```
package vrml.node;

//
// This is the general Node class
//
public abstract class Node extends BaseNode
{
   // Get an EventIn by name. Return value is write-only.
   //   Throws an InvalidEventInException if eventInName isn't a valid
   //   eventIn name for a node of this type.
   public final Field getEventIn(String eventInName);

   // Get an EventOut by name. Return value is read-only.
   //   Throws an InvalidEventOutException if eventOutName isn't a valid
   //   eventOut name for a node of this type.
   public final ConstField getEventOut(String eventOutName);

   // Get an exposed field by name.
   //    Throws  an  InvalidExposedFieldException  if  exposedFieldName  isn't  a
valid
   //   exposedField name for a node of this type.
   public final Field getExposedField(String exposedFieldName);

   public String toString();   // This overrides a method in Object
}

//
// This is the general Script class, to be subclassed by all scripts.
// Note that the provided methods allow the script author to explicitly
// throw tailored exceptions in case something goes wrong in the
// script.
//
public abstract class Script extends BaseNode
{
   // This method is called before any event is generated
   public void initialize();

   // Get a Field by name.
   //   Throws an InvalidFieldException if fieldName isn't a valid
   //   field name for a node of this type.
   protected final Field getField(String fieldName);

   // Get an EventOut by name.
   //   Throws an InvalidEventOutException if eventOutName isn't a valid
   //   eventOut name for a node of this type.
   protected final Field getEventOut(String eventOutName);

   // Get an EventIn by name.
   //   Throws an InvalidEventInException if eventInName isn't a valid
   //   eventIn name for a node of this type.
   protected final Field getEventIn(String eventInName);

   // processEvents() is called automatically when the script receives
   //   some set of events. It shall not be called directly except by its
   //   subclass.
   //   count indicates the number of events delivered.
   public void processEvents(int count, Event events[]);

   // processEvent() is called automatically when the script receives
   // an event.
   public void processEvent(Event event);

   // eventsProcessed() is called after every invocation of processEvents().
   public void eventsProcessed()

   // shutdown() is called when this Script node is deleted.
   public void shutdown();

   public String toString();   // This overrides a method in Object
}
```

B.10 Example of exception class

public class **InvalidEventInException** extends IllegalArgumentException

```
{
   /**
    * Constructs an InvalidEventInException with no detail message.
    */
   public InvalidEventInException(){
      super();
   }
   /**
    * Constructs an InvalidEventInException with the specified detail
    * message.
    * A detail message is a String that describes this particular exception.
    * @param s the detail message
    */
   public InvalidEventInException(String s){
      super(s);
   }
}

public class InvalidEventOutException extends IllegalArgumentException
{
   public InvalidEventOutException(){
      super();
   }
   public InvalidEventOutException(String s){
      super(s);
   }
}
```

```
public class InvalidExposedFieldException extends IllegalArgumentException
{
   public InvalidExposedFieldException(){
      super();
   }
   public InvalidExposedFieldException(String s){
      super(s);
   }
}

public class InvalidFieldChangeException extends IllegalArgumentException
{
   public InvalidFieldChangeException(){
      super();
   }
   public InvalidFieldChangeException(String s){
      super(s);
   }
}

public class InvalidFieldException extends IllegalArgumentException
{
   public InvalidFieldException(){
      super();
   }
   public InvalidFieldException(String s){
      super(s);
   }
}

public class InvalidRouteException extends IllegalArgumentException
{
   public InvalidRouteException(){
      super();
   }
   public InvalidRouteException(String s){
      super(s);
   }
}

public class InvalidVRMLSyntaxException extends Exception
{
   public InvalidVRMLSyntaxException(){
      super();
   }
   public InvalidVRMLSyntaxException(String s){
      super(s);
   }

   public String getMessage();  // This overrides a method in Exception
}
```

Annex C
(normative)

ECMAScript scripting reference

C.1 Introduction and table of contents

This annex describes the ECMAScript programming language that enables Script nodes (see 6.40, Script) to interact with VRML scenes. See 4.12, Scripting, for a general description of scripting languages in ISO/IEC 14772. Note that support for the ECMAScript is not required by ISO/IEC 14772, but any access of ECMAScript from within VRML Script nodes shall conform with the requirements specified in this annex.

C.2 Language

ECMAScript is a general purpose, cross-platform programming language that can be used with ISO/IEC 14772 to provide scripting of events, objects, and actions. ECMAScript is fully described in 2.[ESCR]. Prior to standardization as ECMA-262, ECMAScript was known as Netscape JavaScript. Several syntactic entities in this annex reflect this origin.

C.3 Supported protocol in the Script node's *url* field

C.3.1 *url* field

The *url* field of the Script node may contain URL references to ECMAScript code as illustrated below:

```
Script { url "http://foo.com/myScript.js" }
```

The javascript: protocol allows the script to be placed inline as follows:

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

Browsers supporting the ECMAScript scripting language shall support the javascript: protocol as well as the the other required protocols (see 7, Conformance and minimum support requirements).

The *url* field may contain multiple URL's referencing either a remote file or in-line code as shown in the following example:

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

C.3.2 File extension

The file extension for ECMAScript source code is '.js', unless a protocol returning mime types is used (such as HTTP). In that case, any suffix is allowed as long as the proper mime type is returned (see C.3.3, Mime type).

C.3.3 MIME type

The MIME type for ECMAScript source code is defined as follows:

```
application/x-javascript
```

C.4 eventIn handling

C.4.1 Receiving eventIns

Events sent to the Script node are passed to the corresponding ECMAScript function in the script. The script is specified in the *url* field of the Script node. The function's name is the same as the eventIn and is passed two arguments, the event value and its timestamp (see C.4.2, Parameter passing and the eventIn function). If there is no corresponding ECMAScript function in the script, the browser's behaviour is undefined.

For example, the following Script node has one eventIn field whose name is *start*:

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

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

C.4.2 Parameter passing and the eventIn function

When a Script node receives an eventIn, a corresponding function in the file specified in the *url* field of the Script node is called. This function has two arguments. The value of the eventIn is passed as the first argument and the timestamp of the eventIn is passed as the second argument. The type of the value is the same as the type of the eventIn and the type of the timestamp is SFTime. C.6.1, VRML field to ECMAScript variable conversion, provides a description of how VRML types appear in ECMAScript. The values of the parameters have no visibility outside the function.

C.4.3 eventsProcessed( ) function

Authors may define an eventsProcessed() function that is called after some set of events has been received. This allows Script nodes that do not rely on the ordering of events received to generate fewer events than an equivalent Script node that generates events whenever events are received (see C.4.1, Receiving eventIns).

The eventsProcessed( ) function takes no parameters. Events generated from it are given the timestamp of the last event processed.

C.4.4 initialize( ) function

Authors may define a function named initialize( ) which is invoked before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script (see 4.12.3, Initialize() and shutdown()).

The initialize( ) function has no parameters. Events generated from initialize( ) are given the timestamp of when the Script node was loaded.

C.4.5 shutdown( ) function

Authors may define a function named shutdown( ) which is invoked when the corresponding Script node is deleted or when the world containing the Script node is unloaded or replaced by another world (see 4.12.3, Initialize() and shutdown()).

The shutdown( ) function has no parameters. Events generated from shutdown( ) are given the timestamp of when the Script node was deleted.

C.5 Accessing fields and events

C.5.1 Accessing fields and eventOuts of the Script

The fields and eventOuts of a Script node are accessible from its ECMAScript functions. As in all other nodes, the fields are accessible only within the Script. The eventIns are not accessible. The Script node's eventIns can be routed to and its eventOuts can be routed from. Another Script node with a reference to this node can access its eventIns and eventOuts as for any other node.

A field defined in a Script node is available to the script by using its name. Its value can be read or written. This value is persistent across function calls. EventOuts defined in the script node can also be read. The value is the last value assigned.

C.5.2 Accessing fields and eventOuts of other nodes

The script can access any exposedField, eventIn or eventOut of any node to which it has access:

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

This example 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. However, 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 ISO/IEC 14772.

Events generated by setting an eventIn on a node are sent at the completion of the currently executing function. The eventIn shall be assigned a value of the same datatype; no partial assignments are allowed. For example, it is not possible to assign the red value of an SFColor eventIn. Since eventIns are strictly write-only, the remainder of the partial assignment would have invalid field values. Assigning to the eventIn field multiple times during one execution of the function still only sends one event and that event is the last value assigned.

C.5.3 Sending eventOuts

Assigning to an eventOut of a Script node, or a component of an eventOut (i.e. MF eventOut or a property of an SF eventOut), sends an event to that eventOut. Events are sent at the end of script execution. An eventOut may be assigned a value multiple times within the script, but the value sent shall be the last value assigned to the eventOut. If the value of individual components of an eventOut are changed, the last value given to each component shall be sent. Components that are not changed in the script, send their initial value determined at the beginning of the script execution. For example, the following script segment produces an eventOut value of (4, 3, 1) for the eventOut SFVec3f *foo_changed* with an initial value of (6, 6, 6):

```
a = foo_changed; // copy by reference a(6,6,6)
a.x = 5;         // foo_changed(5,6,6)
a.z = 1;         // foo_changed(5,6,1)
b = foo_changed; // copy by reference b(5,6,1)
b.x = 4;         // foo_changed(4,6,1)
c = a;           // copy by reference c(4,6,1)
c.y = 3;         // foo_changed(4,3,1))
```

C.6 ECMAScript objects

C.6.1 Notational conventions

Since ECMAScript is an untyped language it has no language constructs to describe the types of parameters passed to, or values returned from, functions. Therefore this annex uses a notational convention to describe these types. Parameters passed are preceded by their type, and the type of any return value precedes the function name. Normally these types correspond to VRML field types, so those names are used. In the case of no return value, the identifier *void* is used. In the case of a ECMAScript numeric value or numeric array return, the identifier *numeric* or *numeric[ ]* is used. In the case of a string return, the identifier *String* is used.

C.6.2 VRML field to ECMAScript variable conversion

ECMAScript native datatypes consist of boolean, numeric and string. The language is not typed, so datatypes are implicit upon assignment. The VRML SFBool is mapped to the ECMAScript boolean. In addition to the ECMAScript *true* and *false* constants, the VRML TRUE and FALSE values may be used. The VRML SFInt32, SFFloat and SFTime fields are mapped to the numeric datatype and will be maintained in double precision accuracy. These types are passed by value in function calls. All other VRML fields are mapped to ECMAScript objects. ECMAScript objects are passed by reference.

The ECMAScript boolean, numeric and string are automatically converted to other datatypes when needed. See 2.[ESCR] for more details.

In ECMAScript, assigning a new value to a variable gives the variable the datatype of the new value, in addition to the value. Scalar values (boolean and numeric) are assigned by copying the value. Other objects are assigned by reference.

When assignments are made to eventOuts and fields, the values are converted to the VRML field type. Values assigned are always copied. This contrasts with normal assignment in ECMAScript where all assignments except for scalar are performed by reference.

For eventOut objects, assignment copies the value to the eventOut, which will be sent upon completion of the current function. Assigning an eventOut to an internal variable copies by reference. Subsequent assignments to that internal variable will behave like assignments to the eventOut (i.e., an event will be sent at the end of the function). Field objects behave identically to eventOut objects, except that no event is sent upon completion of the function.

Assigning an element of an MF object to an internal variable creates a reference to that element. The type shall be the corresponding SF object type. If the MF object is an eventOut and an assignment is made to the internal variable, an event will be sent at the end of the function. Assigning an SF object to an element of an MF object which is a field or eventOut (which shall be of the corresponding type) copies the value of the SF object into the MF object element. If the MF object is an eventOut an event will be sent at the end of the function.

C.6.3 Browser object

This subclause lists the class static functions available in the *Browser* object which allow scripts to get and set browser information. Descriptions of the functions are provided in 4.12.10, Browser script interface. The syntax for a call is:

```
    mymfnode = Browser.createVrmlFromString('Sphere {}');
```

Table C.1 describes the Browser object's functions, parameters, and return values.

Table C.1 -- Browser object functions

Return value	Function
String	**getName**( )
String	**getVersion**( )
numeric	**getCurrentSpeed**( )
numeric	**getCurrentFrameRate**( )
String	**getWorldURL**( )
void	**replaceWorld**(MFNode nodes)
MFNode	**createVrmlFromString**(String vrmlSyntax)
void	**createVrmlFromURL**(MFString url, Node node, String event)
void	**addRoute**(SFNode fromNode, String fromEventOut, SFNode toNode, String toEventIn)
void	**deleteRoute**(SFNode fromNode, String fromEventOut, SFNode toNode, String toEventIn)
void	**loadURL**(MFString url, MFString parameter)
void	**setDescription**(String description)

C.6.4 SFColor object

C.6.4.1 Description

The SFColor object corresponds to a VRML SFColor field. All properties are accessed using the syntax *sfColorObjectName.<property>*, where *sfColorObjectName* is an instance of an SFColor object. The properties may also be accessed by the indices [0] for red, [1] for green and [2] blue. All functions are invoked using the syntax *sfColorObjectName.method(<argument-list>)*, where *sfColorObjectName* is an instance of an SFColor object.

C.6.4.2 Instance creation function

sfColorObjectName = new SFColor(float *r*, float *g*, float *b)*

where

r, *g*, and *b* are the red, green, and blue values of the colour. Missing values will be filled by 0.0.

C.6.4.3 Properties

The properties of the SFColor object are described in Table C.2.

Table C.2 -- SFColor properties

Property	**Description**
numeric *r*	red component of the colour
numeric *g*	green component of the colour
numeric *b*	blue component of the colour

C.6.4.4 Functions

The functions of the SFColor object are described in Table C.3.

Table C.3 -- SFColor functions

Function	**Description**
void setHSV(float *h*, float *s*, float *v*)	Sets the value of the colour by specifying the values of *hue*, *saturation*, and *value*.
numeric[3] getHSV( )	Returns the value of the colour in a 3 element numeric array, with *hue* at index 0, *saturation* at index 1, and *value* at index 2.
String toString( )	Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of *r*, *g* and *b*.

C.6.5 SFImage object

C.6.5.1 Description

The SFImage object corresponds to a VRML SFImage field.

C.6.5.2 Instance creation function

sfImageObjectName = new SFImage(numeric *x*, numeric *y*, numeric *comp*, MFInt32 *array)*

where

x is the x-dimension of the image. *y* is the y-dimension of the image. *comp* is the number of components of the image (1 for greyscale, 2 for greyscale+alpha, 3 for rgb, 4 for rgb+alpha). *Array* contains the *x* × *y* values for the pixels of the image. The format of each pixel is an SFImage as in the PixelTexture node.

C.6.5.3 Properties

The properties of the SFImage object are listed in Table C.4.

Table C.4 -- SFImage properties

Property	Description
numeric *x*	x dimension of the image
numeric *y*	y dimension of the image
numeric *comp*	*number of components of the image:* *1: greyscale* *2: greyscale + alpha* *3: rgb* *4: rgb + alpha*
MFInt32 *array*	image data

C.6.5.4 Functions

The function of the SFImage object is described in Table C.5.

Table C.5 -- SFImage function

Function	Description
String toString( )	Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x, y, comp and array.

C.6.6 SFNode object

C.6.6.1 Description

The SFNode object corresponds to a VRML SFNode field.

C.6.6.2 Instance creation function

sfNodeObjectName = new SFNode(String *vrmlstring)*

where

vrmlstring is an ISO 10646 string containing a legal VRML string as described in 4.12.10.9, MFNode createVrmlFromString(SFString vrmlSyntax). If the string produces other than one top-level node, the results are undefined. The string may contain any number of ROUTE's, PROTO's, and EXTERNPROTO's in accordance with 4.12.10.9, MFNode createVrmlFromString(SFString vrmlSyntax).

C.6.6.3 Properties

Each node may assign values to its eventIns and obtain the last output values of its eventOuts using the *sfNodeObjectName.eventName* syntax.

C.6.6.4 functions

The function of the SFNode object is described in Table C.6.

Table C.6 -- SFNode function

Function	Description
String toString( )	Returns the VRML UTF-8 string that, if parsed as the value of an SFNode field, would produce this node. If the browser is unable to reproduce this node, the name of the node followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings.

C.6.7 SFRotation object

C.6.7.1 Description

The SFRotation object corresponds to a VRML SFRotation field. It has four numeric properties: x, y, z (the axis of rotation) and angle. These may also be addressed by indices [0] through [3].

C.6.7.2 Instance creation functions

sfRotationObjectName = new SFRotation(numeric *x*, numeric *y*, numeric *z*, numeric *angle)*

where

x, *y*, and *z* are the axis of the rotation. *angle* is the angle of the rotation (in radians). Missing values default to 0.0, except *y*, which defaults to 1.0.

sfRotationObjectName = new SFRotation(SFVec3f *axis,* numeric *angle*)

where

axis is the axis of rotation. *angle* is the angle of the rotation (in radians)

sfRotationObjectName = new SFRotation(SFVec3f *fromVector,* SFVec3f *toVector*) where

fromVector and *toVector* are normalized and the rotation value that would rotate from the *fromVector* to the *toVector* is stored in the object.

C.6.7.3 Properties

The properties of the SFRotation object are described in Table C.7.

Table C.7 -- SFRotation properties

Property	Description
numeric *x*	first value of the axis vector
numeric *y*	second value of the axis vector
numeric *z*	third value of the axis vector
numeric *angle*	the angle of the rotation (in radians)

C.6.7.4 Functions

The functions of the SFRotation object are described in Table C.8.

Table C.8 -- SFRotation functions

Function	Description
SFVec3f getAxis( )	Returns the axis of rotation.
SFRotation inverse( )	Returns the inverse of this object's rotation.
SFRotation multiply(SFRotation *rot*)	Returns the object multiplied by the passed value.
SFVec3f multVec(SFVec3f *vec*)	Returns the value of *vec* multiplied by the matrix corresponding to this object's rotation.
void setAxis(SFVec3f *vec*)	Sets the axis of rotation to the value passed in *vec*.
SFRotation slerp(SFRotation *dest*, numeric *t*)	Returns the value of the spherical linear interpolation between this object's rotation and *dest* at value 0 <= *t* <= 1. For *t* = 0, the value is this object's rotation. For *t* = 1, the value is *dest*.
String toString( )	Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x, y, z, and angle.

C.6.8 SFVec2f object

C.6.8.1 Description

The SFVec2f object corresponds to a VRML SFVec2f field. Each component of the vector can be accessed using the *x* and *y* properties or using C-style array dereferencing (i.e., *sfVec2fObjectName[0]* or *sfVec2fObjectName[1]).*

C.6.8.2 Instance creation function

sfVec2fObjectName = new SFVec2f(numeric *x*, numeric *y)*

Missing values default to 0.0.

C.6.8.3 Properties

The properties of the SFVec2f object are described in Table C.9.

Table C.9 -- SFVec2f properties

Property	Description
numeric *x*	First value of the vector.
numeric *y*	Second value of the vector.

C.6.8.4 Functions

The functions of the SFVec2f object are described in Table C.10.

Table C.10 -- SFVec2f functions

Function	Description
SFVec2f add(SFVec2f *vec*)	Returns the value of the passed value added, component-wise, to the object.
SFVec2f divide(numeric *n*)	Returns the value of the object divided by the passed value.
numeric dot(SFVec2f *vec*)	Returns the dot product of this vector and the passed value.
numeric length( )	Returns the geometric length of this vector.
SFVec2f multiply(numeric *n*)	Returns the value of the object multiplied by the passed value.
SFVec2f normalize( )	Returns the object converted to unit length .
SFVec2f subtract(SFVec2f *vec*)	*Returns the value of the passed value subtracted, component-wise, from the object.*
String toString( )	Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x and y.

C.6.9 SFVec3f object

C.6.9.1 Description

The SFVec3f object corresponds to a VRML SFVec3f field. Each component of the vector can be accessed using the x, y, and z properties or using C-style array dereferencing (i.e., *sfVec3fObjectName[0]*, *sfVec3fObjectName[1]* or *sfVec3fObjectName[2]).*

C.6.9.2 Instance creation function

sfVec3fObjectName = new SFVec3f(numeric *x,* numeric *y,* numeric *z)*

Missing values default to 0.0.

C.6.9.3 Properties

The properties of the SFVec3f object are described in Table C.11.

Table C.11 -- SFVec2f properties

Property	Description
numeric *x*	First value of the vector.
numeric *y*	Second value of the vector.
numeric *z*	Third value of the vector.

C.6.9.4 Functions

The functions of the SFVec3f object are described in Table C.12.

Table C.12 -- SFVec3f functions

Function	Description
SFVec3f add(SFVec3f *vec*)	Returns the value of the passed value added, component-wise, to the object.
SFVec3f cross(SFVec3f *vec*)	Returns the cross product of the object and the passed value.
SFVec3f divide(numeric *n*)	Returns the value of the object divided by the passed value.
numeric dot(SFVec3f *vec*)	Returns the dot product of this vector and the passed value.
numeric length( )	Returns the geometric length of this vector.

SFVec3f multiply(*numeric n*)	Returns the value of the object multiplied by the passed value.
SFVec3f negate( )	Returns the value of the component-wise negation of the object.
SFVec3f normalize( )	Returns the object converted to unit length .
SFVec3f subtract(SFVec3f *vec*)	Returns the value of the passed value subtracted, component-wise, from the object.
String toString( )	Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x, y, and z.

C.6.10 MFColor object

C.6.10.1 Description

The MFColor object corresponds to a VRML MFColor field. It is used to store a one-dimensional array of SFColor objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfColorObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= length results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFColor (0, 0, 0).

C.6.10.2 Instance creation function

mfColorObjectName = new MFColor(SFColor *c1*, SFColor *c2, ...)*

The creation function shall initialize the array using 0 or more SFColor-valued expressions passed as parameters.

C.6.10.3 Property

The property of the MFColor object is described in Table C.13.

Table C.13 -- MFColor properties

Property	**Description**
numeric *length*	property for getting/setting the number of elements in the array.

C.6.10.4 Function

The single function of the MFColor object is described in Table C.14.

Table C.14 -- MFColor functions

Function	**Description**

String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFColor array.

C.6.11 MFFloat object

C.6.11.1 Description

The MFFloat object corresponds to a VRML MFFloat field. It is used to store a one-dimensional array of SFFloat values. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfFloatObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to 0.0.

C.6.11.2 Instance creation function

mfFloatObjectName = new MFFloat(numeric *n1*, numeric *n2, ...)*

where

The creation function shall initialize the array using 0 or more numeric-valued expressions passed as parameters.

C.6.11.3 Property

The property of the MFFloat object is described in Table C.15.

Table C.15 -- MFFloat properties

Property	Description
numeric *length*	property for getting/setting the number of elements in the array.

C.6.11.4 Function

The single function of the MFFloat object is described in Table C.16.

Table C.16 -- MFFloat function

Function	Description
String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFFloat array.

C.6.12 MFInt32 object

C.6.12.1 Description

The MFInt32 object corresponds to a VRML MFInt32 field. It is used to store a one-dimensional array of SFInt32 values. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfInt32ObjectName*[*index]*, where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to 0.

C.6.12.2 Instance creation function

mfInt32ObjectName = new MFInt32(numeric *n1*, numeric *n2, ...)*

where

The creation function shall initialize the array using 0 or more integer-valued expressions passed as parameters.

C.6.12.3 Property

The property of the MFInt32 object is described in Table C.17.

Table C.17 -- MFInt32 property

Property	Description
numeric *length*	property for getting/setting the number of elements in the array.

C.6.12.4 Function

The single function of the MFInt32 object is described in Table C.18.

Table C.18 -- MFInt32 function

Function	Description
String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFInt32 array.

C.6.13 MFNode object

C.6.13.1 Description

The MFNode object corresponds to a VRML MFNode field. It is used to store a one-dimensional array of SFNode objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfNodeObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to NULL.

C.6.13.2 Instance creation function

mfNodeObjectName = new MFNode(SFNode *n1*, SFNode *n2, ...)*

where

The creation function shall initialize the array using 0 or more SFNode-valued expressions passed as parameters.

C.6.13.3 Property

The property of the MFNode object is described in Table C.19.

Table C.19 -- MFNode property

Property	Description
numeric *length*	property for getting/setting the number of elements in the array.

C.6.13.4 Function

The single function of the MFNode object is described in Table C.20.

Table C.20 -- MFNode function

Function	Description
String toString( )	Returns the VRML UTF-8 string that, if parsed as the value of a MFNode field, would produce this array of nodes. If the browser is unable to reproduce this node, the name of the node followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings

C.6.14 MFRotation object

C.6.14.1 Description

The MFRotation object corresponds to a VRML MFRotation field. It is used to store a one-dimensional array of SFRotation objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfRotationObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFRotation (0, 0, 1, 0).

C.6.14.2 Instance creation function

mfRotationObjectName = new MFRotation(SFRotation *r1*, SFRotation *r2, ...)*

where

The creation function shall initialize the array using 0 or more SFRotation-valued expressions passed as parameters.

C.6.14.3 Property

The property of the MFRotation object is described in Table C.21.

Table C.21 -- MFRotation property

Property	Description

numeric *length*	property for getting/setting the number of elements in the array.

C.6.14.4 Function

The single function of the MFRotation object is described in Table C.22.

Table C.22 -- MFRotation function

Function	Description
String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFRotation array.

C.6.15 MFString object

C.6.15.1 Description

The MFString object corresponds to a VRML 2.0 MFString field. It is used to store a one-dimensional array of String objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfStringObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to the empty string.

C.6.15.2 Instance creation function

mfStringObjectName = new MFString(String *s1*, String *s2, ...)*

where

The creation function shall initialize the array using 0 or more String-valued expressions passed as parameters.

C.6.15.3 Property

The property of the MFString object is described in Table C.23.

Table C.23 -- MFString property

Property	Description
numeric *length*	property for getting/setting the number of elements in the array.

C.6.15.4 Function

The single function of the MFString object is described in Table C.24.

Table C.24 -- MFString function

Function	Description

String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFString array.

C.6.16 MFTime object

C.6.16.1 Description

The MFTime object corresponds to a VRML MFTime field. It is used to store a one-dimensional array of SFTime values. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfTimeObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to 0.0.

C.6.16.2 Instance creation function

mfTimeObjectName = new MFTime(numeric *n1*, numeric *n2, ...)*

The creation function shall initialize the array using 0 or more numeric-valued expressions passed as parameters.

C.6.16.3 Property

The property of the MFTime object is described in Table C.25.

Table C.25 -- MFTime property

Property	Description
numeric *length*	property for getting/setting the number of elements in the array.

C.6.16.4 Function

The function of the MFTime object is described in Table C.26.

Table C.26 -- MFTime function

Function	Description
String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFTime array.

C.6.17 MFVec2f object

C.6.17.1 Description

The MFVec2f object corresponds to a VRML MFVec2f field. It is used to store a one-dimensional array of SFVec2f objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfVec2fObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFVec2f (0, 0).

C.6.17.2 Instance creation function

mfVec2fObjectName = new MFVec2f(SFVec2f *v1*, SFVec2f *v2*, *...)*

The creation function shall initialize the array using 0 or more SFVec2f-valued expressions passed as parameters.

C.6.17.3 Property

The property of the MFVec2f object is described in Table C.27.

Table C.27 -- MFVec2f property

Property	Description
numeric *length*	property for getting/setting the number of elements in the array.

C.6.17.4 Function

The single function of the MFVec2f object is described in Table C.28.

Table C.28 -- MFVec2f function

Function	Description
String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFVec2f array.

C.6.18 MFVec3f object

C.6.18.1 Description

The MFVec3f object corresponds to a VRML MFVec3f field. It is used to store a one-dimensional array of SFVec3f objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfVec3fObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFVec3f (0, 0, 0).

C.6.18.2 Instance creation function

mfVec3fObjectName = new MFVec3f(SFVec3f *v1*, SFVec3f *v2*,*...)*

where

The creation function shall initialize the array using 0 or more SFVec3f-valued expressions passed as parameters.

C.6.18.3 Property

The property of the MFVec3f object is described in Table C.29.

Table C.29 -- MFVec3f property

Property	Description
numeric *length*	property for getting/setting the number of elements in the array.

C.6.18.4 Function

The single function of the MFVec3f object is described in Table C.30.

Table C.30 -- MFVec3f function

Function	Description
String toString( )	Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFVec3f array.

C.6.19 VrmlMatrix object

C.6.19.1 Description

The VrmlMatrix object provides many useful functions for performing manipulations on 4x4 matrices. Each of element of the matrix can be accessed using C-style array dereferencing (i.e., vrmlMatrixObjectName[0][1] is the element in row 0, column 1). The results of dereferencing a VrmlMatrix object using a single index (i.e., vrmlMatrixObjectName[0]) are undefined. The translation elements are in the fourth row. For example, vrmlMatrixObjectName[3][0] is the X offset.

C.6.19.2 Instance creation functions

VrmlMatrixObjectName = new VrmlMatrix(
numeric *f11*, numeric *f12*, numeric *f13*, numeric *f14*,
numeric *f21*, numeric *f22*, numeric *f23*, numeric *f24*,
numeric *f31*, numeric *f32*, numeric *f33*, numeric *f34*,
numeric *f41*, numeric *f42*, numeric *f43*, numeric *f44)*

A new matrix initialized with the values in *f11* through *f44* is created and returned. The translation values will be *f41, f42*, and *f43*.

VrmlMatrixObjectName = new VrmlMatrix( )

A new matrix initialized with the identity matrix is created and returned.

C.6.19.3 Properties

The VRMLMatrix object has no properties.

C.6.19.4 Functions

The functions of the VRMLMatrix object are listed in Table C.31.

Table C.31 -- VRMLMatrix functions

Function	Description
void setTransform(SFVec3f *translation*, SFRotation *rotation*, SFVec3f *scale*, SFRotation *scaleOrientation*, SFVec3f *center*)	Sets the VrmlMatrix to the passed values. Any of the rightmost parameters may be omitted. The function has 0 to 5 parameters. For example, specifying 0 parameters results in an identity matrix while specifying 1 parameter results in a translation and specifying 2 parameters results in a translation and a rotation. Any unspecified parameter is set to its default as specified for the Transform node.
void getTransform(SFVec3f *translation*, SFRotation *rotation*, SFVec3f *scale*)	Decomposes the VrmlMatrix and returns the components in the passed *translation*, *rotation*, and *scale* objects. The types of these passed objects is the same as the first three arguments to **setTransform**. If any passed object is not sent, or if the null object is sent for any value, that value is not returned. Any projection or shear information in the matrix is ignored.
VrmlMatrix inverse( )	Returns a VrmlMatrix whose value is the inverse of this object.
VrmlMatrix transpose( )	Returns a VrmlMatrix whose value is the transpose of this object.
VrmlMatrix multLeft(VrmlMatrix *matrix*)	Returns a VrmlMatrix whose value is the object multiplied by the passed *matrix* on the left.
VrmlMatrix multRight(VrmlMatrix *matrix*)	Returns a VrmlMatrix whose value is the object multiplied by the passed *matrix* on the right.
SfVec3f multVecMatrix(SFVec3f *vec*)	Returns an SFVec3f whose value is the object multiplied by the passed row vector.
SFVec3f multMatrixVec(SFVec3f *vec*)	Returns an SFVec3f whose value is the object multiplied by the passed column vector.
String toString( )	Returns a String containing the values of the VrmlMatrix.

C.7 Examples

The following is an example of a Script node which determines whether a given colour contains a lot of red. The Script node exposes a Color field, an eventIn, and an eventOut:

```
DEF Example_1 Script {
        field    SFColor currentColor 0 0 0
    eventIn  SFColor colorIn
    eventOut SFBool  isRed

    url "javascript:
        function colorIn(newColor, ts) {
            // This function is called when a colorIn event is received
            currentColor = newColor;
        }

        function eventsProcessed( ) {
            if (currentColor[0] >= 0.5)
                // if red is at or above 50%
                isRed = true;
        }"
}
```

Details on when the functions defined in Example_1 Script are called are provided in 4.12.2, Script execution.

The following example illustrate use of the createVrmlFromURL( ) function:

```
 DEF Example_2 Script {
    field   SFNode myself USE Example_2
    field   SFNode root USE ROOT_TRANSFORM
    field   MFString url "foo.wrl"
    eventIn MFNode   nodesLoaded
    eventIn SFBool   trigger_event

    url "javascript:
        function trigger_event(value, ts){
            // do something and then fetch values
            Browser.createVRMLFromURL(url, myself, 'nodesLoaded');
        }

        function nodesLoaded(value, timestamp){
            if (value.length > 5) {
                 // do something more than 5 nodes in this MFNode...
            }
            root.addChildren = value;
        }"
}
```

The following example illustrates use of the addRoute( ) function:

```
DEF Sensor TouchSensor {}

DEF Baa Script {
    field   SFNode myself USE Baa
    field   SFNode fromNode USE Sensor
    eventIn SFBool clicked
    eventIn SFBool trigger_event

    url "javascript:
        function trigger_event(eventIn_value){
            // do something and then add routing
            Browser.addRoute(fromNode, 'isActive', myself, 'clicked');
        }

        function clicked(value){
            // do something
        }"
}
```

The following example illustrates assigning with references and assigning by copying:

```
Script {
    eventIn  SFBool  eI
    eventOut SFVec3f eO
    field    MFVec3f f []

    url "javascript:
        function eI( ) {
            eO = new SFVec3f(0,1,2);  // ’eO’ contains the value
                                      // (0,1,2) which will be sent
                                      // out when the function
                                      // is complete.
            a = eO;                   // ’a’ references the eventOut
                                      // ’e0’
            b = a;                    // ’a’ and ’b’ now both reference
                                      // ’e0’
            a.x = 3;                  // ’e0’ will send (3,1,2) at the
                                      // end of the function
            f[1] = a;                 // ’f[1]’ contains the value
                                      // (3,1,2).
            c = f[1];                 // ’c’ reference the field
                                      // element f[1]
            f[1].y = 4;               // ’f[1]’ and ’c’ both contain
                                      // (3,4,2)
        }"
}
```

The following example illustrates uses of the fields and functions of SFVec3f and MFVec3f:

```
DEF SCR-VEC3F Script {
    eventIn SFTime touched1
    eventIn SFTime touched2
    eventIn SFTime touched3
    eventIn SFTime touched4
    eventOut SFVec3f new_translation
    field SFInt32 count 1
    field MFVec3f verts  []

    url "javascript:
        function initialize( ) {
            verts[0] = new SFVec3f(0, 0, 0);
            verts[1] = new SFVec3f(1, 1.732, 0);
            verts[2] = new SFVec3f(2, 0, 0);
            verts[3] = new SFVec3f(1, 0.577, 1.732);
        }

        function touched1 (value) {
            new_translation = verts[count]; // move sphere around tetra
            count++;
            if (count >= verts.length) count = 1;
        }

        function touched2 (value) {
            var tVec;
            tVec = new_translation.divide(2); // Zeno’s paradox to origin
            new_translation = new_translation.subtract(tVec);
        }

        function touched4 (value) {
            new_translation = new_translation.negate( );
        }

        function touched3 (value) {
            var a;
            a = verts[1].length( );
            a = verts[3].dot(verts[2].cross(verts[1]));
            a = verts[1].x;
            new_translation = verts[2].normalize( );
            new_translation = new_translation.add(new_translation);
        }"
}
```

Annex D
(informative)
Examples

D.1 Introduction and table of contents

This annex provides a variety of VRML examples.

D.2 Simple example

This example contains a simple scene defining a view of a red sphere and a blue box, lit by a directional light:

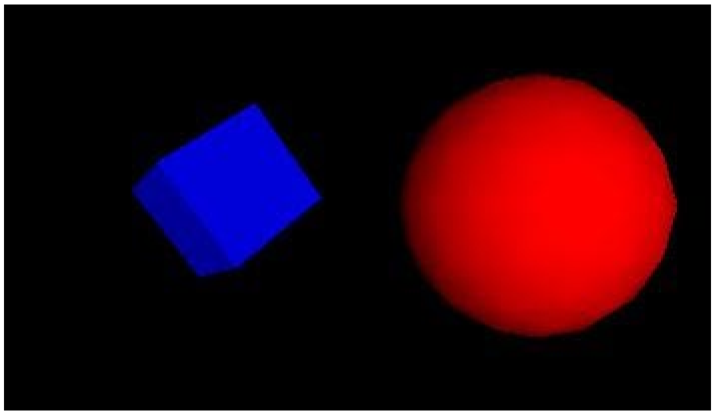


Figure D.1: Red sphere meets blue box

```
#VRML V2.0 utf8
Transform {
  children [
    NavigationInfo { headlight FALSE } # We’ll add our own light

    DirectionalLight {        # First child
        direction 0 0 -1      # Light illuminating the scene
    }

    Transform {               # Second child - a red sphere
      translation 3 0 1
      children [
        Shape {
          geometry Sphere { radius 2.3 }
          appearance Appearance {
            material Material { diffuseColor 1 0 0 }   # Red
         }
        }
      ]
    }

    Transform {               # Third child - a blue box
      translation -2.4 .2 1
      rotation     0 1 1  .9
      children [
        Shape {
          geometry Box {}
          appearance Appearance {
            material Material { diffuseColor 0 0 1 }  # Blue
         }
        }
      ]
    }

  ] # end of children for world
}
```

Click here to view this example in a VRML browser.

D.3 Instancing (sharing)

Reading the following file results in three spheres being drawn. The first sphere defines a unit sphere at the origin 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, will change too:

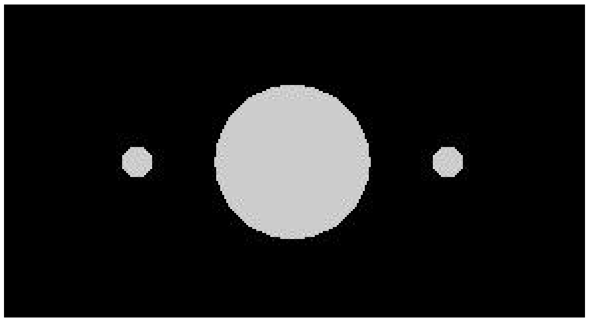


Figure D.2: Instancing

```
#VRML V2.0 utf8
Transform {
  children [
    DEF Joe Shape { geometry Sphere {} }
    Transform {
      translation 2 0 0
      children    DEF Joe Shape { geometry Sphere { radius .2 } }
    }
    Transform {
      translation -2 0 0
      children    USE Joe
    }

  ]
}
```

Click here to view this example in a VRML browser. (Note that the spheres are unlit because no appearance was specified.)

D.4 Prototype example

A simple table with variable colours for the legs and top might be prototyped as:

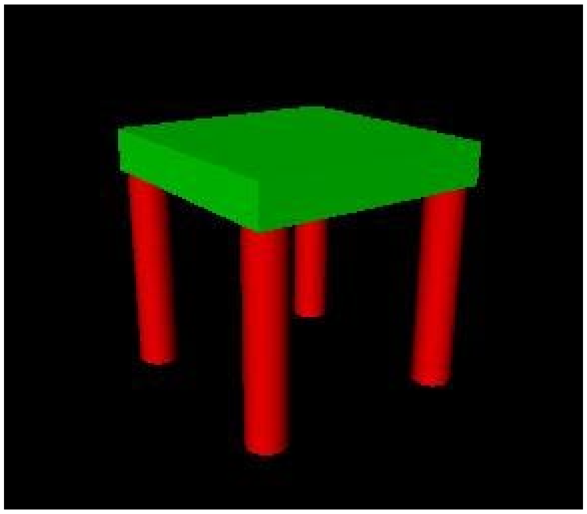


Figure D.3: Prototype

```
#VRML V2.0 utf8
PROTO TwoColorTable [ field SFColor legColor  .8 .4 .7
                      field SFColor topColor .6 .6 .1 ]
{
  Transform {
    children [
      Transform {   # table top
       translation 0 0.6 0
        children
          Shape {
            appearance Appearance {
              material Material { diffuseColor IS topColor }
            }
            geometry Box { size 1.2 0.2 1.2 }
          }
      }

      Transform {   # first table leg
       translation -.5 0 -.5
        children
          DEF Leg Shape {
            appearance Appearance {
              material Material { diffuseColor IS legColor }
            }
            geometry Cylinder { height 1 radius .1 }
          }
      }
      Transform {   # another table leg
       translation .5 0 -.5
        children USE Leg
      }
      Transform {   # another table leg
       translation -.5 0 .5
        children USE Leg
      }
      Transform {   # another table leg
       translation .5 0 .5
        children USE Leg
      }
    ] # End of root Transform's children
  } # End of root Transform
} # End of prototype

# The prototype is now defined. Although it contains a
# number of nodes, only the legColor and topColor fields
# are public. Instead of using the default legColor and
# topColor, this instance of the table has red legs and
# a green top:


TwoColorTable {
  legColor 1 0 0 topColor 0 1 0
}
NavigationInfo { type "EXAMINE" }      # Use the Examine viewer
```

Click here to view this example in a VRML browser.

D.5 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. The Script node combined with a Sphere, a TouchSensor and a Sound node to show how is works. When the pointing device is over the sphere, the *combinationEntered* eventIn of the Script is sent. Then, when the Sphere is touched (typically when the mouse button is pressed) the Script is sent the *openVault* eventIn. This generates the *vaultUnlocked* eventOut which starts a 'click' sound. Here is the example:

```
#VRML V2.0 utf8

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;
    }"
}

Shape {
    appearance Appearance {
        material Material { diffuseColor 1 0 0 }
    }
    geometry Sphere { }
}

Sound {
    source     DEF Click AudioClip {
             url       "click.wav"
             stopTime 1
    }

    minFront   1000
    maxFront   1000
    minBack    1000
    maxBack    1000
}


DEF TS TouchSensor { }

ROUTE TS.isOver TO OpenVault.combinationEntered
ROUTE TS.touchTime TO OpenVault.openVault
ROUTE OpenVault.vaultUnlocked TO Click.startTime
```

Note that the *openVault* eventIn and the *vaultUnlocked* eventOut are of type SFTime, which allows them to be wired directly to a TouchSensor or TimeSensor.

Click here to view this example in a VRML browser.

D.6 Geometric properties

The following IndexedFaceSet (contained in a Shape node) uses all four of the geometric property nodes to specify vertex coordinates, colours per vertex, normals per vertex, and texture coordinates per vertex (note that the material sets the overall transparency):

```
#VRML V2.0 utf8

Shape {
    geometry IndexedFaceSet {
        coordIndex [ 0, 1, 3, -1, 0, 2, 3, -1 ]
        coord Coordinate {
            point [ 0 0 0, 1 0 0, 1 0 -1, 0.5 1 0 ]
        }
        color Color {
            color [ 0.2 0.7 0.8, 0.5 0 0, 0.1 0.8 0.1, 0 0 0.7 ]
        }
        normal Normal {
            vector [ 0 0 1, 0 0 1, 0 0 1, 0 0 1 ]
        }
        texCoord TextureCoordinate {
            point [ 0 0, 1 0, 1 0.4, 1 1 ]
        }
    }
    appearance Appearance {
        material Material { transparency 0.5 }
        texture  PixelTexture {
            image 2 2 1 0xFF 0x80 0x80 0xFF
        }
    }
}
```

Click here to view this example in a VRML browser.

D.7 Prototypes and alternate representations

VRML 2.0 has the capability to define new nodes. The following is an example of a new node RefractiveMaterial. This node behaves as a Material node with an added field, *indexOfRefraction*. The list of URLs for the EXTERNPROTO are searched in order. If the browser recognizes the URN,

urn:inet:foo.com:types:RefractiveMaterial,

it may treat it as a native type (or load the implementation). Otherwise, the URL,

http://www.myCompany.com/vrmlNodes/RefractiveMaterial.wrl,

is used as a backup to ensure that the node is supported on any browsers. See below for the PROTO implementation that treats RefractiveMaterial as a Material (and ignores the *refractiveIndex* field).

```
#VRML V2.0 utf8

# external protype definition
EXTERNPROTO RefractiveMaterial [
    exposedField SFFloat ambientIntensity
    exposedField SFColor diffuseColor
    exposedField SFColor specularColor
    exposedField SFColor emissiveColor
    exposedField SFFloat shininess
    exposedField SFFloat transparency
    exposedField SFFloat indexOfRefraction  ]
[
  "urn:inet:foo.com:types:RefractiveMaterial",
  "http://www.myCompany.com/vrmlNodes/RefractiveMaterial.wrl",
  "refractivematerial.wrl",
]

Shape {
    geometry Sphere { }
    appearance Appearance {
        # Instance of a RefractiveMaterial
        material RefractiveMaterial {
            ambientIntensity  0.2
            diffuseColor      1 0 0
            indexOfRefraction 0.3
        }
    }
}
```

The URL `http://www.myCompany.com/vrmlNodes/RefractiveMaterial.wrl` contains the following:

```
#VRML V2.0 utf8

PROTO RefractiveMaterial [            # prototype definition
    exposedField SFFloat ambientIntensity  0
    exposedField SFColor diffuseColor      0.5 0.5 0.5
    exposedField SFColor specularColor     0 0 0
    exposedField SFColor emissiveColor     0 0 0
    exposedField SFFloat shininess         0
    exposedField SFFloat transparency      0
    exposedField SFFloat indexOfRefraction 0.1 ]
{
    Material {
        ambientIntensity IS ambientIntensity
        diffuseColor     IS diffuseColor
        specularColor    IS specularColor
        emissiveColor    IS emissiveColor
        shininess        IS shininess
        transparency     IS transparency
    }
}
```

Note that the name of the new node type, **RefractiveMaterial**, is not used by the browser to decide if the node is native or not; the URL/URN names determine the node's implementation.

Click here to view this example in a VRML browser.

D.8 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"
  parameter [ "target=name_of_frame" ]
  children Shape { geometry Cylinder {} }
}
```

An Anchor may be used to bind the viewer to a particular *viewpoint* in a virtual world by specifying a URL ending with *#viewpointName*, where *viewpointName* is the DEF name of a viewpoint defined in the world. For example:

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

specifies an anchor that puts the viewer in the *someScene* world bound to the viewpoint named *OverView* when the box is chosen (note that *OverView* is the DEF name of the viewpoint, not the value of the viewpoint's description field).

If no world is specified, the current scene is implied. For example:

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

binds the user’s view to the viewpoint with the DEF name *Doorway* in the current scene.

D.9 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 as shown below:

```
#VRML V2.0 utf8

Group {
    children [
       DEF UnlitShapeOne Transform {
            translation -3 0 0

           children Shape {
               appearance DEF App Appearance {
                   material Material {
                       diffuseColor 0.8 0.4 0.2
                   }
               }
               geometry Box { }
           }
        }

       DEF LitParent Group {
           children [
               DEF LitShapeOne Transform {
                    translation 0 2 0

                   children Shape {
                       appearance USE App
                       geometry Sphere { }
                   }
               }

               # lights the shapes under LitParent
               DirectionalLight { }
               DEF LitShapeTwo Transform {
                    translation 0 -2 0

                   children Shape {
                       appearance USE App
                       geometry Cylinder { }
                    }
                }
           ]
       }

       DEF UnlitShapeTwo Transform {
            translation 3 0 0

           children Shape {
               appearance USE App
               geometry Cone { }
            }
       }
    ]
}
```

Click here to view this example in a VRML browser.

D.10 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 colours:

```
#VRML V2.0 utf8

Shape {
    geometry PointSet {
        coord DEF mypts Coordinate {
            point [ 0 0 0, 2 2 2, 3 3 3 ]
        }
        color Color { color [ 1 0 0, 0 1 0, 0 0 1 ] }
    }
}

Transform {
    translation 2 0 0

    children Shape {
        geometry PointSet {
           coord USE mypts
           color Color { color [ .5 .5 0, 0 .5 .5, 1 1 1 ] }
        }
    }
}
```

Click here to view this example in a VRML browser.

D.11 Level of detail

The LOD node is typically used for switching between different versions of geometry at specified distances from the viewer. However, 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 10 meters (as measured in the coordinate space of the LOD). The browser should display the less detailed version of the object if the viewer is between 10 and 50 meters and should display nothing at all if the viewer is farther than 50 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.

```
#VRML V2.0 utf8

LOD {
    range [ 10, 50 ]
    level [
        LOD {
            level [
                Shape { geometry Sphere { } }
                DEF LoRes Shape { geometry Box { } }
            ]
        }
        USE LoRes,
        Shape { } # Display nothing
    ]
}
```

For best results, ranges should be specified only where necessary and LOD nodes should be nested with and without ranges.

Click here to view this example in a VRML browser.

D.12 Color interpolator

This example interpolates from red to green to blue in a 10 second cycle:

```
#VRML V2.0 utf8

DEF myColor ColorInterpolator {
    key         [   0.0,    0.5,    1.0 ]
    keyValue    [ 1 0 0,  0 1 0,  0 0 1 ] # red, green, blue
}

DEF myClock TimeSensor {
    cycleInterval 10.0      # 10 second animation
    loop          TRUE      # infinitely cycling animation
}

Shape {
    appearance Appearance {
        material DEF myMaterial Material { }
    }
    geometry Sphere { }
}

ROUTE myClock.fraction_changed TO myColor.set_fraction
ROUTE myColor.value_changed TO myMaterial.set_diffuseColor
```

Click here to view this example in a VRML browser.

D.13 TimeSensor

D.13.1 Introduction

The TimeSensor is very flexible. The following are some of the many ways in which it can be used:

e. 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 be routed to the TimeSensor's *startTime* to start the TimeSensor running).

f. a TimeSensor can be made to run continuously upon reading by setting *cycleInterval* > 0, *startTime* > 0, *stopTime* = 0, and *loop* = TRUE.

D.13.2 Click to animate

The first example animates a box when the user clicks on it:

```
#VRML V2.0 utf8

DEF XForm Transform {
    children [
        Shape {
            appearance Appearance {
                material Material { diffuseColor 1 0 0 }
            }
            geometry Box {}
        }
        DEF Clicker TouchSensor {}

        # Run once for 2 sec.
        DEF TimeSource TimeSensor { cycleInterval 2.0 }

        # Animate one full turn about Y axis:
        DEF Animation OrientationInterpolator {
            key      [ 0,      .33,       .66,        1.0 ]
            keyValue [ 0 1 0 0, 0 1 0 2.1, 0 1 0 4.2, 0 1 0 0 ]
        }
    ]
}
ROUTE Clicker.touchTime TO TimeSource.startTime
ROUTE TimeSource.fraction_changed TO Animation.set_fraction
ROUTE Animation.value_changed TO Xform.rotation
```

Click here to view this example in a VRML browser.

D.13.3 Alarm clock

The second example plays chimes once an hour:

```
#VRML V2.0 utf8

Group {
    children [
        DEF Hour TimeSensor {
            loop         TRUE
            cycleInterval 3600.0 # 60*60 seconds == 1 hour
        }

        Sound {
            source DEF Sounder AudioClip {
                url "click.wav" }
            }
        }
    ]
}

ROUTE Hour.cycleTime TO Sounder.startTime
```

Click here to view this example in a VRML browser.

D.14 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 (by default). The *distance* field can be used to change this default. The pendulum rotates its children about the Z axis, from 0 to 3.14159 radians and back again (by default). The *maxAngle* field can be used to change this default.

```
#VRML V2.0 utf8

PROTO Shuttle [
    field        SFTime  rate      1
    field        SFFloat distance  1
    field        MFNode  children  [ ]
    exposedField SFTime  startTime 0
    exposedField SFTime  stopTime  0
    field        SFBool  loop      TRUE
] {
    DEF F Transform { children IS children }
    DEF T TimeSensor {
        cycleInterval IS rate
        startTime IS startTime
        stopTime IS stopTime
        loop IS loop
    }

    DEF S Script {
        field    SFFloat    distance IS distance
        eventOut MFVec3f    position

        url "javascript:
            function initialize() {
                // constructor:setup interpolator,
                pos1 = new SFVec3f(-distance, 0, 0);
                pos2 = new SFVec3f(distance, 0, 0);
                position = new MFVec3f(pos1, pos2, pos1);
            }",
    }

    DEF I PositionInterpolator {
        key [ 0, 0.5, 1 ]
        keyValue [ -1 0 0, 1 0 0, -1 0 0 ]
    }

    ROUTE T.fraction_changed TO I.set_fraction
    ROUTE I.value_changed TO F.set_translation
    ROUTE S.position TO I.set_keyValue
}

PROTO Pendulum [
    field        SFTime  rate      1
    field        SFFloat maxAngle  3.14159
    field        MFNode  children  [ ]
    exposedField SFTime  startTime 0
    exposedField SFTime  stopTime  0
    field        SFBool  loop      TRUE
] {
    DEF F Transform { children IS children }
    DEF T TimeSensor {
        cycleInterval IS rate
        startTime IS startTime
        stopTime IS stopTime
        loop IS loop
    }
    DEF S Script {
        field    SFFloat    maxAngle IS maxAngle
        eventOut MFRotation rotation

        url "javascript:
            function initialize() {
                // constructor:setup interpolator,
                rot1 = new SFRotation(0, 0, 1, 0);
                rot2 = new SFRotation(0, 0, 1, maxAngle/2);
                rot3 = new SFRotation(0, 0, 1, maxAngle);
                rotation = new MFRotation(rot1, rot2, rot3,
                                          rot2, rot1);
            }",
    }
    DEF I OrientationInterpolator {
        key [ 0, 0.25, 0.5, 0.75, 1 ]
        keyValue [ 0 0 1 0,
                   0 0 1 1.57,
                   0 0 1 3.14,
                   0 0 1 1.57,
                   0 0 1 0 ]
    }

    ROUTE T.fraction_changed TO I.set_fraction
    ROUTE I.value_changed TO F.set_rotation
    ROUTE S.rotation TO I.set_keyValue
}

Transform {
    translation -3 0 0
    children Pendulum {
        rate 3
        maxAngle 6.28
        children Shape { geometry Cylinder { height 5 } }
    }
}

Transform {
    translation 3 0 0
    children Shuttle {
        rate 2
        children Shape { geometry Sphere { } }
    }
}
```

Click here to view this example in a VRML browser.

These nodes can be used to do a continuous animation when *loop* is TRUE. When *loop* is FALSE they can perform a single cycle under control of the *startTime* and *stopTime* fields. The *rate* field controls the speed of the animation. The *children* field holds the children to be animated.

D.15 Robot

This example is a simple implementation of a robot. 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 and walks back across a path. He does this whenever the viewer is near. This makes use of the Shuttle and Pendulum of D.14.

```
#VRML V2.0 utf8

EXTERNPROTO Shuttle [
    field        SFTime  rate
    field        SFFloat distance
    field        MFNode  children
    exposedField SFTime  startTime
    exposedField SFTime  stopTime
    field        SFBool  loop
]
"exampleD.14.wrl#Shuttle"

EXTERNPROTO Pendulum [
    field        SFTime  rate
    field        SFFloat maxAngle
    field        MFNode  children
    exposedField SFTime  startTime
    exposedField SFTime  stopTime
    field        SFBool  loop
]
"exampleD.14.wrl#Pendulum"

Viewpoint {
    position 0 0 150
}

DEF Near ProximitySensor { size 200 200 200 }

DEF Walk Shuttle {
    stopTime 1
    rate 10
    distance 20

    children [
        # The Robot
        Transform {
            rotation 0 1 0 1.57

            children [
                Shape {
                    appearance DEF A Appearance {
                        material Material {
                            diffuseColor 0 0.5 0.7
                        }
                    }
                    geometry Box { } # head
                }
                Transform {
                    scale 1 5 1
                    translation 0 -5 0
                    children Shape {
                        appearance USE A
                        geometry Sphere { }
                    } # body
                }
                Transform {
                    rotation 0 1 0 1.57
                    translation 1.5 0 0

                    children DEF Arm Pendulum {
                        stopTime 1
                        rate 1
                        maxAngle 0.52 # 30 degrees

                        children [
                            Transform {
                                translation 0 -3 0

                                children Shape {
                                    appearance USE A
                                    geometry Cylinder {
                                        height 4
                                        radius 0.5
                                    }
                                }
                            }
                        ]
                    }
                }

                # duplicate arm on other side and flip so
                # it swings in opposition
                Transform {
                    rotation 0 -1 0 1.57
                    translation -1.5 0 0
                    children USE Arm
                }
            ]
        }
    ]
}

ROUTE Near.enterTime TO Walk.startTime
ROUTE Near.enterTime TO Arm.startTime
ROUTE Near.exitTime TO Walk.stopTime
ROUTE Near.exitTime TO Arm.stopTime
```

Click here to view this example in a VRML browser.

Move closer to the robot to start the animation.

D.16 Chopper

This example of a helicopter demonstrates how to do simple animation triggered by a TouchSensor. It uses an EXTERNPROTO to include a Rotor node from the Internet which does the actual animation.

```
#VRML V2.0 utf8

EXTERNPROTO Rotor [
    field        SFTime  rate
    field        MFNode  children
    exposedField SFTime  startTime
    exposedField SFTime  stopTime
]
"rotor.wrl"

PROTO Chopper [
    field SFTime rotorSpeed 1
] {
    Group {
        children [
            DEF Touch TouchSensor { } # Gotta get touch events
            Inline { url "chopperbody.wrl" }
            DEF Top Rotor {
                # initially, the rotor should not spin
                stopTime 1
                rate IS rotorSpeed
                children Inline { url "chopperrotor.wrl" }
            }
        ]
    }

    DEF RotorScript Script {
        eventIn  SFTime startOrStopEngine
        eventOut SFTime startEngine
        eventOut SFTime stopEngine
        field    SFBool engineStarted FALSE

        url "javascript:
            function startOrStopEngine(value) {
                // start or stop engine:
                if (!engineStarted) {
                    startEngine = value;
                    engineStarted = TRUE;
                }
                else {
                    stopEngine = value;
                    engineStarted = FALSE;
                }
            }"
    }

    ROUTE Touch.touchTime TO RotorScript.startOrStopEngine
    ROUTE RotorScript.startEngine TO Top.startTime
    ROUTE RotorScript.stopEngine TO Top.stopTime
}

Viewpoint { position 0 0 5 }
DEF MyScene Group {
    children DEF MikesChopper Chopper { }
}
```

Click here to view this example in a VRML browser.

D.17 Guided tour

VRML provides control of the viewer's camera through use of a script. This is useful for things such as guided tours, merry-go-round rides, and transportation devices such as buses and elevators. These next two examples show a couple of ways to use this feature.

This example is a simple guided tour through the world. Upon entry, a guide orb hovers in front of the viewer. Click on this and a tour through the world begins. The orb follows the user around on his tour. A ProximitySensor ensures that the tour is started only if the user is close to the initial starting point. Note that this is done without scripts thanks to the *touchTime* output of the TouchSensor.

```
#VRML V2.0 utf8

Group {
    children [
        Transform {
            translation 0 -1 0

            children Shape {
                appearance Appearance {
                    material Material { }
                }
                geometry Box { size 30 0.2 30 }
            }
        }
        Transform {
            translation -1 0 0

            children Shape {
                appearance Appearance {
                    material Material {
                        diffuseColor 0.5 0.8 0
                    }
                }
                geometry Cone { }
            }
        }
        Transform {
            translation 1 0 0

            children Shape {
                appearance Appearance {
                    material Material {
                        diffuseColor 0 0.2 0.7
                    }
                }
                geometry Cylinder { }
            }
        }

        DEF GuideTransform Transform {
            children [
                DEF TourGuide Viewpoint { jump FALSE },
                DEF ProxSensor ProximitySensor { size 50 50 50 }
                DEF StartTour TouchSensor { },
                Transform {
                    translation 0.6 0.4 8

                    children Shape {
                        appearance Appearance {
                            material Material {
                                diffuseColor 1 0.6 0
                            }
                        }
                        geometry Sphere { radius 0.2 }
                    } # the guide orb
                }
            ]
        }
    ]
}

DEF GuidePI PositionInterpolator {
    key [ 0, 0.2, 0.3, 0.5, 0.6, 0.8, 0.9, 1 ]
    keyValue [ 0 0 0, 0 0 -5,
               2 0 -5, 2 6 -15
               -4 6 -15, -4 0 -5,
               0 0 -5, 0 0 0
    ]
}

DEF GuideRI OrientationInterpolator {
    key [ 0, 0.2, 0.3, 0.5, 0.6, 0.8, 0.9, 1 ]
    keyValue [ 0 1 0 0, 0 1 0 0,
               0 1 0 1.2, 0 1 0 3,
               0 1 0 3.5, 0 1 0 5,
               0 1 0 0, 0 1 0 0,
    ]
}

DEF TS TimeSensor { cycleInterval 30 } # 60 second tour

ROUTE ProxSensor.isActive TO StartTour.set_enabled
ROUTE StartTour.touchTime TO TS.startTime
ROUTE TS.isActive TO TourGuide.set_bind
ROUTE TS.fraction_changed TO GuidePI.set_fraction
ROUTE TS.fraction_changed TO GuideRI.set_fraction
ROUTE GuidePI.value_changed TO GuideTransform.set_translation
ROUTE GuideRI.value_changed TO GuideTransform.set_rotation
```

Click here to view this example in a VRML browser.

D.18 Elevator

This is another example of animating the camera by depicting an elevator to ease access to a multi-storey building. For this example, a 2 storey building is shown and it is assumed that the elevator is already at the ground floor. To go up, the user just steps onto the elevator platform. A ProximitySensor fires and starts the elevator up automatically. Additional features such as call buttons for outside the elevator, elevator doors, and floor selector buttons could be added to make the elevator easier to use.

```
#VRML V2.0 utf8

Transform {
    translation 0 0 -3.5

    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 1 0
            }
        }
        geometry Cone { }
    }
}

Transform {
    translation 0 4 -3.5

    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 1 0 0
            }
        }
        geometry Cone { }
    }
}

Transform {
    translation 0 8 -3.5

    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 0 1
            }
        }
        geometry Cone { }
    }
}

Group {
    children [
        DEF ETransform Transform {
            children [
                DEF EViewpoint Viewpoint { jump FALSE }
                DEF EProximity ProximitySensor { size 2 5 5 }
                Transform {
                    translation 0 -1 0

                    children Shape {
                        appearance Appearance {
                            material Material { }
                        }
                        geometry Box { size 2 0.2 5 }
                    }
                }
            ]
        }
    ]
}

DEF ElevatorPI PositionInterpolator {
    key [ 0, 1 ]
    keyValue [ 0 0 0, 0 8 0 ] # a floor is 4 meters high
}
DEF TS TimeSensor { cycleInterval 10 } # 10 second travel time
```

```
ROUTE EProximity.enterTime TO TS.startTime
ROUTE TS.isActive TO EViewpoint.set_bind
ROUTE TS.fraction_changed TO ElevatorPI.set_fraction
ROUTE ElevatorPI.value_changed TO ETransform.set_translation
```

Click here to view this example in a VRML browser.

D.19

This example illustrates the execution model example described in 4.10.3, Execution model.

```
#VRML V2.0 utf8
DEF TS TouchSensor { }
DEF Script1 Script {
    eventIn  SFTime touchTime
    eventOut SFBool toScript2
    eventOut SFBool toScript3
    eventOut SFString string
    url "javascript:
        function touchTime() {
            toScript2 = TRUE;
        }
        function eventsProcessed() {
            string = 'Script1.eventsProcessed';
            toScript3 = TRUE;
        }"
}
DEF Script2 Script {
   eventIn  SFBool fromScript1
   eventOut SFBool toScript4
    eventOut SFString string
    url "javascript:
        function fromScript1() {
        }
        function eventsProcessed() {
            string = 'Script2.eventsProcessed';
            toScript4 = TRUE;
        }"
}
DEF Script3 Script {
   eventIn  SFBool fromScript1
   eventOut SFBool toScript5
   eventOut SFBool toScript6
   eventOut SFString string
    url "javascript:
        function fromScript1() {
            toScript5 = TRUE;
        }
        function eventsProcessed() {
            string = 'Script3.eventsProcessed';
            toScript6 = TRUE;
        }"
}
```

```
DEF Script4 Script {
    eventIn SFBool fromScript2
    url "javascript:
        function fromScript2() {
        }"
}
DEF Script5 Script {
    eventIn SFBool fromScript3
    url "javascript:
        function fromScript3() {
        }"
}
DEF Script6 Script {
    eventIn  SFBool fromScript3
    eventOut SFBool toScript7
    eventOut SFString string
    url "javascript:
        function fromScript3() {
            toScript7 = TRUE;
        }
        function eventsProcessed() {
            string = ’Script6.eventsProcessed’;
        }"
}
DEF Script7 Script {
    eventIn  SFBool fromScript6
    url "javascript:
        function fromScript6() {
        }"
}
ROUTE TS.touchTime TO Script1.touchTime
ROUTE Script1.toScript2 TO Script2.fromScript1
ROUTE Script1.toScript3 TO Script3.fromScript1
ROUTE Script2.toScript4 TO Script4.fromScript2
ROUTE Script3.toScript5 TO Script5.fromScript3
ROUTE Script3.toScript6 TO Script6.fromScript3
ROUTE Script6.toScript7 TO Script7.fromScript6

# Display the results
DEF Collector Script {
    eventOut   MFString string
    eventIn SFString fromString
    url "javascript:
        function initialize() { string[0] = ’Event Sequence:’; }
        function fromString(s) {
            i = string.length;
            string[i] = ’    ’+i+’) ’+s+’ occurred’;
        }"
}
Transform {
    translation 0 2 0
    children Shape {
        appearance Appearance {
            material Material { diffuseColor 0 0.6 0 }
        }
        geometry Sphere { }
    }
```

```
}
Shape { geometry DEF Result Text { } }
Viewpoint { position 7 -1 18 }
ROUTE Script1.string TO Collector.fromString
ROUTE Script2.string TO Collector.fromString
ROUTE Script3.string TO Collector.fromString
ROUTE Script6.string TO Collector.fromString
ROUTE Collector.string TO Result.string
```

Click here to view this example in a VRML browser.

Clicking on the green sphere should display a text string for each eventsProcessed event. The two possible correct displays for this example are:

```
Event Sequence:
  1) Script1.eventsProcessed occurred
  2) Script2.eventsProcessed occurred
  3) Script3.eventsProcessed occurred
  4) Script6.eventsProcessed occurred
```

or

```
Event Sequence:
  1) Script2.eventsProcessed occurred
  2) Script1.eventsProcessed occurred
  3) Script3.eventsProcessed occurred
  4) Script6.eventsProcessed occurred
```

Annex E
(informative)

Bibliography

This annex contains the informative references in this part of ISO/IEC 14772. These are references to unofficial standards or documents. All official standards are referenced in 2, Normative references.

Identifier	Reference
DATA	"The Data: URL scheme," IETF Internet Draft working document. http://ds.internic.net/internet-drafts/draft-masinter-url-data-03.txt
FOLE	Foley, van Dam, Feiner and Hughes, Computer Graphics Principles and Practice, 2nd Edition, Addison Wesley, Reading, MA, 1990. http://www.awl.com
GIF	"GIF™ - Graphics Interchange Format™" - A standard defining a mechanism for the storage and transmission of raster-based graphics information, Version 89a, CompuServe. http://www.w3.org/pub/WWW/Graphics/GIF/spec-gif89a.txt
JAPI	"The Java™ Application Programming Interface, Volume 1 Core Packages" by James Gosling, Frank Yellin and The Java Team, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63453-8. http://java.sun.com/docs/books/apis/index.html "The Java™ Application Programming Interface, Volume 2 Window Toolkit and Applets" by James Gosling, Frank Yellin and The Java Team, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63459-7. http://java.sun.com/docs/books/apis/index.html
MIME	"The Model Primary Content Type for Multipurpose Internet Mail Extensions," IETF Internet standards track protocol. http://ds.internic.net/rfc/rfc2077.txt
OPEN	"The OpenGL Graphics System: A Specification (Version 1.1)," Silicon Graphics, Inc., 1995. http://www.sgi.com/Technology/openGL/glspec1.1/glspec.html
PERL	"Programming Perl" by Larry Wall, Tom Christiansen and Randal L. Schwartz, O'Reilly & Associates, Sebastapol, CA, 1996. http://www.oreilly.com/
SNDA	"Fundamentals of Computer Music", Dodge & Jerse, Shirmer Books, New York, 1985, pp 20-21.
SNDB	Spatial Audio Work in the Multimedia Computing Group, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA. http://www.cc.gatech.edu/gvu/multimedia/spatsound/spatsound.html

URN	"Universal Resource Name," IETF Internet standards track protocol. http://ds.internic.net/rfc/rfc2141.txt, http://services.bunyip.com:8000/research/ietf/urn-ietf/
WAV	"Waveform Audio File Format, Multimedia Programming Interface and Data Specification v1.0", Issued by IBM & Microsoft, 1991. ftp://ftp.cwi.nl/pub/audio/RIFF-format, http://keck.ucsf.edu/~jwright/RIFF-format.html, http://www.seanet.com/HTML/Users/matts/riffmci/riffmci.htm

Annex F
(informative)

Recommendations for non-normative extensions

F.1 Introduction

This annex describes recommended practice for non-normative extensions to ISO/IEC 14772.

F.2 URNs

URNs are location-independent pointers to a file or to different representations of the same content. In most ways, URNs can be used like URLs except that, when fetched, a smart browser should fetch them from the closest source. URN resolution over the Internet has not yet been standardized. However, URNs may be used now as persistent unique identifiers for referenced entities such as files, EXTERNPROTOs, and textures. General information on URNs is available at E.[URN].

URNs may be assigned by anyone with a domain name. For example, if the company Foo owns foo.com, it may allocate URNs that begin with "urn:inet:foo.com:". An example of such usage is

"urn:inet:foo.com:texture:wood001".

See the draft specification referenced in E.[URN] for a description of the legal URN syntax.

To reference a texture, EXTERNPROTO, or other file by a URN, the URN is included in the *url* field of another node. For example:

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

specifies a URL file as the first choice and a URN as the second choice.

F.3 Browser extensions

Browsers that wish to add functionality beyond the capabilities of ISO/IEC 14772 can 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 in the form of a VRML scene graph), it can 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 content to ensure it is viewable on all browsers.

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

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

This browser will recognize the URN and use the URN resource's 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 are found, the Torus is assumed to be an empty node.

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

```
    #VRML V2.0 utf8
    EXTERNPROTO Ring [field SFFloat bigR, field SFFloat smallR ]
      ["urn:browser.com:library:Torus", "http://.../proto_torus.wrl" ]
    EXTERNPROTO Donut [field SFFloat bigR, field SFFloat smallR ]
      ["urn:browser.com:library:Torus", "http://.../proto_torus.wrl" ]

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