VRML97 specification

Nodes reference

Masaryk University

June 22, 2000

typeset by David Šafránek using ConTEXt   contact: dawe@mail.muni.cz

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

Introduction

This document provides a detailed definition of the syntax and semantics of each node in this part of ISO/IEC 14772. 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 $(-\infty, 1]$ defines the lower bound as $-\infty$ exclusively and the upper bound as 1 inclusively.

For example, this table defines the Collision node declaration:

```
Collision {
      eventIn       MFNode   addChildren
      eventIn       MFNode   removeChildren
      exposedField  MFNode   children         []
      exposedField  SFBool   collide
      field         SFVec3f  bboxCenter       0 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:

a. eventIns, in alphabetical order
b. exposedFields, in alphabetical order
c. fields, in alphabetical order
d. eventOuts, in alphabetical order

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

Anchor

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

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

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

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

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

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

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

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

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

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

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


go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

binds viewer to the viewpoint defined by the "Doorway" viewpoint in the current world when the sphere is activated. In this case, if the Viewpoint is not found, then do nothing on activation. See **"Concepts – URLs and URNs"** for more details on the `url` field.

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

Appearance

Appearance {

exposedField	SFNode	material	NULL
exposedField	SFNode	texture	NULL
exposedField	SFNode	textureTransform	NULL

}

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

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

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

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

AudioClip

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

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

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

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

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

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

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

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

$$fmod(now - -\texttt{startTime}, duration/\texttt{pitch})$$

Background
Billboard
Box

Background

```
Background {
    eventIn        SFBool     set_bind
    exposedField   MFFloat    groundAngle   []
    exposedfield   MFColor    groundColor   []
    exposedField   MFString   backUrl       []
    exposedField   MFString   bottomUrl     []
    exposedField   MFString   frontUrl      []
    exposedField   MFString   leftUrl       []
    exposedField   MFString   rightUrl      []
    exposedField   MFString   topUrl        []
    exposedField   MFFloat    skyAngle      []
    exposedField   MFColor    skyColor      [ 0 0 0 ]
    eventOut       SFBool     isBound
}
```

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

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

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

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

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

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

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

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

Ground colors, sky colors, and panoramic images do not translate with respect to the viewer, though they do rotate with respect to the viewer. That is, the viewer can never get any closer to

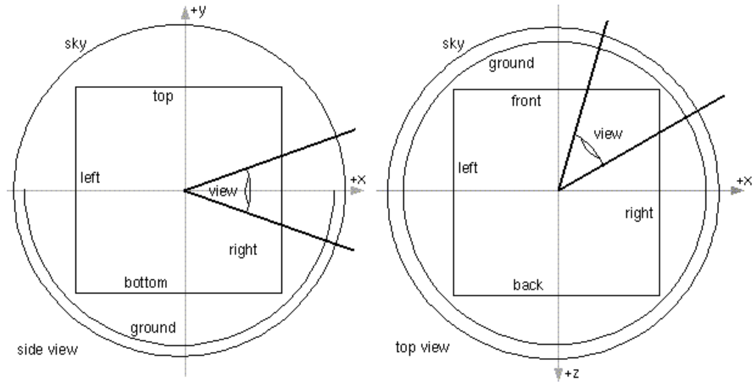


Figure 3.1

the background, but can turn to examine all sides of the panorama cube, and can look up and down to see the concentric rings of ground and sky (if visible).

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

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

Billboard

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

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

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

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

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

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

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

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

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

Box

Box {
    field SFVec3f size 2 2 2
  }

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

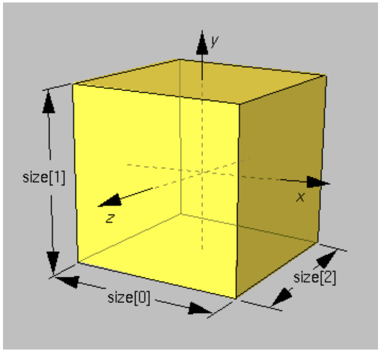


Figure 3.2

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

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

Collision

```
Collision {
    eventIn        MFNode   addChildren
    eventIn        MFNode   removeChildren
    exposedField   MFNode   children        []
    exposedField   SFBool   collide
    field          SFVec3f  bboxCenter      0 0 0       # (-∞,0)
    field          SFVec3f  bboxSize        -1 -1 -1    # (0,∞) or -1,-1,-1
    field          SFNode   proxy           NULL
    eventOut       SFTime   collideTime
}
```

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

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

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

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

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

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

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

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

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

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

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

Color

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

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

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

ColorInterpolator

```
ColorInterpolator {
        eventIn        SFFloat   set_fraction
        exposedField   MFFloat   key             []
        exposedField   MFColor   keyValue        []
        eventOut       SFColor   value_changed
}
```

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

Cone

Cone {

`field`	SFFloat	bottomRadius	1
`field`	SFFloat	height	2
`field`	SFBool	side	`TRUE`
`field`	SFBool	bottom	`TRUE`

}

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

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

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

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

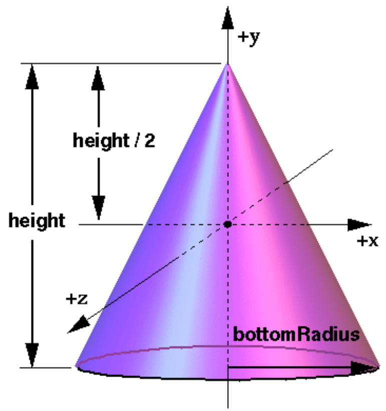


Figure 4.1

Coordinate

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

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

CoordinateInterpolator

CoordinateInterpolator {

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

}

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

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

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

Cylinder

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

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

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

When a texture is applied to a cylinder, it is applied differently to the sides, top, and bottom. On the sides, the texture wraps counterclockwise (from above) starting at the back of the cylinder. The texture has a vertical seam at the back, intersecting the YZ-plane. For the top and bottom caps, a circle is cut out of the unit texture square centered at $(0, +/-\text{height}, 0)$ with dimensions $2 * \text{radius}$ by $2 * \text{radius}$. 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.

go back
contents
a n b o c p d q e r f s g t h u i v j w k x l y m z
find
print
close

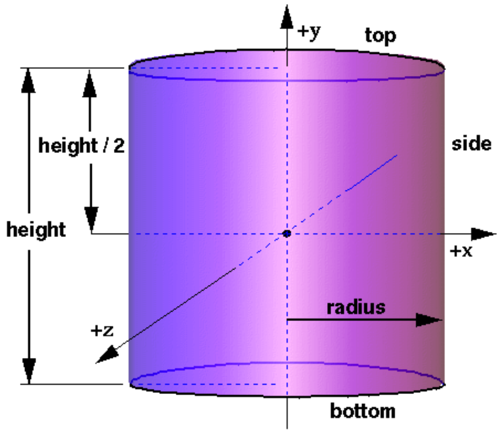


Figure 4.2

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

CylinderSensor

CylinderSensor {

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

}

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

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

The CylinderSensor generates events if the pointing device is activated while over any descendant geometry nodes of its parent group and then moved while activated. Typically, the pointing device is a 2D device such as a mouse. The pointing device is considered to be moving within a plane at a fixed distance from the viewer and perpendicular to the line of sight; this establishes a set of 3D coordinates for the pointer. If a 3D pointer is in use, then the sensor generates events

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

only when the pointer is within the user's field of view. In either case, the pointing device is considered to "pass over" geometry when that geometry is intersected by a line extending from the viewer and passing through the pointer's 3D coordinates. If multiple sensors' geometry intersect this line (hereafter called the bearing), only the nearest will be eligible to generate events.

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

and autoOffset is TRUE, offset is set to the last rotation angle and an offset_changed event is generated. See "**Concepts – Drag Sensors**" for more details.

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

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

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

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

DirectionalLight

DirectionalLight

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

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

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

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

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

ElevationGrid
Extrusion

ElevationGrid

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

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

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

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

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

$P[i, j].x = \text{xSpacing} * i$

$P[i, j].y = \text{height}[i + j * \text{zDimension}]$

$P[i, j].z = \text{zSpacing} * j$

where xDimension and zDimension.

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

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

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

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

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

Extrusion

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

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

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

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

1. Start with the cross section as specified, in the XZ plane.
2. Scale it about $(0, 0, 0)$ by the value for `scale` given for the current joint.
3. Apply a rotation so that when the cross section is placed at its proper location on the spine it will be oriented properly. Essentially, this means that the cross section's Y axis (up vector coming out of the cross section) is rotated to align with an approximate tangent to the spine curve. *For all points other than the first or last:* The tangent for $spine[i]$ is found by normalizing the vector defined by $(spine[i+1] - spine[i-1])$.

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

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

   In the simple case where the spine curve is flat in the XY plane, these rotations are all just rotations about the Z axis. In the more general case where the spine curve is any 3D curve, you need to find the destinations for all 3 of the local X, Y, and Z axes so you can completely specify the rotation. The Z axis is found by taking the cross product of: $(spine[i-1] - -spine[i])$ and $(spine[i+1] - -spine[i])$. If the three points are collinear then this value is zero, so take the value from the previous point. Once you have the Z axis (from the cross product) and the Y axis (from the approximate tangent), calculate the X axis as the cross product of the Y and Z axes.

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

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

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

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

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

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

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

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

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

Fog
FontStyle

Fog

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

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

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

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

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

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

FontStyle

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

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

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

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

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

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

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

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

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

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

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

Major Alignment, horizontal = TRUE:

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

Major Alignment, horizontal = FALSE:

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

Minor Alignment, horizontal = TRUE:

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

Minor Alignment, horizontal = FALSE:

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

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

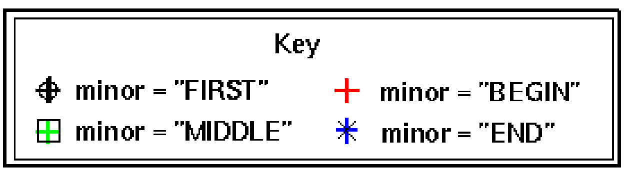


Figure 7.1

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

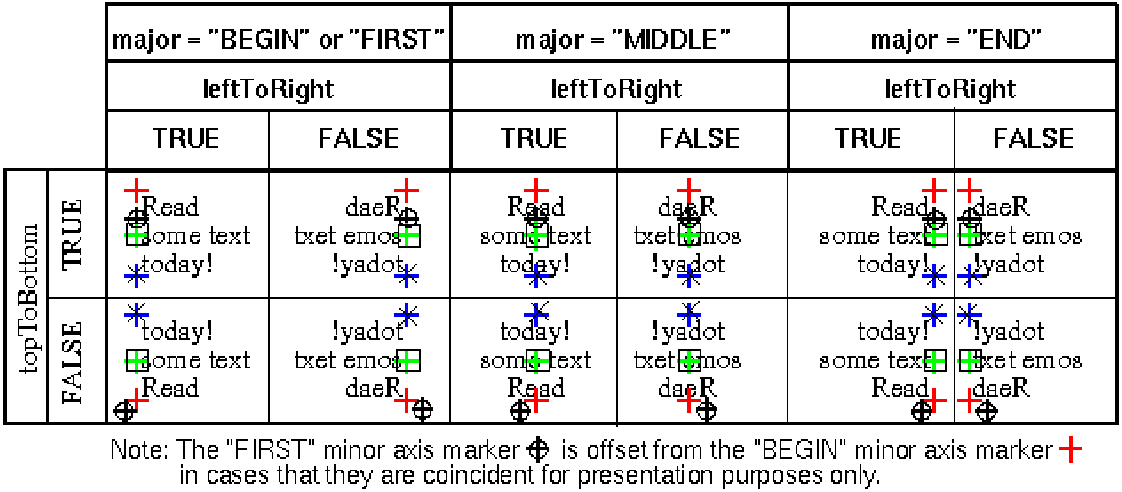
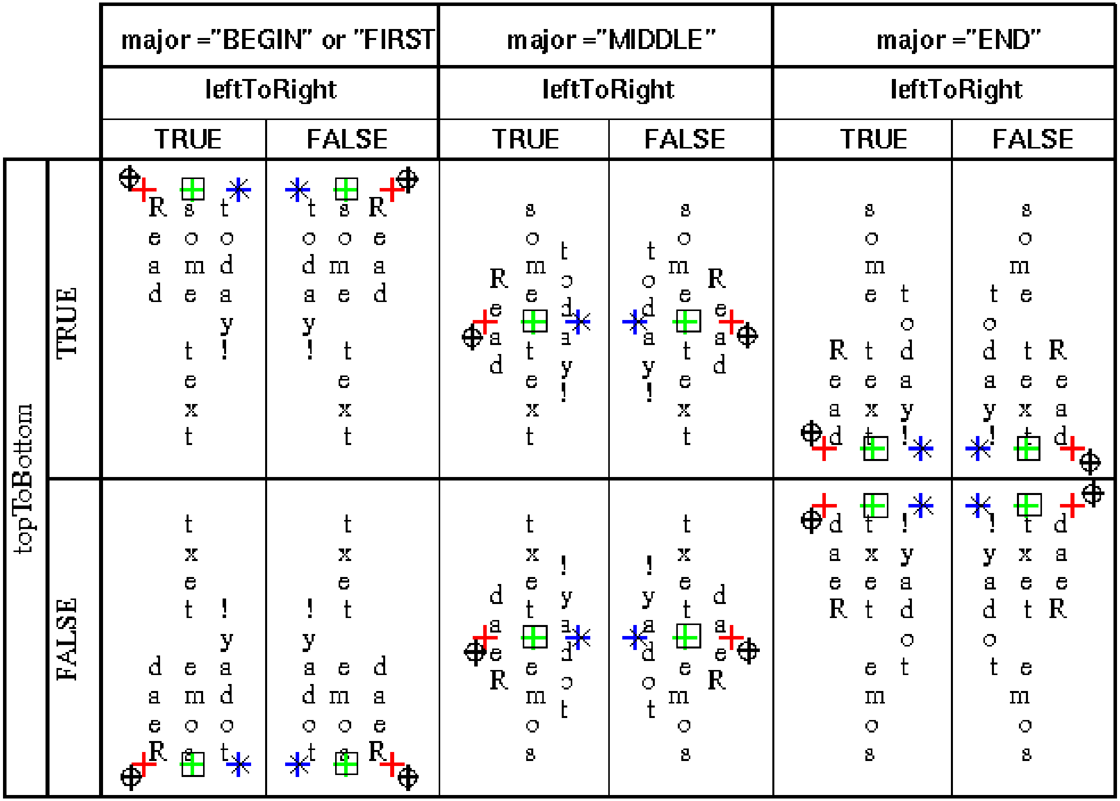


Figure 7.2 horizontal = TRUE

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

Please refer to these sites for more details:

www.chemie.fu-berlin.de (ISO 639), www.chemie.fu-berlin.de (ISO 3166)



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

Figure 7.3 horizontal = FALSE

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

Group

Group

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

A Group node is equivalent to a Transform node, without the transformation fields. See the "**Concepts – Grouping and Children Nodes**" section for a description of the `children`, `addChildren`, and `removeChildren` fields and eventIns.

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

ImageTexture

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

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

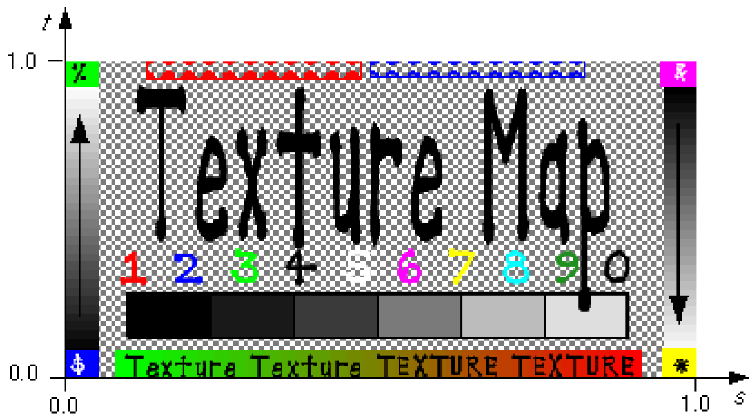


Figure 9.1

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

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

1. Diffuse color is multiplied by the greyscale values in the texture image.
2. Diffuse color is multiplied by the greyscale values in the texture image; material transparency is multiplied by transparency values in texture image.
3. RGB colors in the texture image replace the material's diffuse color.
4. RGB colors in the texture image replace the material's diffuse color; transparency values in the texture image replace the material's transparency.

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

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

IndexedFaceSet

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

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

Coordinate node must contain $N+1$ coordinates (indexed as 0-N). IndexedFaceSet is specified in the local coordinate system and is affected by parent transformations.

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

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

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

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

markers (-1) in exactly the same places as the coordIndex field. If the greatest index in the colorIndex field is N, then there must be $N+1$ colors in the **Color** node.

- If the colorIndex field is empty, then the coordIndex field is used to choose colors from the **Color** node. If the greatest index in the coordIndex field is N, then there must be $N+1$ colors in the **Color** node.

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

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

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

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

If the texCoord field is NULL, a default texture coordinate mapping is calculated using the bounding box of the shape. The longest dimension of the bounding box defines the S coordinates, and the next longest defines the T coordinates. If two or all three dimensions of the bounding box are equal, then ties should be broken by choosing the X, Y, or Z dimension in that order of preference. The value of the S coordinate ranges from 0 to 1, from one end of the bounding box to the other. The T coordinate ranges between 0 and the ratio of the second greatest dimension of the bounding box to the greatest dimension. See the figure below for an illustration of default texture coordinates for a simple box shaped IndexedFaceSet with a bounding box with X dimension twice as large as the Z dimension which is twice as large as the Y dimension:

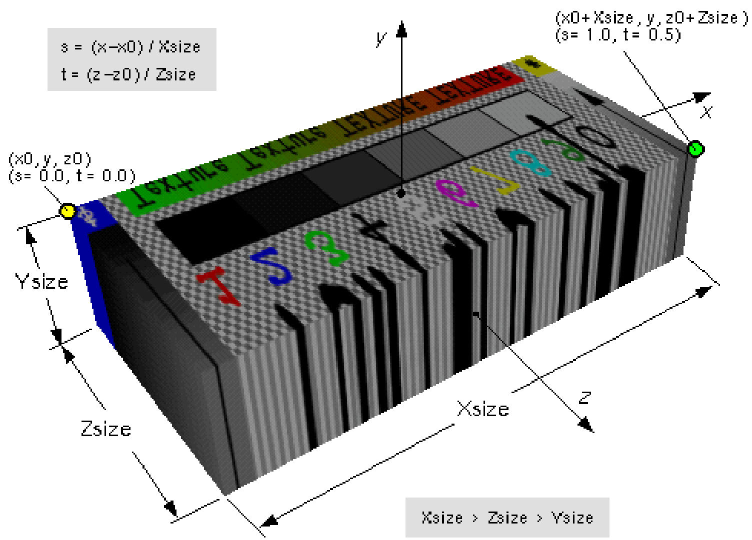


Figure 9.2

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

IndexedLineSet

IndexedLineSet {

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

}

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

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

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

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

- If `colorPerVertex` is `FALSE`:
  - If the `colorIndex` field is not empty, then one color is used for each polyline of the IndexedLineSet. There must be at least as many indices in the `colorIndex` field as there are polylines in the IndexedLineSet. If the greatest index in the `colorIndex` field is N,

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

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

Inline

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

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

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

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

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

LOD

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

LOD

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

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

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

If there are N values in the `range` field, the LOD shall have $N + 1$ nodes in its `level` field. Specifying too few levels will result in the last level being used repeatedly for the lowest levels of detail; if more levels than ranges are specified, the extra levels will be ignored. The exception to this rule is to leave the range field empty, which is a hint to the browser that it should choose a level automatically to maintain a constant display rate. Each value in the `range` field should be greater than the previous value; otherwise results are undefined.

Authors should set LOD ranges so that the transitions from one level of detail to the next are smooth. Browsers may adjust which level of detail is displayed to maintain interactive frame rates, to display an already-fetched level of detail while a higher level of detail (contained in an Inline node) is fetched, or might disregard the author-specified ranges for any other implementation-dependent reason. For best results, specify ranges only where necessary, and nest LOD nodes with and without ranges. Browsers should try to honor the hints given by authors, and authors should try to give browsers as much freedom as they can to choose levels of detail based on performance.

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

Material
MovieTexture

Material

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

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

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

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

- The `diffuseColor` reflects all VRML light sources depending on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light reflects.
- The `ambientIntensity` field specifies how much ambient light from light sources this surface should reflect. Ambient light is omni-directional and depends only on the number of light sources, not their positions with respect to the surface. Ambient color is calculated as `ambientIntensity` * `diffuseColor`.
- The `specularColor` and `shininess` determine the specular highlights–for example, the shiny spots on an apple. When the angle from the light to the surface is close to the angle from the surface to the viewer, the `specularColor` is added to the diffuse and ambient color

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

calculations. Lower shininess values produce soft glows, while higher values result in sharper, smaller highlights.

- Emissive color models "glowing" objects. This can be useful for displaying radiosity-based models (where the light energy of the room is computed explicitly), or for displaying scientific data.
- Transparency is how "clear" the object is, with 1.0 being completely transparent, and 0.0 completely opaque.

This section belong in the Conformance annex.

For rendering systems that do not support the full OpenGL lighting model, the following simpler lighting model is recommended:

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

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

Field	**Supported?**	**Suggested Action**
`ambientIntensity`	No	Ignore
`diffuseColor`	Yes	Use
`specularColor`	No	Ignore
`emissiveColor`	No	If diffuse $==$ 0.8 0.8 0.8, use emissive

`shininess`	Yes	Use
`transparency`	Yes	if $<$ 0.5 then opaque else transparent

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

Likewise, if a system supports ambient intensity but not color, the same thing should be done with the ambient color values to generate the ambient intensity. If a rendering system does not support per-object ambient values, it should set the ambient value for the entire scene at the average ambient value of all objects.

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

MovieTexture

MovieTexture {

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

}

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

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

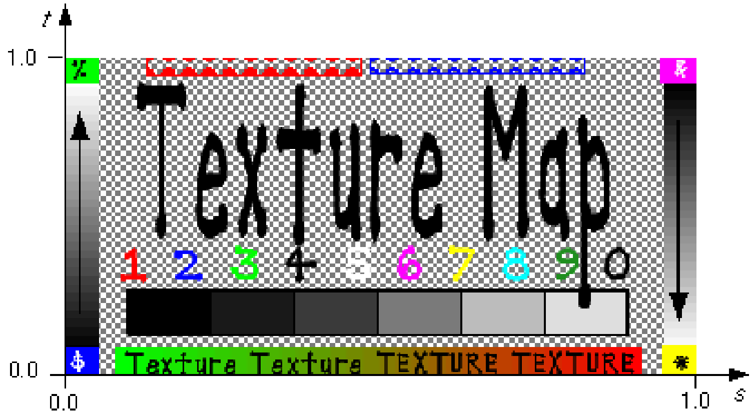


Figure 11.1

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

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

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

If a MovieTexture is inactive when the movie is first loaded, then frame 0 is shown in the texture if `speed` is non-negative, or the last frame of the movie if `speed` is negative. A MovieTexture will always display frame 0 if `speed` $= 0$. For positive values of `speed`, the frame an active

MovieTexture will display at time now corresponds to the frame at movie time (i.e., in the movie's local time system with frame 0 at time 0, at speed $= 1$):

$fmod(now - \text{startTime}, duration/\text{speed})$

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

$duration + fmod(now - \text{startTime}, duration/\text{speed})$.

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

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

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

NavigationInfo
Normal
NormalInterpolator

NavigationInfo

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

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

If a TRUE value is sent to the set_bind eventIn of a NavigationInfo, it is pushed onto the NavigationInfo stack and activated. When a NavigationInfo is bound, the browser uses the fields of the NavigationInfo to set the navigation controls of its user interface and the NavigationInfo is conceptually re-parented under the currently bound Viewpoint. All subsequent scaling changes to the current **Viewpoint**'s coordinate system automatically change aspects (see below) of the NavigationInfo values used in the browser (e.g. scale changes to any parent transformation). A FALSE value of set_bind, pops the NavigationInfo from the stack, results in an isBound

FALSE event, and pops to the next entry in the stack which must be re-parented to the current Viewpoint. See "**Concepts – Bindable Children Nodes**" for more details on the the binding stacks.

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

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

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

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

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

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

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

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

Normal

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

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

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

NormalInterpolator

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

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

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

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

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

OrientationInterpolator

OrientationInterpolator {

eventIn	SFFloat	set_fraction	
exposedField	MFFloat	key	[]
exposedField	MFRotation	keyValue	[]
eventOut	SFRotation	value_changed	

}

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

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

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

0 1 0 0 −> 0 1 0 5.0

is equivalent to the rotation between the two orientations:

$0\ 1\ 0\ 2 * \pi$ −> 0 1 0 5.0

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

PixelTexture

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

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

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

Images may be one component (greyscale), two component (greyscale plus alpha opacity), three component (full RGB color), or four-component (full RGB color plus alpha opacity). An ideal VRML implementation will use the texture image to modify the diffuse color and transparency $(= 1 - alphaopacity)$ of an object's material (specified in a **Material** node), then perform any lighting calculations using the rest of the object's material properties with the modified diffuse color to produce the final image. The texture image modifies the diffuse color and transparency depending on how many components are in the image, as follows:

1. Diffuse color is multiplied by the greyscale values in the texture image.
2. Diffuse color is multiplied by the greyscale values in the texture image; material transparency is multiplied by transparency values in texture image.

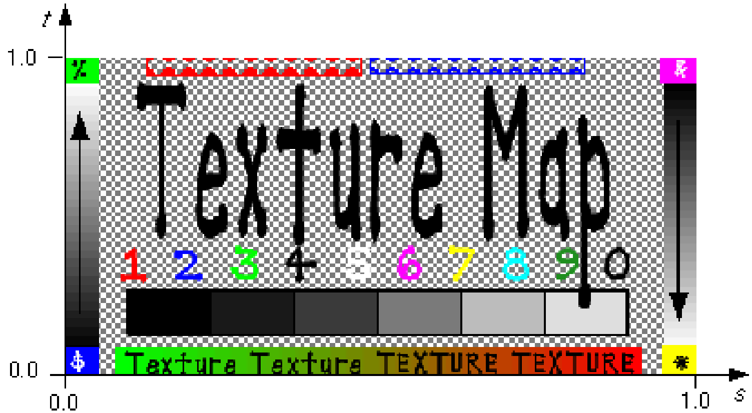


Figure 14.1

3. RGB colors in the texture image replace the material's diffuse color.
4. RGB colors in the texture image replace the material's diffuse color; transparency values in the texture image replace the material's transparency.

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

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

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

are clamped in the S direction to lie within the 0-to-1 range. The `repeatT` field is analogous to the `repeatS` field.

PlaneSensor

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

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

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

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

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

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

minPosition and maxPosition may be set to clamp translation events to a range of values as measured from the origin of the XY plane. If the X or Y component of minPosition is greater than the corresponding component of maxPosition, translation_changed events are not clamped in that dimension. If the X or Y component of minPosition is equal to the corresponding component of maxPosition, that component is constrained to the given value;

this technique provides a way to implement a line sensor that maps dragging motion into a translation in one dimension.

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

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

PointLight

PointLight {

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

}

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

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

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

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

PointSet

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

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

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

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

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

PositionInterpolator

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

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

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

ProximitySensor

ProximitySensor {

exposedField	SFVec3f	center	0 0 0
exposedField	SFVec3f	size	0 0 0
exposedField	SFBool	enabled	TRUE
eventtOut	SFBool	isActive	
eventOut	SFVec3f	position_changed	
eventOut	SFRotation	orientation_changed	
eventOut	SFTime	enterTime	
eventOut	SFTime	exitTime	

}

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

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

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

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

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

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

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

ScalarInterpolator

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

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

Script

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

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

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

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

```
eventIn type name
```

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

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

```
eventOut type name
```

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

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

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

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

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

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

Shape

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

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

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

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

Sound

```
Sound {
    exposedField  SFVec3f  direction   0 0 1
    exposedField  SFFloat  intensity   1
    exposedField  SFVec3f  location    0 0 0
    exposedField  SFFloat  maxBack     10
    exposedField  SFFloat  maxFront    10
    exposedField  SFFloat  minBack     1
    exposedField  SFFloat  minFront    1
    exposedField  SFFloat  priority    0
    exposedField  SFNode   source      NULL
    field         SFBool   spatialize  TRUE
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Sphere

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

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

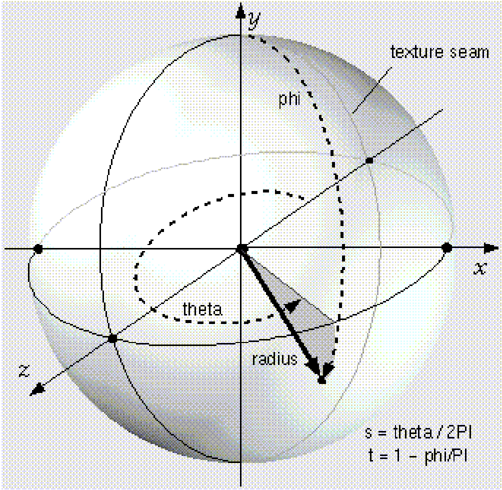


Figure 15.1

When a texture is applied to a sphere, the texture covers the entire surface, wrapping counter-clockwise from the back of the sphere. The texture has a seam at the back where the YZ plane intersects the sphere. **TextureTransform** affects the texture coordinates of the Sphere.

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

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

SphereSensor

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

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

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

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

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

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

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

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

SpotLight

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

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

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

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

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

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

where $exponent = 0.5 * log(0.5)/log(cos(beamWidth))$, intensity is the SpotLight's field value, intensity(angle) is the light intensity at an arbitrary angle from the direction vector, and angle ranges from 0.0 at central axis to cutOffAngle.

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

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

Switch

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

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

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

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

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

Text

Text {

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

}

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

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

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

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

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

Textures are applied to text as follows. The texture origin is at the origin of the first string, as determined by the justification. The texture is scaled equally in both S and T dimensions, with the font height representing 1 unit. S increases to the right, and T increases up. ISO 10646-1:1993 Character Encodings

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

Group-octet	Plane-octet	Row-octet	Cell-octet

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

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

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

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

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

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

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

All following bytes have this format: 10XXXXXX

A two byte example. The symbol for a register trade mark is "circled R registered sign" or 174 in ISO/Latin-1 (8859/1). It is encoded as 0x00AE in UCS-2 of the ISO 10646. In UTF-8 it is has the following two byte encoding 0xC2, 0xAE. See "**Concepts – Lighting Model**" for details on VRML lighting equations and how Appearance, Material and textures interact with lighting. See "**Concepts – Lighting Model**" for details on VRML lighting equations and how Appearance, Material and textures interact with lighting.

The Text node does not perform collision detection.

TextureCoordinate

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

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

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

$Location = (C - floor(C)) * N$

If the texture is not repeated:

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

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

TextureTransform

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

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

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

Given a 2-dimensional texture coordinate T and a TextureTransform node, T is transformed into point T' by a series of intermediate transformations. In matrix-transformation notation, where C (`center`), T (`translation`), R (`rotation`), and S (`scale`) are the equivalent transformation matrices,

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

Note that TextureTransforms cannot combine or accumulate.

TimeSensor

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

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

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

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

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

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

time $= now$
$f = fmod(now - \text{startTime}, \text{cycleInterval})$
if ($f == 0.0$ && $now >$ startTime) fraction_changed $= 1.0$
else

fraction_changed $= f/$cycleInterval

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

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

An active TimeSensor ignores `set_cycleInterval`, and `set_startTime` events. An active TimeSensor also ignores `set_stopTime` events for `set_stopTime` $<$ `startTime`. For example, if a `set_startTime` event is received while a TimeSensor is active, then that `set_startTime` event is ignored (the `startTime` field is not changed, and a `startTime_changed` eventOut is not generated). If an active TimeSensor receives a`set_stopTime` event that is less than now and greater than or equal to `startTime`, it behaves as if the `stopTime` requested is now and sends the final events based on now (note that `stopTime` is set as specified in the eventIn).

TouchSensor

```
TouchSensor {
    exposedField  SFBool   enabled             TRUE
    eventOut      SFVec3f  hitNormal_changed
    eventOut      SFVec3f  hitPoint_changed
    eventOut      SFVec2f  hitTexCoord_changed
    eventOut      SFBool   isActive
    eventOut      SFBool   isOver
    eventOut      SFTime   touchTime
}
```

A TouchSensor tracks the location and state of the pointing device and detects when the user points at geometry contained by the TouchSensor's parent group. This sensor can be enabled or disabled by sending it an `enabled` event with a value of `TRUE` or `FALSE`. If the TouchSensor is disabled, it does not track user input or send output events.

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

The `isOver` eventOut reflects the state of the pointing device with regard to whether it is over the TouchSensor's geometry or not. When the pointing device changes state from a position such that its bearing **does not** intersect any of the TouchSensor's geometry to one in which it does intersect geometry, an `isOver` `TRUE` event is generated. When the pointing device moves from a position such that its bearing intersects geometry to one in which it no longer intersects the geometry, or some other geometry is obstructing the TouchSensor's geometry, an `isOver` `FALSE` event is generated. These events are generated only when the pointing device has moved and changed 'over state; events are not generated if the geometry itself is animating and moving underneath the pointing device.

As the user moves the bearing over the TouchSensor's geometry, the point of intersection (if any) between the bearing and the geometry is determined. Each movement of the pointing device, while `isOver` is `TRUE`, generates `hitPoint_changed`, `hitNormal_changed`, and `hitTexCoord_changed` events. `hitPoint_changed` events contain the 3D point on the surface of the underlying geometry, given in the TouchSensor's coordinate system. `hitNormal_changed` events contain the surface normal vector at the hitPoint. `hitTexCoord_changed` events contain the texture coordinates of that surface at the hitPoint, which can be used to support the 3D equivalent of an image map.

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

The eventOut field `touchTime` is generated when all three of the following conditions are true:

- the pointing device **was** over the geometry when it was **initially activated** (`isActive` is `TRUE`),
- the pointing device **is** currently over the **geometry** (`isOver` is `TRUE`),
- and, the pointing device is **deactivated** (`isActive FALSE` event is also generated).

See "**Concepts – Pointing Device Sensors**" for more details.

Transform

Transform {

`eventIn`	MFNode	addChildren	
`eventIn`	MFNode	removeChildren	
`exposedField`	SFVec3f	center	0 0 0
`exposedField`	MFNode	children	[]
`exposedField`	SFRotation	rotation	0 0 1 0
`exposedField`	SFVec3f	scale	1 1 1
`exposedField`	SFRotation	scaleOrientation	0 0 1 0
`exposedField`	SFVec3f	translation	0 0 0
`field`	SFVec3f	bboxCenter	0 0 0
`field`	SFVec3f	bboxSize	-1 -1 -1

}

A Transform is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its parents. See also "**Concepts – Coordinate Systems and Transformations**."

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

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

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

The `translation`, `rotation`, `scale`, `scaleOrientation` and `center` fields define a geometric 3D transformation consisting of (in order) a (possibly) non-uniform scale about an arbitrary point, a rotation about an arbitrary point and axis, and a translation. The `center` field specifies a translation offset from the local coordinate system's origin, $(0, 0, 0)$. The `rotation` field specifies a rotation of the coordinate system. The `scale` field specifies a non-uniform scale of the coordinate system – `scale` values must be $>=$ 0.0. The `scaleOrientation` specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The `scaleOrientation` applies only to the scale operation. The `translation` field specifies a translation to the coordinate system.

Given a 3-dimensional point P and Transform node, P is transformed into point P' in its parent's coordinate system by a series of intermediate transformations. In matrix-transformation notation, where C (`center`), SR (`scaleOrientation`), T (`translation`), R (`rotation`), and S (`scale`) are the equivalent transformation matrices,

$P' = TxCxRxSRxSx - SRx - TCxP$ (where P) is a column vector)

The Transform node:

Transform {

center	C
rotation	R
scale	S
scaleOrientation	SR
translation	T
children	[...]

}

is equivalent to the nested sequence of:

```
Transform { translation T
 Transform { translation C
   Transform { rotation R
```

```
Transform { rotation SR
   Transform { scale S
      Transform { rotation -SR
         Transform { translation -C
         }
      }
   }
}
}
}

}
```

Viewpoint

Viewpoint

```
Viewpoint {
        eventIn       SFBool      set_bind
        exposedField  SFFloat     fieldOfView  0.785398
        exposedField  SFBool      jump         TRUE
        exposedField  SFRotation  orientation  0 0 1 0
        exposedField  SFVec3f     position     0 0 10
        field         SFString    description  ""
        eventOut      SFTime      bindTime
        eventOut      SFBool      isBound
      }
```

The Viewpoint node defines a specific location in a local coordinate system from which the user might view the scene. Viewpoints are "**Concepts – Bindable Children Nodes**" and thus there exists a Viewpoint stack in the browser in which the top-most Viewpoint on the stack is the currently active Viewpoint. If a `TRUE` value is sent to the `set_bind` eventIn of a Viewpoint, it is moved to the top of the Viewpoint stack and thus activated. When a Viewpoint is at the top of the stack, the user's view is conceptually re-parented as a child of the Viewpoint. All subsequent changes to the Viewpoint's coordinate system change the user's view (e.g. changes to any parent transformation nodes or to the Viewpoint's position or orientation fields). Sending a `set_bind` `FALSE` event removes the Viewpoint from the stack and results in `isBound` `FALSE` and `bindTime` events. If the popped Viewpoint is at the top of the viewpoint stack the user's view is re-parented to the next entry in the stack. See "concepts.htmlBindableLeafNodes" for more details on the the binding stacks. When a Viewpoint is moved to the top of the stack, the existing top of stack Viewpoint sends an `isBound` `FALSE` event and is pushed onto the stack. The Viewpoint node defines a specific location in a local coordinate system from which the user

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

might view the scene. Viewpoints are "**Concepts - Bindable Children Nodes**" and thus there exists a Viewpoint stack in the browser in which the top-most Viewpoint on the stack is the currently active Viewpoint. If a TRUE value is sent to the set_bind eventIn of a Viewpoint, it is moved to the top of the Viewpoint stack and thus activated. When a Viewpoint is at the top of the stack, the user's view is conceptually re-parented as a child of the Viewpoint. All subsequent changes to the Viewpoint's coordinate system change the user's view (e.g. changes to any parent transformation nodes or to the Viewpoint's position or orientation fields). Sending a set_bind FALSE event removes the Viewpoint from the stack and results in isBound FALSE and bindTime events. If the popped Viewpoint is at the top of the viewpoint stack the user's view is re-parented to the next entry in the stack. See "**Concepts – Bindable Nodes**" for more details on the the binding stacks. When a Viewpoint is moved to the top of the stack, the existing top of stack Viewpoint sends an isBound FALSE event and is pushed onto the stack.

Viewpoints have the additional requirement from other binding nodes in that they store the relative transformation from the user view to the current Viewpoint when they are moved to the top of stack. This is needed by the jump field, described below.

An author can automatically move the user's view through the world by binding the user to a Viewpoint and then animating either the Viewpoint or the transformations above it. Browsers shall allow the user view to be navigated relative to the coordinate system defined by the Viewpoint (and the transformations above it), even if the Viewpoint or its parent transformations are being animated.

The bindTime eventOut sends the time at which the Viewpoint is bound or unbound. This can happen during loading, when aset_bind event is sent to the Viewpoint, or when the browser binds to the Viewpoint via its user interface (see below).

The position and orientation fields of the Viewpoint node specify relative locations in the local coordinate system. Position is relative to the coordinate system's origin $(0, 0, 0)$, while orientation specifies a rotation relative to the default orientation; the default orientation has the user looking down the $-Z$ axis with $+X$ to the right and $+Y$ straight up. Viewpoints are affected by the transformation hierarchy.

Navigation types (see **NavigationInfo**) that require a definition of a down vector (e.g. terrain following) shall use the negative Y-axis of the coordinate system of the currently bound Viewpoint. Likewise navigation types (see **NavigationInfo**) that require a definition of an up vector shall use the positive Y-axis of the coordinate system of the currently bound Viewpoint. Note that the `orientation` field of the Viewpoint does not affect the definition of the down or up vectors. This allows the author to separate the viewing direction from the gravity direction.

The `jump` field specifies whether the user's view 'jumps' (or animates) to the position and orientation of a bound Viewpoint. Regardless of the value of `jump` at bind time, the relative viewing transformation between the user's view and the current Viewpoint shall be stored with the current Viewpoint for later use when un-jumping. The following is a re-write of the general bind stack rules described in "**Concepts – Bindable Child Nodes, Bind Stack Behavior**" with additional rules regarding Viewpoints (in **bold**):

1. During read:
   - the first encountered Viewpoint is bound by pushing it to the top of the Viewpoint stack,
     - nodes contained within **Inlines** are not candidates for the first encountered Viewpoint,
     - the first node within a prototype is a valid candidate for the first encountered Viewpoint;
   - the first encountered Viewpoint sends an `isBound` `TRUE` event.
2. When a `set_bind` `TRUE` eventIn is received by a Viewpoint:
   - if it is **not** on the top of the stack:
     - the relative transformation from the current top of stack Viewpoint to the user's view is stored with the current top of stack Viewpoint,
     - the current top of stack node sends an `isBound` eventOut `FALSE`,

   - the new node is **moved** to the top of the stack and becomes the currently bound Viewpoint,
   - the new Viewpoint (top of stack) sends an isBound TRUE eventOut,
   - if jump is TRUE for the new Viewpoint, then the user's view is 'jumped' (or animated) to match the values in the position and orientation fields of the new Viewpoint;
 - else if the node is already at the top of the stack, then this event has no affect.
3. When a set_bind FALSE eventIn is received by a Viewpoint:
   - it is removed from the stack,
   - if it is on the top of the stack:
     - it sends an isBound eventOut FALSE,
     - the next node in the stack becomes the currently bound Viewpoint (i.e. pop) and issues an isBound TRUE eventOut,
     - if its jump is TRUE the user's view is 'jumped' (or animated) to the position and orientation of the next Viewpoint in the stack **with** the stored relative transformation for with this next Viewpoint applied,
4. If a set_bind FALSE eventIn is received by a node not in the stack, the event is ignored and isBound events are not sent.
5. When a node replaces another node at the top of the stack, the isBound TRUE and FALSE eventOuts from the two nodes are sent simultaneously (i.e. identical timestamps).
6. If a bound node is deleted then it behaves as if it received a set_bind FALSE event (see #3).

Note that the jump field may change after a Viewpoint is bound - the rules described above still apply. If jump was TRUE when the Viewpoint is bound, but changed to FALSE before the set_bind FALSE is sent, then the Viewpoint does not un-jump during unbind. If jump was FALSE when the Viewpoint is bound, but changed to TRUE before the set_bind FALSE is sent, then the Viewpoint does perform the un-jump during unbind.

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

The `fieldOfView` field specifies a preferred field of view from this viewpoint, in radians. A small field of view roughly corresponds to a telephoto lens; a large field of view roughly corresponds to a wide-angle lens. The field of view should be greater than zero and smaller than PI; the default value corresponds to a 45 degree field of view. The value of `fieldOfView` represents the maximum viewing angle in any direction axis of the view. For example, a browser with a rectangular viewing projection shall use an angle of `fieldOfView` for the larger direction (depending on aspect ratio) and `fieldOfView` times`aspect ratio` in the smaller direction. If the aspect ratio is 2x1 (i.e. horizontal twice the vertical) and the fieldOfView is 1.0, then the horizontal viewing angle would be 1.0 and the vertical viewing angle would be 0.5. `fieldOfView` is a hint to the browser and may be ignored.

The `description` field identifies Viewpoints that are recommended to be publicly accessible through the browser's user interface (e.g. Viewpoints menu). The string in the `description` field should be displayed if this functionality is implemented. If `description` is empty, then the Viewpoint should not appear in any public user interface. It is recommended that the browser bind and move to a Viewpoint when its `description` is selected, either animating to the new position or jumping directly there. Once the new position is reached both the `isBound` and `bindTime` eventOuts are sent.

The URL syntax "…/scene.wrl#ViewpointName" specifies the user's initial view when entering "scene.wrl" to be the first Viewpoint in file "scene.wrl" that appears as "DEF ViewpointName Viewpoint { … }" – this overrides the first Viewpoint in the file as the initial user view and receives a `set_bind` `TRUE` message. If the Viewpoint "ViewpointName" is not found, then assume that no Viewpoint was specified and use the first Viewpoint in the file. The URL syntax "#ViewpointName" specifies a view within the existing file. If this is loaded, then receives a `set_bind` `TRUE` message.

If a Viewpoint is bound (`set_bind`) and is the child of an LOD, Switch, or any node or prototype that disables its children, then the result is undefined. If a Viewpoint is bound that results in collision with geometry, then the browser performs its self-defined navigation adjustments as if the user navigated to this point (see **Collision**).

VisibilitySensor

go back
contents
a n b o c p d q e r f s g t h u i v j w k x l y m z
find
print
close

VisibilitySensor

```
VisibilitySensor {
    exposedField  SFVec3f  center     0 0 0
    exposedField  SFBool   enabled    TRUE
    exposedField  SFVec3f  size       0 0 0
    eventOut      SFTime   enterTime
    eventOut      SFTime   exitTime
    eventOut      SFBool   isActive
}
```

The VisibilitySensor detects visibility changes of a rectangular box as the user navigates the world. VisibilitySensor is typically used to detect when the user can see a specific object or region in the scene, and to activate or deactivate some behavior or animation in order to attract the user or improve performance.

The `enabled` field enables and disables the VisibilitySensor. If `enabled` is set to FALSE, the VisibilitySensor does not send output events. If `enabled` is TRUE, then the VisibilitySensor detects changes to the visibility status of the box specified and sends events through the `isActive` eventOut. A TRUE event is output to `isActive` when any portion of the box impacts the rendered view, and a FALSE event is sent when the box has no effect on the view. Browsers shall guarantee that if `isActive` is FALSE that the box has absolutely no effect on the rendered view – browsers may error liberally when `isActive` is TRUE (e.g. maybe it does affect the rendering).

The exposed fields `center` and `size` specify the object space location of the box center and the extents of the box (i.e. width, height, and depth). The VisibilitySensor's box is effected by hierarchical transformations of its parents.

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

Each VisibilitySensor behaves independently of all other VisibilitySensors – every enabled VisibilitySensor that is affected by the user's movement receives and sends events, possibly resulting in multiple VisibilitySensors receiving and sending events simultaneously. Unlike TouchSensors, there is no notion of a Visibility Sensor lower in the scene graph "grabbing" events. Instanced (DEF/USE) VisibilitySensors use the **union** of all the boxes defined by their instances to check for enter and exit – an instanced VisibilitySensor will detect enter, motion, and exit for all instances of the box and send output events appropriately.

go back
contents
a n
b o
c p
d q
e r
f s
g t
h u
i v
j w
k x
l y
m z
find
print
close

WorldInfo

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

The WorldInfo node contains information about the world. This node has no effect on the visual appearance or behavior of the world – it is strictly for documentation purposes. The `title` field is intended to store the name or title of the world so that browsers can present this to the user – for instance, in their window border. Any other information about the world can be stored in the `info` field – for instance, the scene author, copyright information, and public domain information.