Chapter 19: Hear, Hear!

Black Silence

In the early days of VR, we joked that the universe of unpopulated cyberspace - almost all of it - consisted of nothing but virtual light-years of "black silence". We imagined it to be a relaxing, quiet void, far removed from the bustle and hubbub of inhabited cyberspace, filled with the sounds and sights of a humanity using it to talk, to teach, and to trade.

It amazes me how much of cyberspace still looks like black silence, as if the VRML revolution never happened. Most worlds consist of a few objects, against a black background, looking as though they'd been burped up by the void. It looks too simple, far too hard-edged to be real, and that contributes - in part - to a perception of cyberspace as unsuitable for real-world applications.

Yet, with the clever use of sounds and backgrounds, almost any environment will begin to look and sound emotionally compelling. Sound is a bridge between the mind and the heart; while the eye remains detached and rational, the ear fills space with emotional meaning. And the background is the most emotional landscape for the eye; as the eye traces into the distance, stares into the sunset or a dewy-colored dawn, the eye senses the passage of time - an emotional event.

Background Check

Quite a bit of information can be conveyed by a well-constructed background; it sets the stage for foreground activities, and can easily connote day/night differences. In VRML, the Background node controls the background coloring for the virtual world:

```
Background {            # definition
      groundAngle []          # MFFloat, exposedField
      groundColor []          # MFColor, exposedField
      backUrl []        # MFString, exposedField
      bottomUrl []            # MFString, exposedField
      frontUrl []       # MFString, exposedField
      leftUrl []        # MFString, exposedField
      rightUrl []       # MFString, exposedField
      topUrl []               # MFString, exposedField
      skyAngle []       # MFFloat, exposedField
      skyColor []       # MFColor, exposedField
}
```

The basic operation of the Background node is to set the "space" behind all visible objects. One field, skyColor, has a default MFColor value of [0 0 0], which creates the default black background you've seen in all of the previous examples. For example, if you wanted to create a VRML world with no objects, but a uniform light blue - for the sky - you might do the following:

```
#VRML V2.0 utf8
# This is the first example on sound
Background {
      skyColor [ 0.7 1 1 ] # create light blue background
}
```

Which gives you a world with no objects, but a uniformly blue background color.

You can define multiple values in the skyColor field, to create the concentric rings of color associated with a real sky, but you must then use the skyAngle field to define the angle of visibility associated with those colors. The skyAngle values, given in radians, express at what angle downward from the north pole (that is, directly up) any successive colors begin. The browser will do its best to blend the colors. Let's say that we want to create a Background which is deep blue at the peak of the sky, and then rather washed out as it approaches the horizon. We'd need two values in the skyColor field, and a single value in the skyAngle field, indicating the angle where we switch from the first value in the skyColor field to the second. It might look like this:

```
#VRML V2.0 utf8
# This is the second example on sound
Background {
      skyColor [0.8 1 1, 0.5 0.9 0.9 ] # two colors
      skyAngle [ 0.785 ] # change at 45 degrees
}
```

Now, as you look up toward the peak of the sky, you see the sky grow brighter and brighter.

Sailor's Delight

It's important to remember that there should always be one more value in the skyColor field than in the skyAngle field; for two values in skyColor, one value in skyAngle will suffice; but with five values in skyColor - as in our next example - you'll need to have four values in skyAngle.

For our third example, let's create something that looks a bit like a sunset:

```
#VRML V2.0 utf8
# This is the third example on sound
Background {
      skyColor [0.1 0.1 0.45  # dim blue
            0.1 0.35 0.1 # green
            0.35 0.35 0.1 # yellow
            0.35 0.25 0.05 # orange
            0.35 0 0 ] # red, five colors
      skyAngle [ 0.785, 1.04, 1.30, 1.40 ] # 45, 60, 75, 80
degrees
}
```

We define skyColor values for dim blue, green, yellow, orange and red - just as we'd see them in a sunset - and set four skyAngle values for them. Does it look realistic? Take a look.

Beginning to look rather sunset-like, isn't it?

Beneath our Feet

Despite the improving realism of the previous example, something is missing. The *ground plane* - the infinite surface where the ground meets the sky - hasn't been added to the scene. The field groundColor allows us to define a ground plane. Let's add a brown ground plane to this scene, and see if it looks even better:

```
      #VRML V2.0 utf8
      # This is the fourth example on sound
      Background {
            skyColor [0.1 0.1 0.45  # dim blue
                  0.1 0.35 0.1 # green
                  0.35 0.35 0.1 # yellow
                  0.35 0.25 0.05 # orange
                  0.35 0 0 ] # red, five colors
            skyAngle [ 0.785, 1.04, 1.30, 1.40 ] # 45, 60, 75, 80
degrees
            groundColor [ 0.15 0.15 0 ] # brown ground plane
      }
```

Now we have a line between the ground and the sky which highlights our sunset sky even better.

The groundColor and groundAngle fields work in much the same way as the skyColor and skyAngle fields; it is possible to define a number of different colors for the ground as it recedes into the distance. The first value in groundColor is the value when looking straight down (south pole), and all successive colors are placed at an angle relative to the south pole, up to the horizon. Let's create three shades of brown, and use these to fade the horizon into darkness:

```
      #VRML V2.0 utf8
      # This is the fifth example on sound
      Background {
            skyColor [0.1 0.1 0.45  # dim blue
                  0.1 0.35 0.1 # green
                  0.35 0.35 0.1 # yellow
                  0.35 0.25 0.05 # orange
                  0.35 0 0 ] # red, five colors
            skyAngle [ 0.785, 1.04, 1.30, 1.40 ] # 45, 60, 75, 80
degrees
            groundColor [ 0.20 0.20 0,  # brown
                        0.12 0.12 0, # darker
                        0.05 0.05 0 ] # darkest
            groundAngle [ 1.04, 1.40 ] # 75 and 80 degrees
      }
```

Now the ground fades into darkness before the sunset.

Caught Between Day and Night

Each of these values are exposedField values, so they can be changed from within scripts. Here's another example using a ColorInterpolator to change the skyColor from the ruddy red of dawn to the bright blue of day and back again, all in 60 seconds:

```
#VRML V2.0 utf8
# This is the sixth example on sound
DEF DAYLIGHT Background {
      skyColor [ 0.1 0 0 ] # starting color
      groundColor [ 0.20 0.20 0,  # brown
                        0.12 0.12 0, # darker
                        0.05 0.05 0 ] # darkest
      groundAngle [ 1.04, 1.40 ] # 75 and 80 degrees
}
# This is an auto-starting timer with 60 second cycle
DEF TIMER TimeSensor {
      loop TRUE
      cycleInterval 60
      startTime 1
      stopTime 0
}
# This color interpolator changes the skyColor
DEF DAYMAKER ColorInterpolator {
      key [ 0, 0.5, 1 ] # start, mid-way, end
      keyValue [ 0.1 0 0, 0.5 0.9 0.9, 0.1 0 0 ]
}
# We need a script to convert between the SFString
# in the ColorInterpolator and the MFString in Background
DEF CONVERTER Script {
      eventIn SFColor inColor
      eventOut MFColor outColor
      field MFColor localColor [ 0 0 0 ] # local MFColor value
      url [ "javascript:
            function inColor(ic) {
                  localColor[0] = ic;      // convert to MFString
                  outColor = localColor; // send it out
            } "
      ]
}
# Route TimeSensor to the ColorInterpolator
ROUTE TIMER.fraction_changed TO DAYMAKER.set_fraction
# ROUTE ColorInterpolator to the Script
ROUTE DAYMAKER.value_changed TO CONVERTER.inColor
# ROUTE Script to the Background
ROUTE CONVERTER.outColor TO DAYLIGHT.set_skyColor
```

We do something in this example we've never had to do before. We have a problem of type conversion; meaning that the ColorInterpolator sends out a steady stream of SFColor values, while the Background node exposedField skyColor wants to receive MFColor values. These data types are not the same, so we created the Script node to act as a converter between the SFColor and MFColor values. All the Script does is read the

incoming SFColor value, place it into an MFColor array as the first element in the array, then it send the array over its eventOut. This is a very common use of a Script in VRML - serving as glue between incompatible field data types.

Now we have a Background that slowly cycles between light and dark.

For the World is Hollow and I Have Wallpapered the Sky

For some applications, you'll need more than a set of colors blending in the distance; you'll need an image that can maintain the illusion of a far-off set of mountains, or a star field in interstellar space. This photographic background is known as a *panorama*, and the Background node has six fields that let you define a panoramic background. The backUrl, bottomUrl, frontUrl, leftUrl, rightUrl, and topUrl fields point to image URLs that create the six faces of the panorama - which you can think of as a gigantic cube that surrounds the entire world. Each of the sides of the cube faces inward, toward the camera.

In the next example, we'll use the panoramic fields of the Background node in conjunction with some descriptive JPEG images, to create a self-explanatory visualization of how the Background node works:

```
#VRML V2.0 utf8
# This is the seventh example on sound
Background {
      backUrl [ "back.jpg" ] # goes behind camera
      bottomUrl [ "bottom.jpg" ] # goes beneath camera
      frontUrl [ "front.jpg" ] # goes before camera
      leftUrl [ "left.jpg" ] # goes to left of camera
      rightUrl [ "right.jpg" ] # goes to right of camera
      topUrl [ "top.jpg" ] # goes above camera
}
```

What we see - as we spin about and examine the background - is our view from within the panoramic cube, with the images facing us.

Each of the images remain forever in the distance; you can walk toward them forever, but they'll never get any closer. You can make these images partially transparent, and use them in conjunction with skyColor, skyAngle, groundColor and groundAngle to create effects with both panoramas and shaded backgrounds. In our last example on backgrounds, we'll have panoramic text floating within our sunset scene:

```
#VRML V2.0 utf8
# This is the eighth example on sound
Background {
      skyColor [0.1 0.1 0.45  # dim blue
            0.1 0.35 0.1 # green
            0.35 0.35 0.1 # yellow
            0.35 0.25 0.05 # orange
            0.35 0 0 ] # red, five colors
```

```
      skyAngle [ 0.785, 1.04, 1.30, 1.40 ] # 45, 60, 75, 80 degrees
      groundColor [ 0.20 0.20 0,  # brown
                    0.12 0.12 0, # darker
                    0.05 0.05 0 ] # darkest
      groundAngle [ 1.30, 1.40 ] # 75 and 80 degrees
      backUrl [ "back_text.gif" ] # goes behind camera
      bottomUrl [ "bottom_text.gif" ] # goes beneath camera
      frontUrl [ "front_text.gif" ] # goes before camera
      leftUrl [ "left_text.gif" ] # goes to left of camera
      rightUrl [ "right_text.gif" ] # goes to right of camera
      topUrl [ "top_text.gif" ] # goes above camera
}
```

Each of these GIF images use transparency; the letters appear to "float" in the background.

In much the same way, you could use a combination of image panoramas (perhaps of mountains or lakes or beaches) together with background colors to create just the visual backdrop for your worlds.

Sounding Off

Backgrounds create less than half of the emotional ambience within a virtual world; the largest portion comes from sound. Sound in VRML is *spatialized*, also known as 3D audio. This means that sound has a placement in the world, just like a light source, and it also means that the quality of the sound varies as you move closer toward it, away from it, or move around it. The Sound node in VRML has plenty of fields to let you tune the qualities of a sound very specifically; it literally lets you build your own speaker, with very unique qualities. Here's the definition:

```
      Sound {           # definition
            direction               # SFVec3f, exposedField
            intensity               # SFFloat, exposedField
            location                # SFFloat, exposedField
            maxBack                 # SFFloat, exposedField
            maxFront                # SFFloat, exposedField
            minBack                 # SFFloat, exposedField
            minFront                # SFFloat, exposedField
            priority                # SFFloat, exposedField
            source                  # SFNode, exposedField
            spatialize        # SFBool
      }
```

Like a SpotLight, every sound has fields for direction (which should be a normalized vector) and location, values relative to the local coordinate system. The default direction value of 0 0 1, points the sound source out along the z-axis. The intensity field acts as the "volume control" for the sound source, and has a range from 0 (silent) to 1.0 (maximum volume).

The four fields maxBack, maxFront, minBack and minFront define an ellipsoid around the sound source which defines the source's audible radius. In front of the source, the

sound first becomes audible when the camera is closer than maxFront, and grows to full intensity as the source is approached, until the camera is closer than minFront, at which time the source is emitting sound at full intensity. From the rear of the source, the sound becomes audible when the camera is closer than maxBack, and grows to full intensity as the source is approached, until the camera is closer than minBack, at which time the source is emitting sound at full intensity. These four fields define a sound source which is louder in one direction than its reverse, which is typically how audio speakers in the real world would behave.

Because a browser can have several active sound sources playing through it at one time, and because playing and mixing several sounds simultaneously can be very taxing on a computer, the priority field allows you to designate the relative importance of sounds. The value can range from 0 (unimportant) to 1.0 (urgent) - so it's possible to create a loudspeaker, for example, which can drown out all of the other audio chatter within a virtual world. You can guarantee that your messages will get heard. The priority field has a default value of 0; unless you establish a different priority value for your sounds, all of them will have the same - very low - priority.

While the browser can present sounds in captivating spatialized audio, it isn't always possible - or necessary - to do so. The spatialize field allows you to instruct the browser to render a sound in 3D, to present it as a regular stereo source, or - as we'll see in a bit - to present it as a pervasive *ambient* sound. The field has a default value of TRUE, so sounds will be presented spatially unless you state otherwise.

The source field allows you to define the audio to be used as the content of the sound; only two nodes are permitted in this field; the MovieTexture node allows you to use the soundtrack associated with an MPEG movie as sound input - and this means that you can have an MPEG movie presented both visually (as a texture map) and aurally.

The most common node supplied in the source field is the AudioClip node. Here's the definition of the node:

```
AudioClip {       # definition
      description       # SFString, exposedField
      loop                    # SFBool, exposedField
      pitch             # SFFloat, exposedField
      startTime               # SFTime, exposedField
      stopTime                # SFTime, exposedField
      url                     # MFString, exposedField
}
```

The url field of the AudioClip node points to a valid sound file; the only sound file type which must be supported within a VRML browser is the WAVE file type - commonplace on all machines running Microsoft operating systems, though WAVE players are common for other platforms as well. The WAVE file is a "sample", that is, a digitized version of a real-world sound. Like other url fields, this field is an MFString, with several possible values, pointing to a first choice, then, if unavailable, a second choice, and a third, and so on.

The description field provides a definition of the sound source; it isn't used by the browser, but you can access it through the Script node, and it makes a very good place to put copyright information, if your sound is under copyright.

The loop, startTime and stopTime fields behave as they do in both the TimeSensor and MovieImage nodes; if loop is TRUE and startTime is greater than or equal than stopTime, the sound will begin playing as soon as the world is loaded. If loop is FALSE, the sound will play once, and only when startTime is greater than or equal to stopTime.

The pitch field controls the speed of the playback of the sound. The default pitch value of 1.0 means that the sampled sound is played back at the same speed as it was recorded; a value of 0.5 plays it at half-speed, while a value of 2.0 plays it at double-speed. When you change the pitch of a sample, you also change its duration, so a fifteen-second sample played at a pitch value of 3.0 will take only five seconds to play.

For our first example, let's place a sound at the center of the space, but let's create the frame of a speaker around it, using a Cone and a Cylinder, so that it actually looks quite real. We'll set the outside range of both maxBack and maxFront to 100 units, and minBack and minFront to 10 units. We'll need to set up the AudioClip node so that it loads and begins playing immediately and continuously, and we'll give it a nice 30-second sample from a talk I gave in San Francisco a few years ago. Here's how all of that might look:

```
#VRML V2.0 utf8
# This is the ninth example on sound
# Give it a background so it looks reasonable
Background {
      skyColor [ 0.5 0.8 0.9 ] # bluish
}
# Create the Cone/Cylinder pair for the speaker
Transform {
      children [
            Shape {
                  appearance Appearance {
                        material DEF SPEAKER Material {
                              diffuseColor 0.25 0.1 0.1
                              specularColor 0.8 0.7 0.8
                              shininess 0.9
                        }
                  }
                  geometry Cone {}
            }
            # The Cylinder inside a transform as well
            Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material USE SPEAKER # reuse
                              }
                              geometry Cylinder {
radius 0.5
                                    height 1.0
```

```
                              }
                        }
                  ]
                  translation 0 0.45 0
            }
      ]
      rotation 1 0 0 -1.78 # speaker rotated to face you
}
# Now let's define the Sound node and AudioClip
Sound {
      intensity 1.0           # full loudness
      maxBack 100       # far away
      maxFront 100            # same to front
      minBack     10          # close up
      minFront  10            # and close in front
      spatialize TRUE   # real 3D sound
      source AudioClip {      # Here's the clip itself
            loop TRUE         # load and play automatically
            startTime 1 # automatic play
            stopTime 0  # automatic play
            url [ "milnet.wav" ] # point to sample
      }
}
```

That's all we'll need for our first example. When we load it into the browser, we can see that we're facing the speaker head-on.

In addition, we can hear the sound that begins loading and playing through the speakers. The Sound will continue to play continuously while this world is loaded, and while we're closer to the speaker than 100 units. If you walk toward the left, the speaker moves toward the right, and the sound comes out of the right speaker.

If you walk to the right, you can hear the speakers pan the sound toward the center, then move off again, toward the left speaker.

If you walk away, you can hear the sound attenuate, until - when you're 100 units away, the sound disappears completely.

It doesn't matter the direction you walk; as long as you're 100 units away, the sound attenuates to silence. Between 100 units and 10 units, it gets steadily louder, until - when you're very close to the speaker, you can hear it at full volume.

If we change the value of the pitch field to 2.0, we can double the pitch of the sound:

```
#VRML V2.0 utf8
# This is the tenth example on sound
# Give it a background so it looks reasonable
Background {
      skyColor [ 0.5 0.7 0.8 ] # bluish
}
# Create the Cone/Cylinder pair for the speaker
Transform {
      children [
```

```
            Shape {
                  appearance Appearance {
                        material DEF SPEAKER Material {
                              diffuseColor 0.25 0.1 0.1
                              specularColor 0.8 0.7 0.8
                              shininess 0.9
                        }
                  }
                  geometry Cone {}
            }
            # The Cylinder inside a transform as well
            Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material USE SPEAKER # reuse
                              }
                              geometry Cylinder {
radius 0.5
                                    height 1.0
                              }
                        }
                  ]
                  translation 0 0.45 0
            }
      ]
      rotation 1 0 0 -1.78 # speaker rotated to face you
}
# Now let's define the Sound node and AudioClip
Sound {
      intensity 1.0           # full loudness
      maxBack 100       # far away
      maxFront 100            # same to front
      minBack     10          # close up
      minFront  10            # and close in front
      spatialize TRUE   # real 3D sound
      source AudioClip {      # Here's the clip itself
            pitch 2.0         # double the pitch
            loop TRUE         # load and play automatically
            startTime 1 # automatic play
            stopTime 0  # automatic play
            url [ "milnet.wav" ] # point to sample
      }
}
```

This does a fine job of making me sound like *Alvin and the Chipmunks*, but does little to make me intelligible. On the other hand, if we set the value of pitch to 0.5, it's so slow that it's almost painful. Try example **19_11.wrl** and see…

Mood Music

In addition to having a sound with a defined range of audibility, it's also possible to create *ambient sound* within a VRML world. Ambient sound is a continuous sound that pervades an entire space and never attenuates anywhere within it - like elevator music or

that stuff they’re always playing in the mall. To make a Sound node deliver ambient sound, you must set the maxBack, maxFront, minBack and minFront fields to the same (preferably large) value, you must set spatialize to FALSE, and provide as a source an AudioClip node with loop set to TRUE. When all that happens, the sound loads and plays uniformly throughout the space. Here’s the ninth example, redone ambient style:

```
#VRML V2.0 utf8
# This is the twelfth example on sound
# Give it a background so it looks reasonable
Background {
      skyColor [ 0.4 0.6 0.7 ] # bluish
}
# Create the Cone/Cylinder pair for the speaker
Transform {
      children [
            Shape {
                  appearance Appearance {
                        material DEF SPEAKER Material {
                              diffuseColor 0.25 0.1 0.1
                              specularColor 0.8 0.7 0.8
                              shininess 0.9
                        }
                  }
                  geometry Cone {}
            }
            # The Cylinder inside a transform as well
            Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material USE SPEAKER # reuse
                              }
                              geometry Cylinder {
radius 0.5
                                    height 1.0
                              }
                        }
                  ]
                  translation 0 0.45 0
            }
      ]
      rotation 1 0 0 -1.78 # speaker rotated to face you
}
# Now let’s define the Sound node and AudioClip
Sound {
      intensity 1.0           # full loudness
      maxBack 100       # all values same
      maxFront 100            # to create
      minBack     100         # ambience in the
      minFront  100           # space
      spatialize FALSE  # must be false for ambient
      source AudioClip {      # Here’s the clip itself
            loop TRUE         # load and play automatically
            startTime 1 # automatic play
            stopTime 0  # automatic play
            url [ "milnet.wav" ] # point to sample
```

```
	}
}
```

Now when we load the world, no matter where we go - even beyond the visibility of the speaker, we can still hear the Sound playing, at full volume.

What's the Score?

While all VRML browsers must be able to play WAVE files, many browsers can also play MIDI (Musical Instrument Digital Interface) files. MIDI presents an instrument "score", rather like a musical score for. a symphony, rather than a prerecorded sound, and the score uses the built-in musical synthesis capability of your computer to execute the score, and play a tune.

There are lots of freely available MIDI files on the World Wide Web - and there are probably a few on your computer already, part of the installation of your operating system or some other programs you've used over the years - MIDI files are often used to create the soundtrack in games such as DOOM.

In this example - which **must** be run under Cosmo Player - we adapt our ambient sound example to use a MIDI file which plays continuously, in the background. We'll also modify the value in the intensity field of the Sound node, so that it doesn't become too annoying:

```
#VRML V2.0 utf8
# This is the thirteenth example on sound
# Give it a background so it looks reasonable
Background {
	skyColor [ 0.4 0.6 0.7 ] # bluish
}
# Create the Cone/Cylinder pair for the speaker
Transform {
	children [
		Shape {
			appearance Appearance {
				material DEF SPEAKER Material {
					diffuseColor 0.25 0.1 0.1
					specularColor 0.8 0.7 0.8
					shininess 0.9
				}
			}
			geometry Cone {}
		}
		# The Cylinder inside a transform as well
		Transform {
			children [
				Shape {
					appearance Appearance {
						material USE SPEAKER # reuse
					}
					geometry Cylinder {
radius 0.5
```

```
                                    height 1.0
                              }
                        }
                  ]
                  translation 0 0.45 0
            }
      ]
      rotation 1 0 0 -1.78 # speaker rotated to face you
}
# Now let's define the Sound node and AudioClip
Sound {
      intensity 0.3           # nice background volume
      maxBack 100       # all values same
      maxFront 100            # to create
      minBack     100         # ambience in the
      minFront  100           # space
      spatialize FALSE  # must be false for ambient
      source AudioClip {      # Here's the clip itself
            loop TRUE         # load and play automatically
            startTime 1 # automatic play
            stopTime 0  # automatic play
            url [ "bach.mid" ] # point to MIDI file
      }
}
```

We've got a MIDI file of Bach's *Toccata and Fugue in D Minor* playing as we enter the world - classical music makes for a classy presentation.

The most amazing thing is that this piece, which is seven and a half minutes long, takes up only 24K. One of the most gratifying things about MIDI files is that they can do a lot in very little space; they average a hundred to a *thousand times* more compact than WAVE files with the same information inside.

Perimeter Alarms

We could use a Script node or a TouchSensor to turn our Sound node on, but until now we've set it up so that it begins to play automatically. But another sensor node, the ProximitySensor, can be used with the Sound node to create some very interesting effects. The ProximitySensor sends events when the users's camera enters a particular volume of space, and, for as long as the user remains in that space, the sensor sends events, relaying the user's current position. The node looks like this:

```
      ProximitySensor {       # definition
            enabled                 # SFString, exposedField
            center                  # SFVec3f, exposedField
            size                    # SFVec3f, exposedField
      }
```

The enabled field, which defaults to TRUE, turns the sensor on and off. The ProxmitySensor has its center at the center of the local coordinate system, unless the center field adds a translation to that. The size of the ProximitySensor - which defines a

six-sided, rectangular shape - is given in the size field of the node. If no size value is given, the ProximitySensor covers no area, as it defaults to a size of 0 0 0.

Once a user has entered a ProximitySensor, it transmits a number of eventOut messages. The eventOut isActive transmits an SFBool of TRUE when the user enters the sensor, and FALSE when the user exits. Another eventOut, position_changed, transmits a constant stream of SFVec3f messages; every time the user moves, the new position - relative to the local coordinate system - is emitted as an event. When the user enters the sensor, eventOut enterTime sends the SFTime value at entry; when the user exits, exitTime sends another SFTime value.

The ProximitySensor can be used to do things like open doors automatically, or turn lights on, or - better yet - to start sounds. In fact, it's so easy to ROUTE the enterTime and exitTime eventOut messages into the startTime and stopTime fields of the Sound node, you might think they were designed to for each other. (They were.)

In our next example, we'll return to the example nine, but now we'll create a ProxmitySensor which turns the Sound on when we're closer than 50 units to the speaker; it'll also be turned off when we're more than 50 units from it. Note that the ProxmitySensor is in the same Transform as the Cone which defines the speaker; that places the sensor in the local coordinate system of the Cone:

```
#VRML V2.0 utf8
# This is the fourteenth example on sound
# Give it a background so it looks reasonable
Background {
      skyColor [ 0.5 0.8 0.9 ] # bluish
}
# Create the Cone/Cylinder pair for the speaker
Transform {
      children [
            # Create the ProximitySensor in this Transform
            DEF SENSOR ProximitySensor {
                  size 50 50 50 # 50 units on a side
            }
            Shape {
                  appearance Appearance {
                        material DEF SPEAKER Material {
                              diffuseColor 0.25 0.1 0.1
                              specularColor 0.8 0.7 0.8
                              shininess 0.9
                        }
                  }
                  geometry Cone {}
            }
            # The Cylinder inside a transform as well
            Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material USE SPEAKER # reuse
                              }
                              geometry Cylinder {
```

```
radius 0.5
                                        height 1.0
                                  }
                            }
                      ]
                      translation 0 0.45 0
                }
      ]
      rotation 1 0 0 -1.78 # speaker rotated to face you
}
# Now let's define the Sound node and AudioClip
Sound {
      intensity 1.0           # full loudness
      maxBack 100       # far away
      maxFront 100            # same to front
      minBack     10          # close up
      minFront  10            # and close in front
      spatialize TRUE   # real 3D sound
      source DEF PLAYER AudioClip { # Here's the clip itself
            loop TRUE         # loop when playing
            startTime 0 # must start play
            stopTime 1  # must start play
            url [ "milnet.wav" ] # point to sample
      }
}
# ROUTE from the ProximitySensor into the AudioClip
# To turn things on when we're close
ROUTE SENSOR.enterTime TO PLAYER.set_startTime
# ROUTE from the ProximitySensor into the AudioClip
# To turn things off when we're far
ROUTE SENSOR.exitTime TO PLAYER.set_stopTime
```

When we're we are less than 50 units away from the speaker - as we are when we enter the world - the ProximitySensor sends an event into the AudioClip node which starts the sound playing. When we move more than 50 units from the speaker, the ProximitySensor sends another event into the AudioClip which stops the sound from being played.

You can hear the sound begin anew every time you trigger the ProximitySensor; this is different behavior from the sound's audible area as set by maxBack, maxFront, minBack and minFront, because the sound starts anew every time.

Framed!

Although the ProximitySensor is there, it's difficult to tell exactly where it lies. Using the IndexedLineSet node and the eight points of the ProximitySensor, let's draw the cube which describes the volume within the ProximitySensor. This will help us to see exactly where the sensor is active:

```
#VRML V2.0 utf8
# This is the fourteenth example on sound
# Give it a background so it looks reasonable
Background {
```

```
      skyColor [ 0.4 0.7 0.8 ] # bluish
}
# Create the Cone/Cylinder pair for the speaker
Transform {
      children [
            # Create the ProximitySensor in this Transform
            DEF SENSOR ProximitySensor {
                  size 50 50 50 # 50 units on a side
            }
            Shape {
                  appearance Appearance {
                        material DEF SPEAKER Material {
                              diffuseColor 0.25 0.1 0.1
                              specularColor 0.8 0.7 0.8
                              shininess 0.9
                        }
                  }
                  geometry Cone {}
            }
            # The Cylinder inside a transform as well
            Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material USE SPEAKER # reuse
                              }
                              geometry Cylinder {
radius 0.5
                                    height 1.0
                              }
                        }
                  ]
                  translation 0 0.45 0
            }
      ]
      rotation 1 0 0 -1.78 # speaker rotated to face you
}
# Now let's define the Sound node and AudioClip
Sound {
      intensity 1.0           # full loudness
      maxBack 100       # far away
      maxFront 100            # same to front
      minBack     10          # close up
      minFront  10            # and close in front
      spatialize TRUE   # real 3D sound
      source DEF PLAYER AudioClip { # Here's the clip itself
            loop TRUE         # loop when playing
            startTime 0 # must start play
            stopTime 1  # must start play
            url [ "milnet.wav" ] # point to sample
      }
}
# ROUTE from the ProximitySensor into the AudioClip
# To turn things on when we're close
ROUTE SENSOR.enterTime TO PLAYER.set_startTime
# ROUTE from the ProximitySensor into the AudioClip
# To turn things off when we're far
```

```
ROUTE SENSOR.exitTime TO PLAYER.set_stopTime
```

Now we can see the ProximitySensor, and hear its effects.

That’s about it for sounds and backgrounds; now let’s put together what we’ve learned, and go on to build something truly useful…