Chapter 18: A Winning Script

State of Mind

While we've covered a lot of the basic features of interactivity in VRML, large gaps remain in the picture. For example, we can make a light switch from a TouchSensor, but it only operates momentarily – while we actively push down upon it. The light switch has no memory – or, as we say in computer-land, it has no *state information*. This state information allows objects to contain persistent qualities, such as whether the switch is now on or off.

We can hold persistent information – state information – within VRML worlds by placing it into a VRML field. But that will only work for state information normally associated with a node. A PointLight node will remember what color it shines, but, there is no field in the PointLight node that tells you a TouchSensor has turned it on.

What we would need – even for something as simple as a light switch – would be the ability to create our own fields, places where we could store our own state information. We'd also need a place where we could work out a little logic, so we could make some basic decisions; if the light is off and the TouchSensor is activated we should turn the light on, but if the light is on and the TouchSensor is activated we should turn the light off. Logic as simple as that could dramatically extend the kinds of interactivity we can create in VRML.

Have It Your Way!

One node in VRML has qualities that mark it as different from other VRML nodes; the Script node has a flexible field definition, which allows you define the fields to maintain your state information; it also has a flexible event definition – you can craft Script nodes with any combination of eventIn and eventOut declarations, so that it exactly fits your needs. Here's the basic definition of the Script node, with its three fields:

```
Script { # Definition of the Script node
      mustEvaluate            # SFBool
      directOutput            # SFBool
      url   []                # MFString, mult values.
}
```

The only field of importance to us now is the url field. The url field may contain a URL pointing to some logic which can be executed by the node (we'll come to understand how that happens in just a minute), or it can contain the logic, entirely within the url field.

What do we mean by "logic"? VRML isn't a programming language, it's a declarative language, meaning there's no way to make decisions inside of VRML. The simple expression, "do this or do that," is far beyond anything we can say in VRML. On the other hand, programming languages like Java and JavaScript are expressly designed to

make short work out of decisions; but they're quite poor at declarative tasks like, "create a green sphere and make it rotate". Putting the two together – VRML and Java or VRML and JavaScript – means that expressive power of a declarative language gets multiplied by the logical power of a programming language, a very potent combination.

The url field can contain JavaScript, within the Script node – just as a Web page can contain JavaScript within its HTML definition. If the url field contains a URL, rather than JavaScript, it can point to a valid JavaScript file, or a valid Java class file. So you have a choice: you can use JavaScript within the Script node, or you an reference Java and JavaScript with a URL. You can't put Java directly into a Script node, because Java must be complied (turned into machine code) before it can be executed by the computer, while JavaScript is interpreted (translated on-the-fly).

For the duration of this book, we'll focus on JavaScript within the Script node; JavaScript is very easy to learn, and its tight integration with the Script node makes it almost trivial to create step-by-step examples that will be easy to understand. Don't be frightened by JavaScript programming; we'll step into it slowly, moving from basic concepts into more advanced material, as we've done with 3D computer graphics - and you understand 3D graphics now, don't you?

A Real Switch

Let's begin by creating a real light switch, adapting the example from chapter 17. While we can use the TouchSensor and Box nodes as the switch itself, we need to add a Script node to handle the state information and logic associated with the switch. What kind of state information do we need? The only important bit of information is whether the switch is on or off, that's a single bit of Boolean information, TRUE or FALSE. That means we'd want to add a field of data type SFBool to our Script node, to hold the current state of the switch.

In our Script node we need to define two events, an eventIn which could receive the isActive eventOut from the TouchSensor, and an eventOut with a type of SFBool that can be routed to the set_on eventIn of the PointLight. Here's how that Script node might look:

```
# A Script node for a light switch
# We maintain state in the mySwitchOn field
# Get events through touchedSwitch
# Send events through lightMe
Script {
      field SFBool mySwitchOn FALSE       # state info.
      eventIn SFBool touchedSwitch        # from TouchSensor
      eventOut SFBool lightMe             # to PointLight
}
```

This definition – which is still incomplete - looks a bit different from what we're used to. To add a field to the Script node, we use the keyword field, followed by the field's data

type, the given name of the field, and then its initial value. In this case, we created a field of type SFBool named mySwitchOn, with an initial value of FALSE.

To create an eventIn, we use the eventIn keyword, followed by the event's data type, and the given name of the event. In the Script node given above, we created an eventIn of type SFBool named touchedSwitch. The same rules apply for creating an eventOut; here we created an eventOut with data type SFBool and a given name lightMe.

We have now defined an input to the Script node, an output from it, and a place to store some state information within in. That's not the whole story, though, because we need some logic – JavaScript within the url field – which processes the eventIn messages, and, based upon that input, generates eventOut messages. To do this, we must create a JavaScript *function* – a function is a basic unit of JavaScript, just as a node is a basic unit of VRML – with the same given name as the eventIn we want it to process. Our Script node has only one eventIn – touchedSwitch – so we'll create a JavaScript function with the name touchedSwitch, and the Script node will send all eventIn messages from touchedSwitch to the JavaScript function. That's how you pass messages from VRML to JavaScript through the Script node; you give them the same names. (This is legal because JavaScript is parsed separately from the VRML, so the names won't interfere with each other.)

The JavaScript consists of an *if-else* test, a basic building block of all programming logic. The if-else test determines if a condition – given in parentheses following the if keyword – is true. If it is, then the if test executes some JavaScript – given in braces – associated with the if. If the test fails – that is, the condition is false – then some JavaScript following the else keyword is executed. In other words, the if-else test says, "Test this thing: here's what to do if the test is true, and here's what to do if the test is false."

Here's how the Script node would look, with our JavaScript logic inside the url field:

```
# A Script node for a light switch
# We maintain state in the mySwitchOn field
# Get events through touchedSwitch
# Send events through lightMe
Script {
      field SFBool mySwitchOn FALSE       # state info.
      eventIn SFBool touchedSwitch        # from TouchSensor
      eventOut SFBool lightMe             # to PointLight
      url ["javascript:       // javascript comments
            function touchedSwitch(ts) {
                  if (mySwitchOn) { // if true, do this
                        mySwitchOn = false; // not true
                        lightMe = false; // turn light off
                  } else {    // if false, do this
                        mySwitchOn = true; // not false
                        lightMe = true; // turn light on
                  }
            } // End the function
       " ] # End of JavaScript, back to VRML comments
}
```

The declaration of the Script node remains the same; we have one field, one eventIn, and one eventOut. The url field has an MFString data type, so we need to enclose our string of JavaScript inside brackets. We open the SFString inside the url field with the keyword javascript, followed by a colon; this tells the parser that inline JavaScript is being used. Once we begin writing the JavaScript, comments must begin with the double-slash (//), and continue all the way to the closing quote of the SFString, when comments revert to the normal VRML pound (#).

On the next line we begin the JavaScript, with the function keyword, followed by the given name of the function, touchedSwitch, which matches the eventIn defined in the declarations of the Script node.

Now here comes a little trick. After the given name of the function, you'll see a pair of parentheses with the letters ts inside. This is the given name of a *variable* – a variable, like a field, is something that holds a value – that's being *passed* the function. Why did we do this? Well, even though VRML can deal with the fact that there's both an eventIn and a JavaScript function named touchedSwitch, JavaScript can't tolerate two items with the same name. So, for the duration of the function touchedSwitch, the eventIn touchedSwitch takes the variable name ts. (We could have used anything, the ts is just to remind us that it stands for touchedSwitch.) We don't use the variable ts in our function, but it's nice to have it, just the same.

Inside the braces, we find the *body* of the function. We begin with the if keyword, followed by a pair of parentheses containing the value we want to test. We want to test the light's state, which we keep in mySwitchOn, so that's what we drop into the parentheses. If mySwitchOn is TRUE, we'll execute the JavaScript in the braces immediately following the if keyword. These two JavaScript *statements* are separated by semi-colons; the first *assigns* the value of mySwitchOn to FALSE – in JavaScript, we use lowercase for true and false, while in VRML we use uppercase for TRUE and FALSE. It's a small difference, but an important one.

The second line contains some magic: when we assign lightMe to false, it actually sends an event through the eventOut. That – once we add a PointLight – will actually turn the light on and off. The assignment causes the event to emit a message, in this case, an SFBool data type with a value of FALSE.

Now we encounter the else keyword: here's where we'll go if the test of mySwitchOn fails, that is, if the light is off. Again we have two statements: the first of them assigns the value true to mySwitchOn – the light's off, we're turning it on; the second statement emits the message TRUE through lightMe. Once we ROUTE lightMe to the set_on eventIn of a PointLight node, this will actually turn the light on.

Putting it all together, here's the final example from chapter 17, with a Script node adding state information, so we have a real light switch:

```
#VRML V2.0 utf8
# This is the first example on scripting
# Keep all of the lightbulb together and above box
```

```
	Transform {
		children [
			DEF LIGHT PointLight { on FALSE }   # it's off
			Shape {
				appearance Appearance {
					material Material {
						emissiveColor 1 1 0.8 # shines
					}
				}
				geometry Sphere { radius 1.25 }
			}
			# Cylinder goes inside translation node
			Transform {
				children [
					Shape {
						appearance Appearance {
							material Material { # metalic
								diffuseColor 0.5 0.5
0.5
								shininess 0.9
								specularColor 1 1 1
							}
						}
						geometry Cylinder { radius 0.5
height 1
}
					}
				]
				translation 0 1.5 0 # move up a bit
			}
		]
		translation 0 5 0 # above the box
	}
	# Using a Group node to put the box and sensor together
	Group {
		children [
			DEF SENSOR TouchSensor { } # fine as default
			# And here's the white Box
			Shape {
				appearance Appearance {
					material Material {
						diffuseColor 1 1 1
					}
				}
				geometry Box { }
			}
		]
	}
	# A Script node for a light switch
	# We maintain state in the mySwitchOn field
	# Get events through touchedSwitch
	# Send events through lightMe
	DEF SWITCH_SCRIPT Script {
		field SFBool mySwitchOn FALSE       # state info.
		eventIn SFBool touchedSwitch        # from TouchSensor
		eventOut SFBool lightMe             # to PointLight
```

```
            url [ "javascript:            // javascript comments
                        function touchedSwitch(ts) {
                        if (mySwitchOn) { // if true, do this
                              mySwitchOn = false; // not true
                              lightMe = false; // turn light off
                        } else {    // if false, do this
                              mySwitchOn = true; // not false
                              lightMe = true; // turn light on
                        }
                  } // End the function
             " ] # End of JavaScript, back to VRML comments
      }

      # Route between the SENSOR eventOut isOver
# and the SWITCH_SCRIPT eventIn touchedSwitch.
ROUTE SENSOR.isActive TO SWITCH_SCRIPT.touchedSwitch
# Route between the SWITCH_SCRIPT eventOut lightMe
# and the LIGHT eventIn set_on
ROUTE SWITCH_SCRIPT.lightMe TO LIGHT.set_on
```

We've given the Script a name, SWITCH_SCRIPT, so we can access its fields and events. The two ROUTE statements pass events from the TouchSensor to the Script, and from the Script to the PointLight.

What we've gotten is exactly the same behavior we had before! Why is that? The answer lies in the isOver eventOut message emitted by the TouchSensor; isOver sends a TRUE message when the pointer is clicked on the Box, but it also sends a FALSE message when the pointer is un-clicked on the Box. We don't check to see if we're clicking or un-clicking – and we only care about the clicks. We'll need to modify the JavaScript a little bit, and test to see if we're clicking or un-clicking. That message – routed into the Script node eventIn touchedSwitch, is available to us in the JavaScript as the variable ts. We'll test that variable, and – only if it is true – will we change the state of the light switch; in other words, we will ignore un-clicks. Here's the updated Script node:

```
      # The second – incomplete – example on scripting
      # A Script node for a light switch
      # We maintain state in the mySwitchOn field
      # Get events through touchedSwitch
      # Send events through lightMe
      DEF SWITCH_SCRIPT Script {
            field SFBool mySwitchOn FALSE       # state info.
            eventIn SFBool touchedSwitch        # from TouchSensor
            eventOut SFBool lightMe             # to PointLight
            url [ "javascript:            // javascript comments
                        function touchedSwitch(ts) {
                        if (ts) { // test – are we clicking?
                              if (mySwitchOn) { // if true, do this
                                    mySwitchOn = false; // not true
                                    lightMe = false; // turn light off
                              } else {    // if false, do this
                                    mySwitchOn = true; // not false
                                    lightMe = true; // turn light on
                              }
```

```
                    } // We ignore unclicks, so no else
                } // End the function
            " ] # End of JavaScript, back to VRML comments
      }
```

We only added one test, in the first line of the JavaScript function. Here we test the value of ts, which is an SFBool; if it is true, we then go through our other tests, and turn the light on or off. If the test fails, we don't do anything – so we don't need an else keyword following the closing brace of the *body* of the if. This should work a bit better.

As you see, we've created a real light switch. The first time you click on the Box, the light goes on, and with the next click the light goes of, and so on, forever. If this seems like a whole lot of work for a just a tiny bit of interactivity, take a closer look at the Script node; only seven lines of JavaScript to create a fully functional *visible* light switch. You couldn't do that in any other computer language in less than a few hundred.

Stop The World, I Want to Get Off!

Although we've learned how to employ the TimeSensor to animate objects when used in conjunction with interpolators, we have only been able to experiment with TimeSensor nodes set to start up automatically when the world is loaded by the browser. Now that we understand how to use the Script node to send messages to other nodes, we can build a Script node designed to send messages to TimeSensor nodes, to start and stop interpolators.

For our next example, let's keep the switch we created from the Box and TouchSensor in our last example, but let's replace the light with a model of the spinning Earth. When we click on the Box, we'll start the animation, and when we click on the Box again, we'll stop the animation. How do we do this? The TimeSensor has two fields, startTime and stopTime, which determine if the TimeSensor is active. If startTime is greater than stopTime, the TimeSensor is active; if stopTime is greater than startTime, the TimeSensor stops.

As it so happens, TimeSensor has an eventOut named touchTime, which sends a message of data type SFTime every time the TimeSensor detects a click. That touchTime value can be processed through a Script node, using the switch logic we've already designed; if the animation is off, the Script starts the animation by passing the touchTime value through to the set_startTime eventIn of a TimeSensor; if the animation is on, the Script sends the touchTime value through to the set_stopTime eventIn of the TimeSensor. All of that might look like this:

```
#VRML V2.0 utf8
# This is the third example on scripts
DEF EARTH_XFORM Transform {
      children [
                  # Create semi-realistic model of Earth
Shape {           #  Create a visible shape
                        appearance Appearance {
                              texture ImageTexture { # texture
```

```
                              url [ "worldmap.jpg" ]
                        }
                  }
                  geometry Sphere { }     # Make Earth!
}
      ]
}
# Using a Transform node to put the box and sensor together
Transform {
      children [
            DEF SENSOR TouchSensor { } # fine as default
            # And here's the white Box
            Shape {
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 1 1
                        }
                  }
                  geometry Box { }
            }
      ]
      translation 0 –10 0 # underneath Earth
}
# An sixty second timer that will not start by itself
DEF TIMER TimeSensor {
      loop TRUE
      cycleInterval 60  # sixty-second timer
      startTime   0     # start less than stop
      stopTime    1     # means it will not start by itself
}
# Move the Sphere through 360 degrees of rotation
DEF SPHERE_ROTATE OrientationInterpolator {
      key [ 0, 0.5, 1 ] # start, stop keyframes
      keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
}
# A Script node to turn rotation on and off
# We maintain state in the mySwitchOn field
# Get events through whenTouched
# Send events through spinTime
DEF SWITCH_SCRIPT Script {
      field SFBool mySwitchOn FALSE       # state info.
      eventIn SFTime whenTouched          # from TouchSensor
      eventOut SFTime startTheTimer       # to startTime
      eventOut SFTime stopTheTimer        # to stopTime
      url [ "javascript:            // javascript comments
            function whenTouched(wt) { // eventIn name
passed as wt
            if (mySwitchOn) { // if true, do this
                  mySwitchOn = false; // not true
                  stopTheTimer = wt; // stopTime is click
time
            } else {    // if false, do this
                  mySwitchOn = true; // not false
                  startTheTimer = wt; // startTime is click
time
            }
      } // End the function
```

```
        " ] # End of JavaScript, back to VRML comments
}
# Route Box's click event to the Script node
ROUTE SENSOR.touchTime TO SWITCH_SCRIPT.whenTouched
# Route Script startTime message to TimeSensor
ROUTE SWITCH_SCRIPT.startTheTimer TO TIMER.set_startTime
# Route Script stopTime message to TimeSensor
ROUTE SWITCH_SCRIPT.stopTheTimer TO TIMER.set_stopTime
# First we ROUTE the TimeSensor into the OrientationInterpolator
ROUTE TIMER.fraction_changed TO SPHERE_ROTATE.set_fraction
# Last we ROUTE the OrientationInterpolator into the Transform
ROUTE SPHERE_ROTATE.value_changed TO EARTH_XFORM.set_rotation
```

The declaration of the Script node looks a bit different from our last example – most importantly, there are now two eventOut declarations – startTheTimer gets routed to the set_startTime eventIn of the TimeSensor, while stopTheTimer gets routed to the set_stopTime eventIn of the TimeSensor.

Every time you click on the Box, the JavaScript function whenTouched executes; if mySwitchOn is FALSE, the value of whenTouched is emitted by startTheTimer and the TimeSensor starts – because the TimeSensor field startTime is now greater than the stopTime field. If mySwitchOn is TRUE, the value of whenTouched is emitted by stopTheTimer and the TimeSensor stops, because the TimeSensor field stopTime is greater than the startTime field.

Does it work? We should see a static Earth when we enter the world, which begins to turn after we click on the Box underneath it. If we click on the Box again, the Earth should cease spinning. It does that just fine. There is a bit of a problem though – when we click on the Box again, after we've stopped the Earth from spinning, it starts up again – but from the beginning of its rotation, not from where we've stopped it. Why is this? When we restart the TimeSensor, the message emitted by the eventOut fraction_changed returns to its initial value of 0, so the OrientationInterpolator rewinds the animation to the beginning; the result is an awkward-looking resetting of the animation. How do you get around it? You'd need to change the key frame information in the keyValue field of OrientationInterpolator to reflect a partially uncompleted portion of the cycle; once the TimeSensor restarts, the animation picks up from where it left off. That's a bit too complicated for us at this moment, though we'll cover this topic in a later example.

Context and Text

We try to avoid using text in virtual worlds, because its use often weakens the impact of the world, and makes a good design flawed. Of course, no blanket statement can every be completely true; a time and place for judicious use of text will always exist. We haven't talked about text in the virtual world, but VRML features an extensive facility for the layout of text in various sizes, fonts, and styles. It's all described within two nodes: Text, which handles the display a particular string of characters, and FontStyle, which allows you to manipulate the visible qualities of displayed text. The Text node lays out all of the basics:

```
      Text { # Definition of the Script node
            string []         # MFString, exposedField
            fontStyle               # SFNode, exposedField
            length []               # MFFloat, exposedField
            maxExtent               # SFFloat, exposedField
      }
```

A single Text node can display multiple strings of text, because the string field has a data type of MFString; each string is separated from the others by a comma. This means that you can "compose" a long string of text from several smaller strings – a very powerful feature, as we'll soon see.

The length field contains a list of floating-point numbers that specify the length of each string within the string field, in the units of the local coordinate system. For each SFString within the string field, there should be a matching value in the length field. For example, if I wanted to display the string "Hello" and make it 2 units long, here's how I might do it:

```
      #VRML V2.0 utf8
      # This is the fourth example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello" ]  # this is the text
                        length [ 2 ]            # and how long it is
            }
      }
```

We should see the word "Hello" floating in the black void of cyberspace.

If we wanted to add the phrase "world!" – thus creating the magical phrase uttered by every programmer learning a new programming language – we'd add another SFString value in the string field, and another value to the length field:

```
      #VRML V2.0 utf8
      # This is the fifth example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello", "world!" ]  # this is the text
                              length [ 2, 2 ]   # and how long it is
            }
      }
```

Which puts the strings in top-down order, one over another.

If we don't use put values into the length field, the browser makes up its own mind about how big the text string are:

```
      #VRML V2.0 utf8
      # This is the sixth example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello", "world!" ]  # this is the text
            }
      }
```

This example doesn't look very much different.

But if we set the length value to 5 for the first string and 1 for the second, we'll see the difference:

```
      #VRML V2.0 utf8
      # This is the seventh example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello", "world!" ]  # this is the text
                  length [ 5, 1 ] # wide and thin
            }
      }
```

The top string is too wide; while the bottom string is too thin, all jumbled together.

A Sense of Style

The one big problem with using text in the virtual world comes from the nature of text as a language-specific medium; what I tell you in English won't make sense in French or Japanese. For that matter, the left-to-right, top-to-bottom direction of English – or any other languages which use the "Roman" alphabet – won't work with Hebrew, written right-to-left, or Japanese, written top-to-bottom, right-to-left, the exact inverse of English.

The VRML FontStyle node allows you to have precise control over the visible characteristics and placement of characters displayed using the Text node. Here's the definition of FontStyle:

```
      FontStyle { # Definition of the Script node
            family []               # MFString, mult. values
            horizontal        # SFBool
            justify []        # MFString, mult. values
            language                # SFString
            leftToRight       # SFBool
            size                    # SFFloat
            spacing                 # SFFloat
            style             # SFString
            topToBottom       # SFBool
      }
```

Most of these fields have names that are self-explanatory; the horizontal field allows you to create vertical text if you set it to FALSE. That means you can create marquee text! In our next example, we'll use Text and FontStyle to spell out "Hello!" vertically:

```
      #VRML V2.0 utf8
      # This is the eighth example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello" ]  # this is the text
                  fontStyle FontStyle { # text style
                        horizontal FALSE  # vertically
                  }
            }
      }
```

Which does create a marquee effect.

The leftToRight field controls the direction of character placement; its default value, TRUE, creates the normal left-to-right placement associated with English, but if we set the value to FALSE, we get right-to-left placement:

```
      #VRML V2.0 utf8
      # This is the ninth example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello" ]  # this is the text
                  fontStyle FontStyle { # text style
                        leftToRight FALSE # write right to left
                  }
            }
      }
```

In this example we set leftToRight to FALSE, resulting in characters that spell themselves out in reverse.

The topToBottom field controls up-down flow of text; if TRUE, its default value, the text flows from the top to the bottom of the scene. But, if FALSE, the text reverses direction and flows from the bottom to the top. Here we take our eighth example, but now it flows both vertically and bottom to top:

```
      #VRML V2.0 utf8
      # This is the tenth example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello" ]  # this is the text
                  fontStyle FontStyle { # text style
                        horizontal FALSE  # vertically
                        topToBottom FALSE  # flow upward
                  }
            }
      }
```

Now we see a word that has its first character at the bottom of the screen, and its last at the top.

The size and spacing fields of the FontStyle node allow you to have precise control over the size of the displayed characters. The size specifies the height in VRML units of the rendered characters, while the spacing field allows you to adjust the spacing between lines; a value of 1.0 - the default - implies single spacing, while a value of 2.0 is double spacing. Here's our "Hello world!" example; the characters are double their normal size (the default value is 1.0) and at one-and-a-half spacing:

```
      #VRML V2.0 utf8
      # This is the eleventh example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello", "world!" ]  # this is the text
                  fontStyle FontStyle {
                        size 2.0  # double character sizes
                        spacing 1.5 # space-and-a-half
                  }
      }
```

We see bigger characters with more space between the lines.

The Font of Wisdom

The family field allows you to select the font family; fonts typically have qualities such as *serif* – meaning the characters have small “hooks” on the strokes, or *sans-serif*, meaning they have no hooks. In addition, there are *monospaced* fonts, where every character uses the same spacing. This makes monospaced fonts look as if they’ve been printed on a typewriter; the code examples in this book are all printed with monospaced fonts. The three possible SFString values for the family field reflect these three selections; SERIF means use a serif font, SANS, a sans-serif font, and TYPEWRITER tells the browser to use a monospace font.

In the next example, we’ll spell out “Hello there world!” using the three different font families, so you can see the differences between them. We need to use Transform nodes to keep the Text separated properly:

```
      #VRML V2.0 utf8
      # This is the twelfth example on scripting
      # Describe the serif string “Hello”
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Hello" ]  # this is the text
            }
      }
      Transform {
            children [

                  # Describe the sans-serif string “there”
Shape {           #  Create a visible shape
                        appearance Appearance {
                              material Material { # texture
                                    diffuseColor 1 1 1 # White text
                              }
                        }
                        geometry Text {
                              string [ "there" ]  # more text
                              fontStyle FontStyle { # style info
                                    family [ "SANS" ] # sans-serif
                              }
                        }
                  }
                  # Final Transform for last string
                  Transform {
                        children [
                              # Describe the monospaced string “there”
Shape {           #  Create a visible shape
                              appearance Appearance {
```

```
                                          material Material { # texture
                                                diffuseColor 1 1 1 #
White text
                                          }
                                    }
                                    geometry Text {
                                          string [ "world!" ]  # more
text
                                          fontStyle FontStyle { # style
info
                                                family [ "TYPEWRITER" ]
# mono
                                          }
                                    }
                              }
                        ]
                        translation 0 –2 0 # underneath
                  }
            ]
            translation 0 –2 0  # put it underneath
      }
```

We see the three strings, each with distinct visual characteristics.

The top string features the “hooks” common to serif font families, while the middle example features the streamlined look of sans-serif fonts. The final string really does look like its been printed by a virtual typewriter.

The family field contains an MFString data type – this means that you can define multiple font families within the TextStyle node. Why would you do this? You can’t always guarantee that a computer can support any of these font types – it might depend more upon the computer’s support for fonts than the browser. You can give several choices, in descending order of preference – just as you can with a URL.

However, if your computer has excellent support for fonts – as many do – you can also specify the fonts by name. This may mean that you won’t get the same results from user to user, but you can produce some mind-blowing text. Here we’ll take the previous example, but use *Impact* font for the first word, *Arial* font for the second, and *Lucida Handwriting* for the third – but in each case we give a backup, just in case the font doesn’t exist on another user’s computer.

```
      #VRML V2.0 utf8
      # This is the thirteenth example on scripting
      # Describe the Impact string
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "Impact Font" ]  # this is the text
                  fontStyle FontStyle {
```

```
                        family [ "Impact", "SERIF" ] # name & backup
                  }
            }
      }
      Transform {
            children [

                  # Describe the Arial string
Shape {           #  Create a visible shape
                        appearance Appearance {
                              material Material { # texture
                                    diffuseColor 1 1 1 # White text
                              }
                        }
                        geometry Text {
                              string [ "Arial" ]  # more text
                              fontStyle FontStyle { # style info
                                    family [ "Arial", "SANS" ] # font,
backup
                              }
                        }
                  }
                  # Final Transform for last string
                  Transform {
                        children [
                              # Describe the monospaced string “there”
Shape {           #  Create a visible shape
                              appearance Appearance {
                                    material Material { # texture
                                          diffuseColor 1 1 1 #
White text
                                    }
                              }
                              geometry Text {
                                    string [ "Lucida Handwriting
Font" ]
                                    fontStyle FontStyle { # style
info
                                          family [ "Lucida
Handwriting" "SANS" ]
                                    }
                              }
                        }
                        ]
                        translation 0 –2 0 # underneath
                  }
            ]
            translation 0 –2 0  # put it underneath
      }
```

Because my system has these fonts installed, I can see them displayed correctly.

If you want to find out which fonts your system has installed, your word processor generally presents a list of them to you when you modify the font of a selection of text.

Be Bold!

The style field of the FontStyle node allows you to select the essential types of emphasis we normally associate with fonts – bolding and italics. By default, the SFString value inside the field is PLAIN, meaning no bold or italic. Using the value BOLD in the field bolds the text (making it look as if it has been broaden and darkened), while using the value ITALIC in the field will give the characters the slant characteristic of italicization. If you need to use both together, you'd use the field value BOLDITALIC. We'll adapt our twelfth example, bold the first phrase, italicize the second phrase, and do both to the third phrase:

```
      #VRML V2.0 utf8
      # This is the fourteenth example on scripting
      # Describe the bold string.
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "This is Bold." ]  # this is the text
                  fontStyle FontStyle {
                        style "BOLD"      # bold characters
                  }
            }
      }
      Transform {
            children [

                  # Describe the italic string.
Shape {           #  Create a visible shape
                        appearance Appearance {
                              material Material { # texture
                                    diffuseColor 1 1 1 # White text
                              }
                        }
                        geometry Text {
                              string [ "This is italic." ]  # more text
                              fontStyle FontStyle { # style info
                                    style "ITALIC" # italicize
                              }
                        }
                  }
                  # Final Transform for last string
                  Transform {
                        children [
                              # Describe the bold and italic string
Shape {           #  Create a visible shape
                                    appearance Appearance {
                                          material Material { # texture
                                                diffuseColor 1 1 1 #
White text
                                          }
```

```
                                    }
                                    geometry Text {
                                          string [ "This is bold and
italic" ]
                                          fontStyle FontStyle { # style
info
                                                style "BOLDITALIC" #
bold italic
                                          }
                                    }
                              }
                        ]
                        translation 0 –2 0 # underneath
                  }
            ]
            translation 0 –2 0  # put it underneath
      }
```

Each of these strings will use the default SERIF font, but with different styles applied to each of them.

Each of the fields of FontStyle can be freely mixed with other fields, so the possibilities are enormous. The final field we'll cover here, the language field, opens the door to languages other than those written with Roman character sets. If your computer has support for other languages – mine doesn't – you can specify the character set as a string as an SFString value in the language field. You can't just put "Japanese" into the language field and expect the browser to understand – there's a standard for language abbreviations, which is available on the Web at the following URL:

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

This document gives the International Standard Organization's two-letter abbreviations for any of the languages you're ever likely to encounter. If a written language exists, you can probably find it in the list. For example, the abbreviation for Japanese is jp, while the abbreviation for Yiddish is ji – so if you want to create a document with Japanese text strings, you must set to jp the language field of the FontStyle node associated with the Text node containing the Japanese character string.

Here's an example of how to create a Japanese text string:

```
      #VRML V2.0 utf8
      # This is the fifteenth example on scripting
      # Create visible object, it's a string of text!
Shape {           #  Create a visible shape
            appearance Appearance {
                  material Material { # texture
                        diffuseColor 1 1 1 # White text
                  }
            }
            geometry Text {
                  string [ "VRML 2.0" ]  # this is the text
                  fontStyle FontStyle { # text style
                        language "jp"     # use Japanese
```

```
                }
        }
}
```

This example will only work correctly if you have support for Japanese fonts on your machine – otherwise, you'll see it in your regular font. If you don't specify a value in the language field, you'll use your normal character set – whatever that might be. (It varies from country to country.)

Much Ado About Text

If you're wondering why we suddenly stopped our discussion of Script nodes to begin a digression on the use of Text and FontStyle, you've probably never programmed a computer before. You see, when good programs go bad – when they fail – the programmer has to *debug* them. VRML has no facilities for debugging Script nodes – and the help you get from the browser ranges from the nonexistent (Cosmo) to the only slightly more helpful (WorldView). A long time ago programmers learned how to *trace* their programs by emitting text strings at certain points in the program's execution. That way, the programmer could follow the path of the program. It's a rudimentary technique – but it will help you immeasurably when you get lost inside JavaScript functions that should work but don't.

First, let's develop a very modest debugger – using the Text node – that tells us the state of our light switch from the second example. We'd need to setup a Text node with a string value of "Off", because that's the light is turned off when we load the world. However, we'll have to modify the JavaScript in the Script node, so that – in addition to turning the light on and off – it also modifies the string field of the Text node.

```
#VRML V2.0 utf8
# This is the sixteenth example on scripting
# Keep all of the lightbulb together and above box
Transform {
      children [
            DEF LIGHT PointLight { on FALSE }   # it's off
            Shape {
                  appearance Appearance {
                        material Material {
                              emissiveColor 1 1 0.8 # shines
                        }
                  }
                  geometry Sphere { radius 1.25 }
            }
            # Cylinder goes inside translation node
            Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material Material { # metalic
                                          diffuseColor 0.5 0.5
0.5
                                          shininess 0.9
```

```
                                                specularColor 1 1 1
                                          }
                                    }
                                    geometry Cylinder { radius 0.5
height 1
}
                              }
                        ]
                        translation 0 1.5 0 # move up a bit
                  }
            ]
            translation 0 5 0 # above the box
      }
      # Using a Group node to put the box and sensor together
      Group {
            children [
                  DEF SENSOR TouchSensor { } # fine as default
                  # And here's the white Box
                  Shape {
                        appearance Appearance {
                              material Material {
                                    diffuseColor 1 1 1
                              }
                        }
                        geometry Box { }
                  }
            ]
      }
      # Inside a Transform node we'll put our text debugger
      Transform {
            children [
                  # And here's the green text
                  Shape {
                        appearance Appearance {
                              material Material {
                                    diffuseColor 0 1 0
                              }
                        }
                        geometry DEF DEBUG_TEXT Text {
string [ "Off" ] # light starts off
}
                  }
            ]
            translation –0.5 –2 1 # below and in front
      }
      # A Script node for a light switch
      # We maintain state in the mySwitchOn field
      # Get events through touchedSwitch
      # Send events through lightMe
      DEF SWITCH_SCRIPT Script {
            field SFBool mySwitchOn FALSE       # state info.
            field MFString offText [ "Off" ] # Preset string
            field MFString onText [ "On" ] # Preset string
            eventIn SFBool touchedSwitch        # from TouchSensor
            eventOut SFBool lightMe             # to PointLight
            eventOut MFString changedText       # to Text node
```

```
            url [ "javascript:            // javascript comments
                        function touchedSwitch(ts) {
                        if (ts) { // test – are we clicking?
                              if (mySwitchOn) { // if true, do this
                                    mySwitchOn = false; // not true
                                    lightMe = false; // turn light off
                                    changedText = offText; // text
reads off
                              } else {    // if false, do this
                                    mySwitchOn = true; // not false
                                    lightMe = true; // turn light on
                                    changedText = onText; // text reads
on
                              }
                        } // We ignore unclicks, so no else
                  } // End the function
             " ] # End of JavaScript, back to VRML comments
      }
      # Route from the TouchSensor to the Script
      ROUTE SENSOR.isActive TO SWITCH_SCRIPT.touchedSwitch
      # Route from Script to PointLight
      ROUTE SWITCH_SCRIPT.lightMe TO LIGHT.set_on
      # Route from Script to Text
      ROUTE SWITCH_SCRIPT.changedText TO DEBUG_TEXT.set_string
```

This Script node defines one eventIn, from the TouchSensor, and three field values. One of the fields maintains the state of PointLight, while they other two define pre-made MFString values which will be used in the Text debugger. When the light is switched off, changedText emits a message of "Off"; when the light is switched on, it emits a message of "On".

That's pretty cool – now we can use Text nodes to help us understand what's going on inside of our Script nodes. But let's take this example just a little further. As you can see, the color of the Text is green, which it gets from the associated Material node. But we should really color the Text red when the switch is off, and green when the switch is on. We can do that by adding yet another eventOut to the Switch node, which sends an SFColor value to that Material node during every transition. Once again, we'll use fields in the Script node to establish preset SFColor values:

```
      #VRML V2.0 utf8
      # This is the seventeenth example on scripting
      # Keep all of the lightbulb together and above box
      Transform {
            children [
                  DEF LIGHT PointLight { on FALSE }   # it's off
                  Shape {
                        appearance Appearance {
                              material Material {
                                    emissiveColor 1 1 0.8 # shines
                              }
                        }
                        geometry Sphere { radius 1.25 }
                  }
                  # Cylinder goes inside translation node
```

```
                  Transform {
                        children [
                              Shape {
                                    appearance Appearance {
                                          material Material { # metalic
                                                diffuseColor 0.5 0.5
0.5
                                                shininess 0.9
                                                specularColor 1 1 1
                                          }
                                    }
                                    geometry Cylinder { radius 0.5
height 1
}
                              }
                        ]
                        translation 0 1.5 0 # move up a bit
                  }
            ]
            translation 0 5 0 # above the box
      }
      # Using a Group node to put the box and sensor together
      Group {
            children [
                  DEF SENSOR TouchSensor { } # fine as default
                  # And here's the white Box
                  Shape {
                        appearance Appearance {
                              material Material {
                                    diffuseColor 1 1 1
                              }
                        }
                        geometry Box { }
                  }
            ]
      }
      # Inside a Transform node we'll put our text debugger
      Transform {
            children [
                  # And here's the green text
                  Shape {
                        appearance Appearance {
                              material DEF TEXT_COLOR Material {
                                    diffuseColor 1 0 0 # start red
                              }
                        }
                        geometry DEF DEBUG_TEXT Text {
string [ "Off" ] # light starts off
}
                  }
            ]
            translation –0.5 –2 1 # below and in front
      }
      # A Script node for a light switch
      # We maintain state in the mySwitchOn field
      # Get events through touchedSwitch
      # Send events through lightMe
```

```
      DEF SWITCH_SCRIPT Script {
            field SFBool mySwitchOn FALSE       # state info.
            field MFString offText [ "Off" ] # Preset string
            field MFString onText [ "On" ] # Preset string
            field SFColor textRed 1 0 0 # preset color
            field SFColor textGreen 0 1 0 # preset color
            eventIn SFBool touchedSwitch        # from TouchSensor
            eventOut SFBool lightMe             # to PointLight
            eventOut MFString changedText       # to Text node
            eventOut SFColor changedColor       # to Material node
            url [ "javascript:            // javascript comments
                        function touchedSwitch(ts) {
                        if (ts) { // test – are we clicking?
                              if (mySwitchOn) { // if true, do this
                                    mySwitchOn = false; // not true
                                    lightMe = false; // turn light off
                                    changedText = offText; // text
reads off
                                    changedColor = textRed; // text
turns red
                              } else {    // if false, do this
                                    mySwitchOn = true; // not false
                                    lightMe = true; // turn light on
                                    changedText = onText; // text reads
on
                                    changedColor = textGreen; // text
green
                              }
                        } // We ignore unclicks, so no else
                  } // End the function
             " ] # End of JavaScript, back to VRML comments
      }
      # Route from the TouchSensor to the Script
      ROUTE SENSOR.isActive TO SWITCH_SCRIPT.touchedSwitch
      # Route from Script to PointLight
      ROUTE SWITCH_SCRIPT.lightMe TO LIGHT.set_on
      # Route from Script to Text
      ROUTE SWITCH_SCRIPT.changedText TO DEBUG_TEXT.set_string
      # Route from Script to Material
      ROUTE SWITCH_SCRIPT.changedColor TO TEXT_COLOR.set_diffuseColor
```

Now the color follows the switch, too.

The Countdown Begins

For our final projects using Script and Text nodes, let’s build a countdown timer. Nothing fancy, let’s have it count down from 60 seconds to 0 seconds. We’ll need a TimeSensor with a cycleInterval of one second, and we’ll need to be listening to its cycleTime eventOut, which triggers every time the TimeSensor begins another cycle – that is, every second. We can use that event inside of a Script node to maintain an internal countdown value, and change the value of the string field of a Text node accordingly.

Here’s how that might look:

```
      #VRML V2.0 utf8
      # This is the eighteenth example on scripting
      # Begin with the Shape of the Text
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 1 1 1
                  }
            }
            geometry DEF COUNTER_TEXT Text {
string [ "60" ] # starts at 60 seconds
            }
      }
      DEF TIMER TimeSensor {
            loop TRUE         # repeat every second
            cycleInterval 1   # one second per
            startTime 1       # automatic start
            stopTime 0        # automatic start
      }
      DEF COUNTDOWN Script {
            field SFInt32 counter 60            # maintains 60 seconds
            field MFString countText [ "60" ] # output string
            eventIn SFTime eachSecond     # get eventIn every second
            eventOut MFString newText     # send updated text
            eventOut SFTime stopTheTimer  # send stop timer
            url [ "javascript:            // javascript comments
                        function eachSecond (es) {
                        counter = counter – 1; // one less second
                        countText[0] = counter;       // new value
                        newText = countText; // post the countdown
                        if (counter == 0) {     // at zero?
                              stopTheTimer = es;      // stop
TimeSensor
                        }
                  }
            " ] # End of JavaScript, back to VRML comments
      }
      # Route TimeSensor to Script
      ROUTE TIMER.cycleTime TO COUNTDOWN.eachSecond
      # Route Script to Text
      ROUTE COUNTDOWN.newText TO COUNTER_TEXT.set_string
      # Route Script to TimeSensor
      ROUTE COUNTDOWN.stopTheTimer TO TIMER.set_stopTime
```

A lot goes on in these few lines. First we define the Text node, then an automatically starting TimeSensor which cycles every second. But the bulk of the action takes place in the Script node. The cycleTime events from the TimeSensor enter the Script through eachSecond; the JavaScript function of the same name subtracts one from the counter field, which keeps track of the seconds remaining in the count.

The next line, countText[0] = counter, is a JavaScript feature we haven't covered before. The data type of countText is MFString, so the field can have multiple strings inside of it. How do we reach in and change a particular string? The strings are arranged in what programmers call an *array* of values. Arrays have implied index number, just like the

points in the Coordinate node, and, just like those points, they begin with index 0, rather than 1. So this JavaScript statement means “take the first string in countText and set to the value of counter”.

Once countText has the new value inside of it, it is assigned to newText, which sends the event to the Text node that changes the value of the string field.

Lastly, the value counter is tested – the double equals sign == compares the value of counter to 0; the comparison is true if the values match. When the values match, counter is empty, and the current time, passed into the Script by eachSecond, is emitted on stopTheTimer, which sets the stopTime value of the TimeSensor to a value greater than the startTime of the TimeSensor, so the countdown stops.

And yes, it does countdown.

Because the TimeSensor emits a cycleTime event as soon as it starts, we never see the countdown at 60 seconds; it’s already moved along to 59 seconds by the time it gets rendered to the display. That’s exactly as it should be.

There’s one thing we could add to this countdown timer – we could change the color of the numerals as the countdown nears its end. We could keep it green for the first 30 seconds, yellow for the next 20, and red for the last 10. This would all happen within the Script node; we’d need to add an eventOut to pass the SFColor to the Material node associated with the Text node. It might look like this:

```
      #VRML V2.0 utf8
      # This is the ninteenth example on scripting
      # Begin with the Shape of the Text
      Shape {
            appearance Appearance {
                  material DEF COUNTER_COLOR Material {
                        diffuseColor 0 1 0 # green
                  }
            }
            geometry DEF COUNTER_TEXT Text {
string [ "60" ] # starts at 60 seconds
            }
      }
      DEF TIMER TimeSensor {
            loop TRUE         # repeat every second
            cycleInterval 1   # one second per
            startTime 1       # automatic start
            stopTime 0        # automatic start
      }
      DEF COUNTDOWN Script {
            field SFInt32 counter 60            # maintains 60 seconds
            field MFString countText [ "60" ] # output string
            field MFColor textColors [ 1 1 0, 1 0 0 ] # the colors
            eventIn SFTime eachSecond     # get eventIn every second
            eventOut MFString newText     # send updated text
            eventOut SFTime stopTheTimer  # send stop timer
            eventOut SFColor newColor     # send updated color
```

```
            url [ "javascript:            // javascript comments
                        function eachSecond (es) {
                        counter = counter – 1; // one less second
                        countText[0] = counter;       // new value
                        newText = countText; // post the countdown
                        if (counter == 30 ) { // 30 seconds left?
                              newColor = textColors[0];
                        }
                        if (counter == 10) { // 10 seconds left?
                              newColor = textColors[1];
                        }
                        if (counter == 0) {     // at zero?
                              stopTheTimer = es;      // stop
TimeSensor
                        }
                  }
            " ] # End of JavaScript, back to VRML comments
      }
      # Route TimeSensor to Script
      ROUTE TIMER.cycleTime TO COUNTDOWN.eachSecond
      # Route Script to Text
      ROUTE COUNTDOWN.newText TO COUNTER_TEXT.set_string
      # Route Script to TimeSensor
      ROUTE COUNTDOWN.stopTheTimer TO TIMER.set_stopTime
      # Route Script to Material
      ROUTE COUNTDOWN.newColor TO COUNTER_COLOR.set_diffuseColor
```

Rather than creating two SFColor fields within the Script node, we create an array of SFColor values using the MFColor field named textColors. The first value – array index 0 – is yellow, while the second value – array index 1 – is red. Each second, we test to see if we've reached the 30 second countdown. If we do, we emit array value 0 from textColors. At 10 seconds, we emit array value 1 from textColors – turning the text red.

The Script node opens VRML up to almost endless interactivity – and the Text node allows you to watch it all in great detail, but it's often far better to hear something than to watch it – in fact, I can hear a growing roar of Sound in the distance…