Chapter 20: A Working Prototype

Picking Up the Pieces

We've learned enough VRML that we can begin to piece these lessons into larger projects - worlds which do something useful. In the next two chapters, we focus on examples which integrate geometry, sensors and scripts - the basic elements of every real-world VRML project - while we also examine the design process for each project. VRML worlds are complicated enough that they can't be built willy-nilly - you need to have a clear idea of where you're going when you set out; only then can you guarantee that you'll arrive at your goal.

The example for this chapter is known as *The Material Builder*. I originally designed it to help my students understand the relationship between the different fields in the Material node and their visible qualities. Growing tired of waving my hands trying to explain the difference between diffuseColor and emissiveColor, I built a tool that could demonstrate that difference clearly, without any need for wordy explanations.

The basic interface element in the Material Builder is the *slider*. Sliders are a very common element on the computer desktop; for example, your word processor probably has a scroll bar along the right side of the page, that's one form of a slider. The volume control is also likely implemented as a slider; move it one way, the sound grows louder, move it in the opposite direction, and it grows fainter. Sliders are intuitive because so many examples of them exist in the real world - dimmer controls on lights, on a stereo or an audio mixing board. People like them because they "feel" right, that is, they present a tactile, sensual interface to information.

Slip Sliding Away

The slider is composed of two pieces; the *thumb*, which the user drags upon, and the *post*, which constrains the movement of the thumb to a predefined path. We can use Box nodes for both of these pieces, to keep the polygon count low, and we can stretch and translate the boxes so that it looks as if the thumb is resting within the post. The basic slider might look something like this:

```
#VRML V2.0 utf8
# The first example on the Material Builder
# We want to Group the Post and Thumb together
Group {
      children [
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor 1 0 0 # make it red
                        }
                  }
```

```
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            DEF THUMB_XFORM Transform {
                  children [
                        # Define the shape of the Thumb
                        DEF THUMB Shape {
                              appearance Appearance {
                                    material DEF THUMBCOLOR Material {
                                          diffuseColor 0.5 0.5 0.5 #
gray
                                    }
                              }
                              geometry Box {
                                    size 4.95 2.5 2.5
                              }
                        }
                  ]
                  translation 0 0 1.0 # push it forward a bit
            }
      ] # end list of children in Group node
} # end group node
```

A lot of what we're doing here is preparing us for further down the road; we've given names to anything that might be useful, and we've placed all of the geometry into a Group node. This helps us keep it together as a unit - you'll see why that's important in just a little while. What we get is a red slider with a thumb in the center of its range.

The Plane Truth

To make the thumb mobile, we need to attach a sensor to it. We could use a TouchSensor, which sends an eventOut with a data type of SFVec3f as the mouse is dragged over the object. But there's another sensor better suited to this kind of the task. The PlaneSensor works much like the TouchSensor, but tracks only two dimensions - x and y - which makes it better suited for use with the slider, whose thumb only moves in one dimension (up/down). Here's the definition of the PlaneSensor:

```
PlaneSensor {           # Definition
      autoOffset  # SFBool, exposedField
      enabled           # SFBool, exposedField
      maxPosition # SFVec2f, exposedField
      minPosition # SFVec2f, exposedField
      offset            # SSFVec3f, exposedField
}
```

The PlaneSensor node has an eventOut named translation_changed, which sends out a stream of SFVec3f messages when the user drags their mouse over an object that's been attached to the sensor. We'll ROUTE these eventOut messages back into the Transform node which contains the thumb, but we'll have to put the PlaneSensor into a Group node along with the thumb - this attaches the PlaneSensor to the thumb:

```
#VRML V2.0 utf8
# The second example on the Material Builder
# We want to Group the Post and Thumb together
Group {
      children [
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor 1 0 0 # make it red
                        }
                  }
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            # Group the PlaneSensor and thumb together
            Group {
                  children [
                        DEF SENSOR PlaneSensor {
                              maxPosition 0 11.25 # top of range
                              minPosition 0 -11.25 # bottom of range
                              offset 0 0 1.5 # maintain thumb's z axis
                        }
                        DEF THUMB_XFORM Transform {
                              children [
                                    # Define the shape of the Thumb
                                    DEF THUMB Shape {
                                          appearance Appearance {
                                                material DEF THUMBCOLOR
Material {
                                                      diffuseColor 0.5
0.5 0.5 # gray
                                                }
                                          }
                                          geometry Box {
                                                size 4.95 2.5 2.5
                                          }
                                    }
                              ]
                              translation 0 0 1.0 # push it forward a
bit
                        }
                  ]
            }
      ] # end list of children in Group node
} # end group node
# Route the PlaneSensor to the thumb's Transform
ROUTE SENSOR.translation_changed TO THUMB_XFORM.set_translation
```

The PlaneSensor makes this kind of work easy. It has two fields, maxPosition and minPosition, which allow you to define a range for the PlaneSensor - when you move outside of that range, the PlaneSensor simply "pins" you to the maxPosition or minPositon values. In this example we set maxPosition to 0 11.25, because we don't want any motion in X, and we only want to go up to the top of the slider. (The post is

12.5 units above the origin, while the thumb is 2.5 units tall, so allowing the thumb to rise or fall a maximum of 11.25 units places the top or bottom of the thumb in alignment with the top or bottom of the post.)

The PlaneSensor has another field, autoOffset, which allows you to track an object as it gets dragged around; when TRUE, the PlaneSensor constantly updates its local coordinate space to match the new coordinate space of the dragged object. It defaults to TRUE, so we didn't bother to set it in the VRML file. We did set the value of the offset field to preserve the 1.5 unit translation in z that we give thumb inside of its Transform node; if we didn't do this, the PlaneSensor would emit a z value of 0 on translation_changed, and put the thumb back into the post.

Finally, we ROUTE the eventOut translation_changed from the PlaneSensor into the Transform node containing the thumb. Once that ROUTE is made, the thumb tracks mouse movement, and stays within the range of the slider.

You Don't Say?

What does a slider do? A slider allows you to select a *scalar* value, that is, a single number. Scalar values in VRML are represented as SFFloat data types, so somehow we've got to translate from a position of a thumb and a post into a number - this sounds like a job for the Script node. We can take the same translation_changed eventOut value and send it into the Script. (Wiring an eventOut to multiple eventIn is known as *fan out*; you can have unlimited fan out in VRML.)

Once we've gotten the value within the Script node, we can do a little math, calculate the scalar value of the slider, and provide that on an eventOut from the Script. What's more (and really quite helpful) we'll also create an MFString eventOut, which contains a text string with the scalar value. We'll wire that up to a Text node which we place onto the thumb, and presto! - the thumb now tells us what its current value is. Let's see how all of that might look:

```
#VRML V2.0 utf8
# The third example on the Material Builder
# We want to Group the Post and Thumb together
Group {
      children [
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor 1 0 0 # make it red
                        }
                  }
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            # Group the PlaneSensor and thumb together
            Group {
```

```
                  children [
                        DEF SENSOR PlaneSensor {
                              maxPosition 0 11.25 # top of range
                              minPosition 0 -11.25 # bottom of range
                              offset 0 0 1.5 # maintain thumb’s z axis
                        }
                        DEF THUMB_XFORM Transform {
                              children [
                                    # Define the shape of the Thumb
                                    DEF THUMB Shape {
                                          appearance Appearance {
                                                material DEF THUMBCOLOR
Material {
                                                      diffuseColor 0.5
0.5 0.5 # gray
                                                }
                                          }
                                          geometry Box {
                                                size 4.95 2.5 2.5
                                          }
                                    }
                                    # Place text slightly in front of
Thumb
                                    Transform {
                                          children [
                                                # Define some Text on
the Thumb
                                                Shape {
                                                      appearance
Appearance {
                                                            material
Material {

      diffuseColor 1 1 1
                                                            }
                                                      }
                                                      geometry DEF
TEXT_VALUE Text {
                                                            string [ ""
]
                                                            fontStyle
FontStyle {
                                                                  size
1.3
                                                            }
                                                      }
                                                }
                                          ]
                                          translation -2.1 -0.2 1.5 #
just in front
                                    }
                              ]
                              translation 0 0 1.0 # push it forward a
bit
                        }
                  ]
            }
```

```
      ] # end list of children in Group node
} # end group node
DEF MAKE_SCALAR Script {
eventIn SFVec3f whereIsIt
eventOut SFFloat theScalar
eventOut MFString scalarString
field MFString myString [ “0” ]
field SFFloat aScalar 0
url [ “javascript:            // begin the dirty work
            function whereIsIt(where) {
                  aScalar = where.y;      // just want the y portion
            aScalar = aScalar + 11.25; // correct for negative values
                  aScalar = aScalar / 22.5; // make value between zero
and one
                  theScalar = aScalar;    // send it out
            myString[0] = aScalar;  // make a string of it
                  scalarString = myString;      // and send that out,
too
      } “
]
}
# Route the PlaneSensor to the thumb’s Transform
ROUTE SENSOR.translation_changed TO THUMB_XFORM.set_translation
# Route the PlaneSensor to the Script, this is fan out of an eventOut
ROUTE SENSOR.translation_changed TO MAKE_SCALAR.whereIsIt
# Route the Script to the Text node
ROUTE MAKE_SCALAR.scalarString TO TEXT_VALUE.set_string
```

We’ve added a Text node inside of a Transform node, so that the text is placed in front of the thumb. The Script is pretty simple; one eventIn, whereIsIt receives the translation_changed value from the PlaneSensor. Within the JavaScript routine whereIsIt, the y portion of the translation is extracted, adjusted to ensure that it is zero or greater, than divided by the range of the thumb (22.5 units) to give a scalar range between zero and one. That value is sent out on the eventOut named theScalar, and is converted to a string in the MFString array myString, then that is emitted on the eventOut named scalarString. Although we haven’t routed theScalar to any eventIn, we know its current value - written on the thumb, in big white letters.

There’s only one more thing we need to be doing; we need to set up the initial value for the slider. A good initial value for the slider would be 0.0; of course, we’d want the Text to say that as well. But - as you may have already noticed - the Text doesn’t appear on the thumb until you begin to drag the thumb around; you haven’t generated any translation_changed events, so the JavaScript routine which sets the Text hasn’t been called. Fortunately, the Script node allows you to create a JavaScript function named initialize, which is automatically executed as soon as the VRML world is loaded. If we set the translation for the thumb to the bottom of its range, and setup an initialize function which sets the Text field string to 0.0, we should get the desired effect:

```
#VRML V2.0 utf8
# The fourth example on the Material Builder
# We want to Group the Post and Thumb together
Group {
      children [
```

```
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor 1 0 0 # make it red
                        }
                  }
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            # Group the PlaneSensor and thumb together
            Group {
                  children [
                        DEF SENSOR PlaneSensor {
                              maxPosition 0 11.25 # top of range
                              minPosition 0 -11.25 # bottom of range
                              offset 0 -11.25 1.5 # maintain thumb's z
axis
                        }
                        DEF THUMB_XFORM Transform {
                              children [
                                    # Define the shape of the Thumb
                                    DEF THUMB Shape {
                                          appearance Appearance {
                                                material DEF THUMBCOLOR
Material {
                                                      diffuseColor 0.5
0.5 0.5 # gray
                                                }
                                          }
                                          geometry Box {
                                                size 4.95 2.5 2.5
                                          }
                                    }
                                    # Place text slightly in front of
Thumb
                                    Transform {
                                          children [
                                                # Define some Text on
the Thumb
                                                Shape {
                                                      appearance
Appearance {
                                                            material
Material {

      diffuseColor 1 1 1
                                                            }
                                                }
                                                geometry DEF
TEXT_VALUE Text {
                                                      string [ ""
]
                                                      fontStyle
FontStyle {
```

```
                                                                  size
1.3
                                                                  }
                                                              }
                                                          }
                                                  ]
                                                  translation -2.1 -0.2 1.5 #
just in front
                                          }
                                  ]
                                  translation 0 -11.25 1.0 # bottom of
range & forward
                          }
                  ]
          }
      ] # end list of children in Group node
} # end group node
DEF MAKE_SCALAR Script {
eventIn SFVec3f whereIsIt
eventOut SFFloat theScalar
eventOut MFString scalarString
field MFString myString [ “0.0” ]  # initial value for Text
field SFFloat aScalar 0
url [ “javascript:            // begin the dirty work
            function whereIsIt(where) {
                  aScalar = where.y;      // just want the y portion
            aScalar = aScalar + 11.25; // correct for negative values
                  aScalar = aScalar / 22.5; // make value between zero
and one
                  theScalar = aScalar;    // send it out
            myString[0] = aScalar;  // make a string of it
                  scalarString = myString;      // and send that out,
too
      }
      // This function is called when the Script is loaded
      function initialize() {
            scalarString = myString;      // set Text value
      } “
]
}
# Route the PlaneSensor to the thumb’s Transform
ROUTE SENSOR.translation_changed TO THUMB_XFORM.set_translation
# Route the PlaneSensor to the Script, this is fan out of an eventOut
ROUTE SENSOR.translation_changed TO MAKE_SCALAR.whereIsIt
# Route the Script to the Text node
ROUTE MAKE_SCALAR.scalarString TO TEXT_VALUE.set_string
```

We don’t need to add any ROUTE statements - everything’s already in place. All we’re doing is adding a JavaScript initialize function which sets everything up as we load the world. Now we see the thumb at the bottom of its range, and displaying its initial value.

Why Type When You Can Prototype?

In just about a hundred lines of VRML, we’ve created a wonderfully useful widget - so useful, in fact, that we’d want to use it almost everywhere. Imagine using this slider to

control the cycleInterval value on a TimeSensor, the intensity on a SpotLight, or the pitch of an AudioClip. But here we begin to face a bit of a problem. We've named all of these components, and if we wanted to create another slider we'd have to add another hundred lines of VRML, using different names for the components. We'd need to do this for every slider - so if we wanted to use a dozen (as we soon will) we'd have 1200 lines of code that's doing more or less the same thing twelve times.

Fortunately, VRML allows you to avoid all of this typing and renaming with its PROTO capability. The PROTO allows you to create faux VRML nodes; they look like they're nodes as defined in the language, but you get to define what they are, how they operate, and how they talk to the rest of the world. You can have as much or as little complexity as you like hidden beneath the cover a PROTO - everything inside the PROTO becomes *opaque*, meaning it can't be seen by the VRML world. You can't see into a VRML node - with the exception of its exposedField entries; the same thing is true with a PROTO.

A PROTO is created in two pieces; the first part, in brackets, defines all of the interfaces to the PROTO - that is, whatever is *exposed* to the VRML world. This can include field, exposedField, eventIn and eventOut definitions of any kind, in any number. Then, within a set of braces, the internal definition of the PROTO describes the visible and interactive characteristics of the PROTO. This includes all geometry, sensors, scripts and routes; all of them can live - invisibly - within a PROTO.

Let's create a PROTO for our slider, so we can make real widget of it. First, we need to think about what inputs the slider might have. It doesn't really need to have any eventIn interfaces, but we can also expose some of the field values internal to the PROTO. We should expose an SFColor exposedField which controls the color of the post of the slider; that way, we can *instance* (a fancy way of saying "create") sliders with posts of any color we might desire. The slider will have one eventOut, which is the scalar value with a data type of SFFloat. The interface for the PROTO might look like this:

```
PROTO Slider # PROTO keyword, then given name
[
      exposedField SFColor postColor 1 1 1 # defaults white
      eventOut SFFloat sliderScalar
]
```

The PROTO keyword is followed by the given name for the PROTO; this follows the convention of nodes, which start with an upper-case letter. This name is all that's required to instance a copy of our slider, just like any other VRML node. The bracket opens the PROTO interface, and the next line creates an exposedField, followed by an eventOut. The interface concludes with the closing bracket.

The body of the PROTO is almost exactly a carbon copy of our slider, all gathered within an outer set of braces. However, there are two very important exceptions:

```
{
# We want to Group the Post and Thumb together
Group {
      children [
```

```
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor IS postColor # connects to
interface
                        }
                  }
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            # Group the PlaneSensor and thumb together
            Group {
                  children [
                        DEF SENSOR PlaneSensor {
                              maxPosition 0 11.25 # top of range
                              minPosition 0 -11.25 # bottom of range
                              offset 0 -11.25 1.5 # maintain thumb's z
axis
                        }
                        DEF THUMB_XFORM Transform {
                              children [
                                    # Define the shape of the Thumb
                                    DEF THUMB Shape {
                                          appearance Appearance {
                                                material DEF THUMBCOLOR
Material {
                                                      diffuseColor 0.5
0.5 0.5
                                                }
                                          }
                                          geometry Box {
                                                size 4.95 2.5 2.5
                                          }
                                    }
                                    # Place text slightly in front of
Thumb
                                    Transform {
                                          children [
                                                # Define some Text on
the Thumb
                                                Shape {
                                                      appearance
Appearance {
                                                            material
Material {

diffuseColor 1 1 1
                                                            }
                                                      }
                                                   geometry DEF
TEXT_VALUE Text {
                                                            string [ ""
]
                                                            fontStyle
FontStyle {
```

```
                                                                  size
1.3
                                                                  }
                                                              }
                                                          }
                                                  ]
                                                  translation -2.1 -0.2 1.5 #
just in front
                                          }
                                  ]
                                  translation 0 -11.25 1.0 # bottom of
range & forward
                          }
                  ]
            }
      ] # end list of children in Group node
} # end group node
DEF MAKE_SCALAR Script {
eventIn SFVec3f whereIsIt
eventOut SFFloat theScalar IS sliderScalar # connects to interface
eventOut MFString scalarString
field MFString myString [ “0.0” ]  # initial value for Text
field SFFloat aScalar 0
url [ “javascript:            // begin the dirty work
            function whereIsIt(where) {
                  aScalar = where.y;      // just want the y portion
            aScalar = aScalar + 11.25; // correct for negative values
                  aScalar = aScalar / 22.5; // make value between zero
and one
                  theScalar = aScalar;    // send it out
            myString[0] = aScalar;  // make a string of it
                  scalarString = myString;      // and send that out,
too
      }
      // This function is called when the Script is loaded
      function initialize() {
            scalarString = myString;      // set Text value
      } “
]
}
# Route the PlaneSensor to the thumb’s Transform
ROUTE SENSOR.translation_changed TO THUMB_XFORM.set_translation
# Route the PlaneSensor to the Script, this is fan out of an eventOut
ROUTE SENSOR.translation_changed TO MAKE_SCALAR.whereIsIt
# Route the Script to the Text node
ROUTE MAKE_SCALAR.scalarString TO TEXT_VALUE.set_string
}
```

If you look at the Material node for the post, you’ll see the line diffuseColor IS postColor - this is the magic of the PROTO. With that line, the exposed interface postColor is connected to the opaque internal field diffuseColor, and any changes to postColor, which is an exposedField, will effect the diffuseColor field. The same thing happens in the Script node, where you’ll see theScalar IS sliderScalar - connecting the eventOut of the Script to the interface of the PROTO.

None of this actually creates anything; if we put this into a VRML file, and loaded it into a browser, we'd see nothing at all, because we haven't instanced any copies of the slider. However, all we need to do is add one very simple line, and a Slider is created. Here's how it all looks, put together:

```
#VRML V2.0 utf8
# The fifth example on the Material Builder
# PROTO definitions should always come early in a VRML world
PROTO Slider # PROTO keyword, then given name
[                                       # exposed interfaces
      exposedField SFColor postColor 1 1 1 # defaults white
      eventOut SFFloat sliderScalar
]
{                                       # opaque interior of Slider
# We want to Group the Post and Thumb together
Group {
      children [
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor IS postColor # connects to
interface
                        }
                  }
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            # Group the PlaneSensor and thumb together
            Group {
                  children [
                        DEF SENSOR PlaneSensor {
                              maxPosition 0 11.25 # top of range
                              minPosition 0 -11.25 # bottom of range
                              offset 0 -11.25 1.5 # maintain thumb's z
axis
                        }
                        DEF THUMB_XFORM Transform {
                              children [
                                    # Define the shape of the Thumb
                                    DEF THUMB Shape {
                                          appearance Appearance {
                                                material DEF THUMBCOLOR
Material {
                                                      diffuseColor 0.5
0.5 0.5
                                                }
                                          }
                                          geometry Box {
                                                size 4.95 2.5 2.5
                                          }
                                    }
                                    # Place text slightly in front of
Thumb
                                    Transform {
```

```
                                          children [
                                                # Define some Text on
the Thumb
                                                Shape {
                                                      appearance
Appearance {
                                                            material
Material {

diffuseColor 1 1 1
                                                            }
                                                      }
                                                   geometry DEF
TEXT_VALUE Text {
                                                            string [ ""
]
                                                            fontStyle
FontStyle {
                                                                  size
1.3
                                                            }
                                                      }
                                                }
                                          ]
                                          translation -2.1 -0.2 1.5 #
just in front
                                    }
                              ]
                              translation 0 -11.25 1.0 # bottom of
range & forward
                        }
                  ]
            }
      ] # end list of children in Group node
} # end group node
DEF MAKE_SCALAR Script {
eventIn SFVec3f whereIsIt
eventOut SFFloat theScalar IS sliderScalar # connects to interface
eventOut MFString scalarString
field MFString myString [ “0.0” ]  # initial value for Text
field SFFloat aScalar 0
url [ “javascript:            // begin the dirty work
            function whereIsIt(where) {
                  aScalar = where.y;      // just want the y portion
            aScalar = aScalar + 11.25; // correct for negative values
                  aScalar = aScalar / 22.5; // make value between zero
and one
                  theScalar = aScalar;    // send it out
            myString[0] = aScalar;  // make a string of it
                  scalarString = myString;      // and send that out,
too
      }
      // This function is called when the Script is loaded
      function initialize() {
            scalarString = myString;      // set Text value
      } “
]
```

```
}
# Route the PlaneSensor to the thumb's Transform
ROUTE SENSOR.translation_changed TO THUMB_XFORM.set_translation
# Route the PlaneSensor to the Script, this is fan out of an eventOut
ROUTE SENSOR.translation_changed TO MAKE_SCALAR.whereIsIt
# Route the Script to the Text node
ROUTE MAKE_SCALAR.scalarString TO TEXT_VALUE.set_string
}
# Now we create an instance of the PROTO, with a green post
Slider { postColor 0 1 0 }          # That's all it takes!
```

The very last line does all of the work in this VRML world; everything that comes before it defines a PROTO node named Slider, but only the last line creates the green slider.

Three's a Crowd

Any PROTO can be instanced any number of times. Here's a fragment of code which creates three sliders - red, green, blue - all side by side.

```
#VRML V2.0 utf8
# The sixth example on the Material Builder
# We'll skip the PROTO definition for Slider, but it's here…
. . . . . .
# Now we instance three Slider nodes, in two Transform nodes
# The first slider has a red post
DEF RED_SLIDER Slider { postColor 1 0 0 }
# Transform with two Slider nodes inside
Transform {
      children [
            # Second slider has green post
            DEF GREEN_SLIDER Slider { postColor 0 1 0 }
            # Transform with one slider inside
            Transform {
                  children [
                        # Third slider has blue post
                        DEF BLUE_SLIDER Slider { postColor 0 0 1 }
                  ]
                  translation 5 0 0 # side-by-side
            }
      ]
      translation 5 0 0 # side-by-side
}
```

We place the Slider nodes next to each other with the Transform nodes, so we get a solid block of colors with three thumbs lined up, side-by-side.

All Routed Up and Nowhere to Go

We've got sliders now - everywhere. We should begin to harness their strengths, to ROUTE the scalar values they create into something. Since these sliders will be controlling color values within a Material node, perhaps we can use the Slider to control the intensity of the post color of the Slider - a nice form of visual feedback. We'd need to

create a Script node with three eventIn inputs - one from each slider - which can take in the scalar values, convert them to SFColor values, then send them back to the Slider, via its exposedField postColor. Here's how that might look:

```
#VRML V2.0 utf8
# The seventh example on the Material Builder
# We'll skip the PROTO definition for Slider, but it's here…
. . . . . .
# Now we instance three Slider nodes, in two Transform nodes
# The first slider has a red post
DEF RED_SLIDER Slider { postColor 0 0 0 } # no red at start
# Transform with two Slider nodes inside
Transform {
      children [
            # Second slider has green post
            DEF GREEN_SLIDER Slider { postColor 0 0 0 } # no green
start
            # Transform with one slider inside
            Transform {
                  children [
                        # Third slider has blue post
                        DEF BLUE_SLIDER Slider { postColor 0 0 0 } # no
blue
                  ]
                  translation 5 0 0 # side-by-side
            }
      ]
      translation 5 0 0 # side-by-side
}
DEF COLORIST Script {
      eventIn SFFloat redValue            # from Red slider
      eventIn SFFloat greenValue    # from Green slider
      eventIn SFFloat blueValue     # from Blue slider
      eventOut SFColor newRedColor  # output color
      eventOut SFColor newGreenColor # output color
      eventOut SFColor newBlueColor  # output color
      field SFColor localColor 0 0 0 # temp. storage
      url [ "javascript:
            function redValue (rv) {      // on Red slider events
                  localColor.r = rv;      // use the slider value for
red
                  localColor.g = 0; // no green
                  localColor.b = 0; // no blue
                  newRedColor = localColor;     // send color out
            }
            function greenValue (gv) { // on Green slider events
                  localColor.r = 0; // no red
                  localColor.g = gv;      // use slider value for green
                  localColor.b = 0; // no blue
                  newGreenColor = localColor; // send color out
            }
            function blueValue (bv) { // on Blue slider events
                  localColor.r = 0; // no red
                  localColor.g = 0; // no green
                  localColor.b = bv; // use slider value for blue
                  newBlueColor = localColor; // send value out
```

```
            } “
      ]
}
# ROUTE from Red Slider to Script
ROUTE RED_SLIDER.sliderScalar TO COLORIST.redValue
# ROUTE from Script back to Red Slider
ROUTE COLORIST.newRedColor TO RED_SLIDER.set_postColor
# ROUTE from Green Slider to Script
ROUTE GREEN_SLIDER.sliderScalar TO COLORIST.greenValue
# ROUTE from Script back to Green Slider
ROUTE COLORIST.newGreenColor TO GREEN_SLIDER.set_postColor
# ROUTE from Blue Slider to Script
ROUTE BLUE_SLIDER.sliderScalar TO COLORIST.blueValue
# ROUTE from Script back to Blue Slider
ROUTE COLORIST.newBlueColor TO BLUE_SLIDER.set_postColor
```

Each Slider has its eventOut routed into the Script node, which takes the scalar value, places it into an SFColor data type, then sends emits that data type through a corresponding eventOut. The result, when we enter the world, is three thumbs floating in black space.

However, as we manipulate the thumbs, their posts begin to glow with color.

Going All the Way

It’s time to put the entire Material Builder together. There are six exposedField entries in the Material node; diffuseColor, emissiveColor, specularColor, shininess, transparency and ambientIntensity. The first three have SFColor data types, while the last three are SFFloat values - scalars. That’s a total of twelve sliders required, in four groups of three. We’ll create a Sphere to the right of the sliders, which will display the Material as it is being created. Each of the Slider instances will need its own handling within a Script, and its own ROUTE statements, so this example will be a little bigger than we’ve seen before. It’s presented here in its entirety:

```
#VRML V2.0 utf8
# The eighth example on the Material Builder
# PROTO definitions should always come early in a VRML world
PROTO Slider # PROTO keyword, then given name
[                                          # exposed interfaces
      exposedField SFColor postColor 1 1 1 # defaults white
      eventOut SFFloat sliderScalar
]
{                                          # opaque interior of Slider
# We want to Group the Post and Thumb together
Group {
      children [
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor IS postColor # connects to
interface
                        }
```

```
                  }
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            # Group the PlaneSensor and thumb together
            Group {
                  children [
                        DEF SENSOR PlaneSensor {
                              maxPosition 0 11.25 # top of range
                              minPosition 0 -11.25 # bottom of range
                              offset 0 -11.25 1.5 # maintain thumb's z
axis
                        }
                        DEF THUMB_XFORM Transform {
                              children [
                                    # Define the shape of the Thumb
                                    DEF THUMB Shape {
                                          appearance Appearance {
                                                material DEF THUMBCOLOR
Material {
                                                      diffuseColor 0.5
0.5 0.5
                                                }
                                          }
                                          geometry Box {
                                                size 4.95 2.5 2.5
                                          }
                                    }
                                    # Place text slightly in front of
Thumb
                                    Transform {
                                          children [
                                                # Define some Text on
the Thumb
                                                Shape {
                                                      appearance
Appearance {
                                                            material
Material {
diffuseColor 1 1 1
                                                            }
                                                      }
                                                   geometry DEF
TEXT_VALUE Text {
                                                            string [ ""
]
                                                            fontStyle
FontStyle {
                                                                  size
1.3
                                                            }
                                                      }
                                          ]
```

```
                                          translation -2.1 -0.2 1.5 #
just in front
                                    }
                              ]
                              translation 0 -11.25 1.0 # bottom of
range & forward
                        }
                  ]
            }
      ] # end list of children in Group node
} # end group node
DEF MAKE_SCALAR Script {
eventIn SFVec3f whereIsIt
eventOut SFFloat theScalar IS sliderScalar # connects to interface
eventOut MFString scalarString
field MFString myString [ “0.0” ]  # initial value for Text
field SFFloat aScalar 0
url [ “javascript:            // begin the dirty work
            function whereIsIt(where) {
                  aScalar = where.y;      // just want the y portion
            aScalar = aScalar + 11.25; // correct for negative values
                  aScalar = aScalar / 22.5; // make value between zero
and one
                  theScalar = aScalar;    // send it out
            myString[0] = aScalar;  // make a string of it
                  scalarString = myString;      // and send that out,
too
      }
      // This function is called when the Script is loaded
      function initialize() {
            scalarString = myString;      // set Text value
      } “
]
}
# Route the PlaneSensor to the thumb’s Transform
ROUTE SENSOR.translation_changed TO THUMB_XFORM.set_translation
# Route the PlaneSensor to the Script, this is fan out of an eventOut
ROUTE SENSOR.translation_changed TO MAKE_SCALAR.whereIsIt
# Route the Script to the Text node
ROUTE MAKE_SCALAR.scalarString TO TEXT_VALUE.set_string
}
# Now we instance three Slider nodes, in two Transform nodes, for
diffuseColor
# The first slider has a red post
DEF RED_DIFFUSE Slider { postColor 1 0 0 }
# Transform with two Slider nodes inside
Transform {
      children [
            # Second slider has green post
            DEF GREEN_DIFFUSE Slider { postColor 0 1 0 }
            # Transform with one slider inside
            Transform {
                  children [
                        # Third slider has blue post
                        DEF BLUE_DIFFUSE Slider { postColor 0 0 1 }
                  ]
                  translation 5 0 0 # side-by-side
```

```
            }
      ]
      translation 5 0 0 # side-by-side
}
# Now we instance three Slider nodes, in three Transform nodes, for
emissiveColor
Transform {
      children [
# The first slider has a red post
DEF RED_EMISSIVE Slider { postColor 1 0 0 }
# Transform with two Slider nodes inside
Transform {
                  children [
                        # Second slider has green post
                        DEF GREEN_EMISSIVE Slider { postColor 0 1 0 }
                        # Transform with one slider inside
                        Transform {
                              children [
                                    # Third slider has blue post
                                    DEF BLUE_EMISSIVE Slider {
postColor 0 0 1 }
                              ]
                              translation 5 0 0 # side-by-side
                        }
                  ]
                  translation 5 0 0 # side-by-side
}
      ]
      translation 25 0 0 # near diffuseColor sliders
}

# Now we instance three Slider nodes, in three Transform nodes, for
specularColor
Transform {
      children [
# The first slider has a red post
DEF RED_SPECULAR Slider { postColor 1 0 0 }
# Transform with two Slider nodes inside
Transform {
                  children [
                        # Second slider has green post
                        DEF GREEN_SPECULAR Slider { postColor 0 1 0 }
                        # Transform with one slider inside
                        Transform {
                              children [
                                    # Third slider has blue post
                                    DEF BLUE_SPECULAR Slider {
postColor 0 0 1 }
                              ]
                              translation 5 0 0 # side-by-side
                        }
                  ]
                  translation 5 0 0 # side-by-side
}
      ]
      translation 0 -30 0 # underneath diffuseColor sliders
}
```

```
# Three final Slider nodes, in three Transform nodes, for shininess,
transparency, ambient
Transform {
      children [
# The first slider has a yellow post
DEF SHINY Slider { postColor 1 1 0 }
# Transform with two Slider nodes inside
Transform {
                  children [
                        # Second slider has a mauve post
                        DEF TRANSPARENT Slider { postColor 1 0 1 }
                        # Transform with one slider inside
                        Transform {
                              children [
                                    # Third slider has white post
                                    DEF AMBIENT Slider { postColor 1 1
1 }
                              ]
                              translation 5 0 0 # side-by-side
                        }
                  ]
                  translation 5 0 0 # side-by-side
}
      ]
      translation 25 -30 0 # underneath emissiveColor sliders
}
# Now add a Sphere which has a Material node being manipulated by all
of this.
Transform {
      children [
            Shape {
                  appearance Appearance {
                        material DEF BUILDER Material {
                              diffuseColor 0 0 0      # everything off
to start
                              emissiveColor 0 0 0 # off
                              specularColor 0 0 0 # off
                              shininess 0
                              transparency 0
                              ambientIntensity 0
                        }
                  }
                  geometry Sphere { radius 10 }
            }
      ]
      translation 60 -15 0 # center the Sphere on the right
}
# Now the Script which receives the Slider events and adjusts the
Material
DEF MAKER Script {
      eventIn SFFloat redDiffuse          # one eventIn per slider
      eventIn SFFloat greenDiffuse
      eventIn SFFloat blueDiffuse
      eventIn SFFloat redEmissive
      eventIn SFFloat greenEmissive
      eventIn SFFloat blueEmissive
      eventIn SFFloat redSpecular
```

```
      eventIn SFFloat greenSpecular
      eventIn SFFloat blueSpecular
      field SFColor localDiffuse 0 0 0          # local storage
      field SFColor localEmissive 0 0 0
      field SFColor localSpecular 0 0 0
      eventOut SFColor newDiffuse         # one eventOut per
exposedField
      eventOut SFColor newEmissive
      eventOut SFColor newSpecular
      url [ “javascript:
            // Three functions for the diffuse colors
            function redDiffuse (rd) {
                  localDiffuse.r = rd;    // Record color component
                  newDiffuse = localDiffuse;    // send it out
            }
            function greenDiffuse (gd) {
                  localDiffuse.g = gd;    // Record color component
                  newDiffuse = localDiffuse;    // send it out
            }
            function blueDiffuse (bd) {
                  localDiffuse.b = bd;    // Record color component
                  newDiffuse = localDiffuse;    // send it out
            }
            // Three functions for the emissive colors
            function redEmissive (re) {
                  localEmissive.r = re;
                  newEmissive = localEmissive;
            }
            function greenEmissive (ge) {
                  localEmissive.g = ge;
                  newEmissive = localEmissive;
            }
            function blueEmissive (be) {
                  localEmissive.b = be;
                  newEmissive = localEmissive;
            }
            // Three functions for the specular colors
            function redSpecular (rs) {
                  localSpecular.r = rs;
                  newSpecular = localSpecular;
            }
            function greenSpecular (gs) {
                  localSpecular.g = gs;
                  newSpecular = localSpecular;
            }
            function blueSpecular (bs) {
                  localSpecular.b = bs;
                  newSpecular = localSpecular;
            } “
      ]
}
# Now we have ROUTE statements for days.
# We need one for each eventIn and eventOut in the Script
# ROUTE from the diffuse sliders to the Script, and from the Script to
Material
ROUTE RED_DIFFUSE.sliderScalar TO MAKER.redDiffuse
ROUTE GREEN_DIFFUSE.sliderScalar TO MAKER.greenDiffuse
```

```
ROUTE BLUE_DIFFUSE.sliderScalar TO MAKER.blueDiffuse
ROUTE MAKER.newDiffuse TO BUILDER.set_diffuseColor
# ROUTE from the emissive sliders to the Script, and from the Script to
Material
ROUTE RED_EMISSIVE.sliderScalar TO MAKER.redEmissive
ROUTE GREEN_EMISSIVE.sliderScalar TO MAKER.greenEmissive
ROUTE BLUE_EMISSIVE.sliderScalar TO MAKER.blueEmissive
ROUTE MAKER.newEmissive TO BUILDER.set_emissiveColor
# ROUTE from the specular sliders to the Script, and from the Script to
Material
ROUTE RED_SPECULAR.sliderScalar TO MAKER.redSpecular
ROUTE GREEN_SPECULAR.sliderScalar TO MAKER.greenSpecular
ROUTE BLUE_SPECULAR.sliderScalar TO MAKER.blueSpecular
ROUTE MAKER.newSpecular TO BUILDER.set_specularColor
# The scalar fields of shininess, transparency and ambientIntensity
# Can be sent straight in - no Script node processing required!
ROUTE SHINY.sliderScalar TO BUILDER.set_shininess
ROUTE TRANSPARENT.sliderScalar TO BUILDER.set_transparency
ROUTE AMBIENT.sliderScalar TO BUILDER.set_ambientIntensity
```

We need to ROUTE all of the Slider nodes which manipulate SFColor data types through the Script node, so the scalar values can be converted to SFColor values; however, shininess, transparency and ambientIntensity are scalar values, and were routed directly from the Slider to the correct exposedField. What you get is a four sets of three sliders, with no other visible objects.

But, once you begin to play with the sliders, you can see the Sphere appear to the right.

Let's Go Crazy

That's just the beginning of the Material Builder. Next, we should make the Sphere rotate - so that highlights become more visible. We should add a SpotLight pointing down onto the Sphere. We should even be able to move that SpotLight around a bit:

```
#VRML V2.0 utf8
# The ninth example on the Material Builder
# PROTO definitions should always come early in a VRML world
PROTO Slider # PROTO keyword, then given name
[                                          # exposed interfaces
      exposedField SFColor postColor 1 1 1 # defaults white
      eventOut SFFloat sliderScalar
]
{                                          # opaque interior of Slider
# We want to Group the Post and Thumb together
Group {
      children [
            # Define the shape of the post
            DEF POST Shape {
                  appearance Appearance {
                        material DEF POSTCOLOR Material {
                              diffuseColor IS postColor # connects to
interface
                        }
                  }
```

```
                  geometry Box {
                        size 5 25 2.5
                  }
            }
            # Group the PlaneSensor and thumb together
            Group {
                  children [
                        DEF SENSOR PlaneSensor {
                              maxPosition 0 11.25 # top of range
                              minPosition 0 -11.25 # bottom of range
                              offset 0 -11.25 1.5 # maintain thumb's z
axis
                        }
                        DEF THUMB_XFORM Transform {
                              children [
                                    # Define the shape of the Thumb
                                    DEF THUMB Shape {
                                          appearance Appearance {
                                                material DEF THUMBCOLOR
Material {
                                                      diffuseColor 0.5
0.5 0.5
                                                }
                                          }
                                          geometry Box {
                                                size 4.95 2.5 2.5
                                          }
                                    }
                                    # Place text slightly in front of
Thumb
                                    Transform {
                                          children [
                                                # Define some Text on
the Thumb
                                                Shape {
                                                      appearance
Appearance {
                                                            material
Material {
diffuseColor 1 1 1
                                                            }
                                                      }
                                                   geometry DEF
TEXT_VALUE Text {
                                                            string [ ""
]
                                                            fontStyle
FontStyle {
                                                                  size
1.3
                                                            }
                                                      }
                                          ]
                                          translation -2.1 -0.2 1.5 #
just in front
```

```
                                    }
                              ]
                              translation 0 -11.25 1.0 # bottom of
range & forward
                        }
                  ]
            }
      ] # end list of children in Group node
} # end group node
DEF MAKE_SCALAR Script {
eventIn SFVec3f whereIsIt
eventOut SFFloat theScalar IS sliderScalar # connects to interface
eventOut MFString scalarString
field MFString myString [ "0.0" ]  # initial value for Text
field SFFloat aScalar 0
url [ "javascript:            // begin the dirty work
            function whereIsIt(where) {
                  aScalar = where.y;      // just want the y portion
            aScalar = aScalar + 11.25; // correct for negative values
                  aScalar = aScalar / 22.5; // make value between zero
and one
                  theScalar = aScalar;    // send it out
            myString[0] = aScalar;  // make a string of it
                  scalarString = myString;      // and send that out,
too
      }
      // This function is called when the Script is loaded
      function initialize() {
            scalarString = myString;      // set Text value
      } "
]
}
# Route the PlaneSensor to the thumb's Transform
ROUTE SENSOR.translation_changed TO THUMB_XFORM.set_translation
# Route the PlaneSensor to the Script, this is fan out of an eventOut
ROUTE SENSOR.translation_changed TO MAKE_SCALAR.whereIsIt
# Route the Script to the Text node
ROUTE MAKE_SCALAR.scalarString TO TEXT_VALUE.set_string
}
NavigationInfo {
      headlight FALSE  # Use scene lighting
}
# Now we instance three Slider nodes, in two Transform nodes, for
diffuseColor
# The first slider has a red post
DEF RED_DIFFUSE Slider { postColor 1 0 0 }
# Transform with two Slider nodes inside
Transform {
      children [
            # Second slider has green post
            DEF GREEN_DIFFUSE Slider { postColor 0 1 0 }
            # Transform with one slider inside
            Transform {
                  children [
                        # Third slider has blue post
                        DEF BLUE_DIFFUSE Slider { postColor 0 0 1 }
                  ]
```

```
                translation 5 0 0 # side-by-side
            }
      ]
      translation 5 0 0 # side-by-side
}
# Now we instance three Slider nodes, in three Transform nodes, for
emissiveColor
Transform {
      children [
# The first slider has a red post
DEF RED_EMISSIVE Slider { postColor 1 0 0 }
# Transform with two Slider nodes inside
Transform {
                children [
                      # Second slider has green post
                      DEF GREEN_EMISSIVE Slider { postColor 0 1 0 }
                      # Transform with one slider inside
                      Transform {
                            children [
                                  # Third slider has blue post
                                  DEF BLUE_EMISSIVE Slider {
postColor 0 0 1 }
                            ]
                            translation 5 0 0 # side-by-side
                      }
                ]
                translation 5 0 0 # side-by-side
}
      ]
      translation 25 0 0 # near diffuseColor sliders
}

# Now we instance three Slider nodes, in three Transform nodes, for
specularColor
Transform {
      children [
# The first slider has a red post
DEF RED_SPECULAR Slider { postColor 1 0 0 }
# Transform with two Slider nodes inside
Transform {
                children [
                      # Second slider has green post
                      DEF GREEN_SPECULAR Slider { postColor 0 1 0 }
                      # Transform with one slider inside
                      Transform {
                            children [
                                  # Third slider has blue post
                                  DEF BLUE_SPECULAR Slider {
postColor 0 0 1 }
                            ]
                            translation 5 0 0 # side-by-side
                      }
                ]
                translation 5 0 0 # side-by-side
}
      ]
      translation 0 -30 0 # underneath diffuseColor sliders
```

```
}
# Three final Slider nodes, in three Transform nodes, for shininess,
transparency, ambient
Transform {
      children [
# The first slider has a yellow post
DEF SHINY Slider { postColor 1 1 0 }
# Transform with two Slider nodes inside
Transform {
                  children [
                        # Second slider has a mauve post
                        DEF TRANSPARENT Slider { postColor 1 0 1 }
                        # Transform with one slider inside
                        Transform {
                              children [
                                    # Third slider has white post
                                    DEF AMBIENT Slider { postColor 1 1
1 }
                              ]
                              translation 5 0 0 # side-by-side
                        }
                  ]
                  translation 5 0 0 # side-by-side
}
      ]
      translation 25 -30 0 # underneath emissiveColor sliders
}
# Now add a Sphere which has a Material node being manipulated by all
of this.
Transform {
      children [
            DEF SPOT_HANDLE SphereSensor { }

            DEF ROTATE_SPOT Transform {
                  children [
                        DEF SPOT SpotLight {
                              location 0 30 30
                              direction 0 -0.9 -0.7
                              beamWidth 0.40
                              cutOffAngle 0.60
                        }
                  ]
            }
            DEF BUILDER_ROTATE Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material DEF BUILDER Material {
                                          diffuseColor 0 0 0      #
everything off to start
                                          emissiveColor 0 0 0 # off
                                          specularColor 0 0 0 # off
                                          shininess 0
                                          transparency 0
                                          ambientIntensity 0
                                    }
                              }
```

```
                              geometry Sphere { radius 15 }
                        }
                  ]
            }
      ]
      translation 60 -15 0 # center the Sphere on the right
}

# Create a TimeSensor to spin the object every sixty seconds
DEF BUILDER_TIMER TimeSensor {
      cycleInterval 60
      loop TRUE
      startTime 1
      stopTime 0  # setup for autostart
}
# Create an OrientationInterpolator to spin the object
DEF BUILDER_SPINNER OrientationInterpolator {
      key [ 0, 0.5, 1 ]
      keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
}
# Now the Script which receives the Slider events and adjusts the
Material
DEF MAKER Script {
      eventIn SFFloat redDiffuse          # one eventIn per slider
      eventIn SFFloat greenDiffuse
      eventIn SFFloat blueDiffuse
      eventIn SFFloat redEmissive
      eventIn SFFloat greenEmissive
      eventIn SFFloat blueEmissive
      eventIn SFFloat redSpecular
      eventIn SFFloat greenSpecular
      eventIn SFFloat blueSpecular
      field SFColor localDiffuse 0 0 0          # local storage
      field SFColor localEmissive 0 0 0
      field SFColor localSpecular 0 0 0
      eventOut SFColor newDiffuse         # one eventOut per
exposedField
      eventOut SFColor newEmissive
      eventOut SFColor newSpecular
      url [ "javascript:
            // Three functions for the diffuse colors
            function redDiffuse (rd) {
                  localDiffuse.r = rd;    // Record color component
                  newDiffuse = localDiffuse;    // send it out
            }
            function greenDiffuse (gd) {
                  localDiffuse.g = gd;    // Record color component
                  newDiffuse = localDiffuse;    // send it out
            }
            function blueDiffuse (bd) {
                  localDiffuse.b = bd;    // Record color component
                  newDiffuse = localDiffuse;    // send it out
            }
            // Three functions for the emissive colors
            function redEmissive (re) {
                  localEmissive.r = re;
                  newEmissive = localEmissive;
```

```
            }
            function greenEmissive (ge) {
                  localEmissive.g = ge;
                  newEmissive = localEmissive;
            }
            function blueEmissive (be) {
                  localEmissive.b = be;
                  newEmissive = localEmissive;
            }
            // Three functions for the specular colors
            function redSpecular (rs) {
                  localSpecular.r = rs;
                  newSpecular = localSpecular;
            }
            function greenSpecular (gs) {
                  localSpecular.g = gs;
                  newSpecular = localSpecular;
            }
            function blueSpecular (bs) {
                  localSpecular.b = bs;
                  newSpecular = localSpecular;
            } "
      ]
}
# Now we have ROUTE statements for days.
# We need one for each eventIn and eventOut in the Script
# ROUTE from the diffuse sliders to the Script, and from the Script to
Material
ROUTE RED_DIFFUSE.sliderScalar TO MAKER.redDiffuse
ROUTE GREEN_DIFFUSE.sliderScalar TO MAKER.greenDiffuse
ROUTE BLUE_DIFFUSE.sliderScalar TO MAKER.blueDiffuse
ROUTE MAKER.newDiffuse TO BUILDER.set_diffuseColor
# ROUTE from the emissive sliders to the Script, and from the Script to
Material
ROUTE RED_EMISSIVE.sliderScalar TO MAKER.redEmissive
ROUTE GREEN_EMISSIVE.sliderScalar TO MAKER.greenEmissive
ROUTE BLUE_EMISSIVE.sliderScalar TO MAKER.blueEmissive
ROUTE MAKER.newEmissive TO BUILDER.set_emissiveColor
# ROUTE from the specular sliders to the Script, and from the Script to
Material
ROUTE RED_SPECULAR.sliderScalar TO MAKER.redSpecular
ROUTE GREEN_SPECULAR.sliderScalar TO MAKER.greenSpecular
ROUTE BLUE_SPECULAR.sliderScalar TO MAKER.blueSpecular
ROUTE MAKER.newSpecular TO BUILDER.set_specularColor
# The scalar fields of shininess, transparency and ambientIntensity
# Can be sent straight in - no Script node processing required!
ROUTE SHINY.sliderScalar TO BUILDER.set_shininess
ROUTE TRANSPARENT.sliderScalar TO BUILDER.set_transparency
ROUTE AMBIENT.sliderScalar TO BUILDER.set_ambientIntensity
# Spin the Sphere every sixty seconds
ROUTE BUILDER_TIMER.fraction_changed TO BUILDER_SPINNER.set_fraction
ROUTE BUILDER_SPINNER.value_changed TO BUILDER_ROTATE.set_rotation
# Attach the SphereSensor to the SpotLight
ROUTE SPOT_HANDLE.rotation_changed TO ROTATE_SPOT.set_rotation
```

You'll notice that I slipped in another sensor - SphereSensor - while adding the interactive features for this example. SphereSensor allows you to track a rotation, and it's used give you an opportunity to move the SpotLight over the Sphere. You can grab the Sphere, and as you drag on it, the SpotLight moves around the Sphere in every possible direction.

Well, there's still lots more that could be done with the Material Builder. You could add Viewpoint nodes so that you can read the values on the sliders easily. You could label the sliders by which field they manipulate. You could change the object on display from a Sphere to a Cone or Cube or whatever. The possibilities are endless, as is VRML.