Chapter 16: Illuminating Ideas

Bambi in the Headlights

We've forgotten something! In the 3D graphics primer, we noted that objects in the virtual world reflect the light shining on them, but we never talked about where that light came from. So where's the light source that's been illuminating all of these worlds? We haven't specified one in any of our VRML examples – something we'll learn to do in this chapter – but just the same, we've got light.

It's quite common for creators of worlds to forget to add light sources to those worlds – so common that the browser designers have added a feature known as the *headlight* to the VRML world. The headlight, like a lamp that's attached to the camera, shines light upon whatever you're looking at. Here's the trick - if you specify light sources in your world, the headlight won't come on, but if you neglect to specify light sources, the headlight will come on, and you'll be able to see your way around. Without the headlight, most VRML worlds would be pitch black.

You can turn the headlight on or off using the NavigationInfo node of VRML. Here's how it looks:

```
# Definition of the NavigationInfo node
NavigationInfo {
      avatarSize        # MFFloat, mult. values
      speed             # SFFloat
      type                    # MFString, mult. values
      headlight               # SFBool
}
```

The NavigationInfo gives the VRML browser direction on how to move you through the world. The speed field allows you to set how many units per second you'll move as you move through the virtual environment, while the headlight field is set to TRUE if you want the headlight to be on and FALSE if you want it to be off – so, even if you have light sources in a VRML world, you can force the headlight to be on.

The type field specifies all types of navigation modes that are available to the user; the field has an input value of MFString, so a list of navigation types can be given. Most browsers support both WALK and FLY modes, and many support an EXAMINE mode; if you don't want the user to be able to navigate through the environment – perhaps because you've established a pre-planned tour and don't want the user to get lost – you should set the value of the field to NONE.

Here's an example of a green Sphere with a NavigationInfo node that turns off the headlight and all navigation:

```
#VRML V2.0 utf8
# This is the first example on lights
```

```
# Turns off light and navigation
NavigationInfo {
      headlight FALSE         # no headlight, please
type [ "NONE" ]         # And no navigation
}
# And here's the green Sphere
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0 1 0
            }
      }
      geometry Sphere { }
}
```

When we load that into the browser we can't see very much. While the green Sphere is there – though very dimly – we don't see any of the navigation controls that we've always associated with the VRML browser. They're suppressed when the value in the type field is NONE.

Most browsers come up in WALK mode by default; sometimes that's inappropriate. In many cases we'd rather come up in an examination mode – "Study" in WorldView and "Spin" in Cosmo; do to this, we need to make EXAMINE the first value in the type field. In order to preserve the other navigation modes, we'd also need to have ALL as a field value – otherwise, only EXAMINE mode will be available. Here's how that might look, with the example given above, and the headlight turned on:

```
#VRML V2.0 utf8
# This is the second example on lights
# Examine mode navigation, with light
NavigationInfo {
      headlight TRUE                # we like headlights
type [ "EXAMINE", "ALL" ] # open in examine mode
}
# And here's the green Sphere
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0 1 0
            }
      }
      geometry Sphere { }
}
```

Once again we have our navigation controls, we can examine the object – and we have some light.

The NavigationInfo node gives you a lot of control over what the user can and can not do within your worlds; used wisely, it can guide users without letting them get lost.

Thomas Edison in Cyberspace

The basic light source in VRML, created with the PointLight node, makes a unidirectional light. It's very much like a bare lightbulb, shining equally in all directions. It has no visible shape – strangely enough, lights can exist without being visible entities. Here's a description of the PointLight node:

```
# Definition of the PointLight node
PointLight {
      on                    # SFBool
      intensity             # SFFloat
      ambientIntensity  # SFFloat
      color             # SFColor
      location              # SFVec3f
      radius                # SFFloat
      attenuation       # SFVec3f
}
```

The on field of the PointLight node explains itself; if TRUE, the light is on, and if FALSE, it's off. The intensity field requires a value between 0 – a light that's been dimmed out, and 1.0, a light that's shining as brightly as it can. The color field determines the color of the light – yes, you can have mood lighting in cyberspace – and allows you to have any combination between 0 0 0 (black and not visible) and 1 1 1 (white and completely saturated) as its value. The color defaults to 1 1 1 – a bright, white light.

The light can have a placement independent of the local coordinate system, by specifying a value in the location field. The radius field allows you to set how far the rays from this light will travel – such things are possible in cyberspace – and has a default value of 100 units; many times this value will be too small, and a light won't shine on things that it clearly should be illuminating. Don't reset the radius field unless you know it doesn't work with the default value – lots of lights shining on lots of objects slows down the rendering speed of most computers significantly.

For our first example, we'll position a light directly above a white Box:

```
#VRML V2.0 utf8
# This is the third example on lights
# Create white light
PointLight {
      location 0 5 0          # Above cube
}
# And here's the white Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 1 1
            }
      }
      geometry Box { }
}
```

When we examine this in the browser, we see a light shining from above.

Now, with a small change, let's use the color field to make that light red:

```
#VRML V2.0 utf8
# This is the fourth example on lights
# Red light from above
PointLight {
      color 1 0 0       # Red light
      location 0 5 0          # Above cube
}
# And here's the white Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 1 1
            }
      }
      geometry Box { }
}
```

And now let's cut the intensity in half:

```
#VRML V2.0 utf8
# This is the fifth example on lights
# Dim red light from above
PointLight {
      intensity 0.5           # dim light
      color 1 0 0       # Red light
      location 0 5 0          # Above cube
}
# And here's the white Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 1 1
            }
      }
      geometry Box { }
}
```

This dims the entire scene.

You can also have multiple light sources in any scene; although many lights will begin to slow the computer down, a judicious use of values in the radius field should alleviate that; use the radius field so that lights illuminate only those objects that they're intended to illuminate, rather than every object in the world. Here's the same cube, faced on every side by lights of a different color:

```
#VRML V2.0 utf8
# This is the sixth example on lights
# Create white light from above
PointLight {
      location 0 5 0          # Above cube
```

```
}
# Create purple light from underneath
PointLight {
      color 1 0 1
      location 0 -5 0   # beneath cube
}
# Create green light in front
PointLight {
      color 0 1 0
      location 0 0 5    # before cube
}
# Create red light from behind
PointLight {
      color 1 0 0
      location 0 0 -5   # behind cube
}
# Create blue light from the right
PointLight {
      color 0 0 1
      location 5 0 0    # Right of cube
}
# Create yellow light from the left
PointLight {
      color 1 1 0
      location -5 0 0   # Left of cube
}
# And here's the white Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 1 1
            }
      }
      geometry Sphere { }
}
```

We now have a sphere illuminated, in color, from all sides.

A great idea!

As I mentioned earlier, lights are not visible; they make objects visible by shining light on them, but they have no visible characteristic of themselves. However, they can be placed inside of "solid" geometry, and will shine through it. Here we use a Sphere, a Cylinder and a Transform node to make a "body" for the light source, and, for a change, we get to use emissiveColor in the Material node, because this object shines:

```
#VRML V2.0 utf8
# This is the seventh example on lights
# Keep all of the lightbulb together and above box
Transform {
      children [
            PointLight { }    # Fine just as it is.
            Shape {
                  appearance Appearance {
```

```
                              material Material {
                                    emissiveColor 1 1 0.8 # shines
                              }
                        }
                        geometry Sphere { radius 1.25 }
                  }
                  # Cylinder goes inside translation node
                  Transform {
                        children [
                              Shape {
                                    apppearance 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
      }
      # And here's the white Box
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 1 1 1
                  }
            }
            geometry Box { }
      }
```

We get a big lightbulb shining over a little box.

While it isn't always necessary to put geometry around your light sources, it's often a good visual cue someone browsing your worlds – visible lights are realistic lights; torches, flames, flashlights and spotlights all have their uses – and a light wrapped in the correct geometry becomes an intuitive flash of insight!

Track Lighting In Cyberspace

While many lights in our world resemble the "classic" lightbulb that illuminates everything equally in all directions, others – things like flashlights, spot lights and track lighting, are directional light sources – that means that the light has a direction – just like a camera – toward which its beam projects. You can create directed lights sources in VRML using the SpotLight node, a sister of the PointLight node. It has a few extra fields:

```
# Definition of the PointLight node
SpotLight {
      on                    # SFBool
      intensity             # SFFloat
      ambientIntensity  # SFFloat
      color             # SFColor
      location              # SFVec3f
      direction             # SFVec3f
      beamWidth             # SFFloat
      cutOffAngle       # SFFloat
      radius                # SFFloat
      attenuation       # SFVec3f
}
```

The three extra fields in the SpotLight node, direction, beamWidth, and cutOffAngle determine the directional characteristics of the SpotLight. The direction is a normalized vector – we discussed them in chapter 9 which determines how the light is pointing; the default value of 0 0 –1 means it points in, toward the scene. The beamWidth is a value in radians to describe the total width of the beam of light; it has a default value of 1.57 radians or 90 degrees. The cutOffAngle is the point at which the SpotLight begins to attenuate; this gives the light a central circle of full brightness, known as the *umbra*, and an outer circle, where it fades away to darkness, known as the *penumbra*. By default cutOffAngle is set to 0.7853 radians, or 45 degrees.

To illustrate, let's create a long flat surface – a floor, actually – and position a default SpotLight above it:

```
# This is the eighth example on lights
# Create white light
SpotLight {
      location 0 50 0         # Above floor
}
# And here's the Gray Floor
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 0.5 0.5
            }
      }
      geometry Box { size 100 0.01 100 }
}
```

When we look at this in the browser, we see that the floor is dark.

The default direction of the SpotLight points in, rather than down. Adding some direction information to the node will fix this:

```
# This is the ninth example on lights
# Create white light
SpotLight {
      location 0 50 0         # Above floor
      direction 0 –1 0        # Point down
}
```

```
# And here's the Gray Floor
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 0.5 0.5
            }
      }
      geometry Box { size 100 0.01 100 }
}
```

And now we can see that the upper surface of the floor is uniformly lit.

We can have multiple SpotLight nodes in a VRML world. Here, we add a blue SpotLight to illuminate the underside of the floor:

```
# This is the tenth example on lights
# Create white light above
SpotLight {
      location 0 50 0         # Above floor
      direction 0 –1 0        # Point down
}
# And the blue light below
SpotLight {
      location 0 50 0         # Above floor
      color 0 0 1             # color it blue
      direction 0 1 0         # Point up
}
# And here's the Gray Floor
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 0.5 0.5
            }
      }
      geometry Box { size 100 0.01 100 }
}
```

And now we have two differently colored faces of the floor.

For our final example with the SpotLight node, let's rework the sixth example to use six directional light sources:

```
#VRML V2.0 utf8
# This is the eleventh example on lights
# Create white light from above
SpotLight {
      location 0 5 0          # Above cube
      direction 0 –1 0 # point downward
}
# Create purple light from underneath
SpotLight {
      color 1 0 1
      location 0 -5 0   # beneath cube
      direction 0 1 0  # point upward
}
```

```
# Create green light in front
SpotLight {
      color 0 1 0
      location 0 0 5    # before cube
      direction 0 0 -1  # point in
}
# Create red light from behind
SpotLight {
      color 1 0 0
      location 0 0 -5   # behind cube
      direction 0 0 1  # point outward
}
# Create blue light from the right
SpotLight {
      color 0 0 1
      location 5 0 0    # Right of cube
      direction -1 0 0  # point left
}
# Create yellow light from the left
SpotLight {
      color 1 1 0
      location -5 0 0   # Left of cube
      direction 1 0 0  # point right
}
# And here's the white Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 1 1
            }
      }
      geometry Sphere { }
}
```

Once again we get a Sphere that's colored by lights from all sides.

SpotLight nodes can be used to create distinctive lighting effects, such as the highlighting of features on a tour of a virtual world, or an art gallery, or something as simple as a flashlight. Experimentation is key here – play around until you find just the combination of field values to produce the effect you're looking for.

Tune In, Turn On

Back when we introduced the PointLight node, we saw that its first field, on, could turn set the light source as either on or off. Much more than this, the on field can be used as a light switch – it can actually switch the light on or off. To understand how this happens – because our next project will be a light switch – we need to cover the basic concepts of interactive VRML, specifically *events* and *routes*.

Simply put, events are messages that can are sent from node in response to some activity – a user click, a timer counting down, the completion of an MPEG movie – and then passed along – or routed – to another node.

Events are composed of two different types of information; first there's a time stamp, which indicated the precise moment the event occurred; following that there's some data specific to the event, it's always one of the legal VRML data types. This package gets squirted out of a node, and routed to another node through a construction known as a route.

One way to think of events and routes would be to visualize a tin-can and string telephone; the mouth at one end emits event, the string routes the event to an ear, which receives the event.

Certain VRML nodes can emit events; these emitters are identified as eventOut items. Most VRML nodes have fields which can receive events; those receivers are identified as eventIn items. These complementary types, eventOut and eventIn, are the mouth and ear of VRML.

Start Making Sensor

One class of VRML nodes which can send events, that is, which have eventOut fields, are known as *sensors*. Sensors generate events – either on their own accord, or in response to some user-inspired activity. One of the most generic of the sensor nodes is known as the TouchSensor – it sends messages when ever a user touches (clicks on) objects with the mouse. Here's how the node looks – including its eventOut and eventIn info:

```
# Definition of the PointLight node
TouchSensor {
      enabled                 # SFBool
      isOver                  # eventOut, SFBool
      isActive                # eventOut, SFBool
      hitPoint_changed  # eventOut, SFVec3f
      hitNormal_changed # eventOut, SFVec3f
      hitTexCoord_changed     # eventOut, SFVec3f
      touchTime               # eventOut, SFTime
}
```

The TouchSensor node has only one field capable of storing a value, enabled, which defaults to TRUE. Everything else is information that the sensor can send as things happen to it. The isOver event sends a message with TRUE if the mouse is over the object that the TouchSensor has been attached to, otherwise it sends FALSE. If the user puts the mouse over the attached object and clicks on the mouse button, isActive sends a TRUE event as well.

Where would these events get sent? If you peek at our definition for the PointLight node, you'll see that the on field has a field data type of SFBool. TouchSensor events isActve and isOver send messages of data type SFBool. That means that we could route the output of either isActive or isOver from the TouchSensor to a PointLight.

How can an eventOut affect a field in another node? Get ready for a big revelation – most fields in most nodes are actually a combination of an eventIn, a field value, and an eventOut. The eventIn provides a way to change the value of a field, the field value

provides a place to store the value of the field, and the eventOut provides a way to send a message if the value of a field changes. Any field like this is known as an exposedField, and we'll start to note them when we come across them. For example the PointLight node really has a definition that looks like this:

```
# Definition of the PointLight node
PointLight {
      on                    # SFBool, exposedField
      intensity             # SFFloat, exposedField
      ambientIntensity  # SFFloat, exposedField
      color             # SFColor, exposedField
      location              # SFVec3f, exposedField
      radius                # SFFloat, exposedField
      attenuation       # SFVec3f, exposedField
}
```

PointLight has an exposedField named on. That means that on is actually an eventIn named set_on, a field value named on, and an eventOut named on_changed. Any event which changed the value of on would have to be routed into the PointLight through set_on. If some event did change the value of on, then a message would be emitted through on_changed.

An example will make all of this a whole lot clearer. Let's attach a TouchSensor to our lightbulb, and set things up so that if the mouse is over the lightbulb, the lightbulb turns on. We'll have to route a message from the TouchSensor to the PointLight:

```
#VRML V2.0 utf8
# This is the twelfth example on lights
# Keep all of the lightbulb together and above box
Transform {
      children [
            # Putting the TouchSensor in the same
            # children [] field attaches it to all objects
            # inside the children field
            DEF SENSOR TouchSensor { } # fine as default
            DEF LIGHT PointLight { }      # Fine just as it is.
            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
   }
   # And here's the white Box
   Shape {
         appearance Appearance {
               material Material {
                     diffuseColor 1 1 1
               }
         }
         geometry Box { }
   }
   # And now ladies and gentlement, we use a ROUTE
   # for the very first time.  This ROUTE establishes
   # the path between the SENSOR eventOut isOver
# and the LIGHT eventIn set_on.
ROUTE SENSOR.isOver TO LIGHT.set_on
```

To begin with, the TouchSensor goes into the children field of the Transform node which contains the lightbulb – this "attaches" the TouchSensor to all of the visible objects within the children field – including the Cylinder that's nested inside another Transform node. The TouchSensor is given the name SENSOR, so that its events can be identified. The PointLight is given the name LIGHT, so that its events can also be identified.

The very last line of the VRML file is a bit of magic. The ROUTE statement "wires" the eventOut isOver TO the eventIn set_on. That's the real reason we need to give objects names in VRML – without that capability, we'd never be able to meaningfully refer to the eventIn and eventOut parts of a node.

Does it work? Does the light turn on when you hold the mouse over it? You bet it does! Congratulations – you just created your first interactive world!

TouchSensor also has the isActive eventOut, which sends an event when the mouse is clicked on items attached to the TouchSensor. Here's the previous example, modified to use isActive:

```
#VRML V2.0 utf8
# This is the thirteenth example on lights
# Keep all of the lightbulb together and above box
Transform {
      children [
            # Putting the TouchSensor in the same
            # children [] field attaches it to all objects
            # inside the children field
```

```
                  DEF SENSOR TouchSensor { } # fine as default
                  DEF LIGHT PointLight { }      # Fine just as it is.
                  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
      }
      # And here's the white Box
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 1 1 1
                  }
            }
            geometry Box { }
      }
      # ROUTE between the SENSOR eventOut isOver
# and the LIGHT eventIn set_on.
ROUTE SENSOR.isActive TO LIGHT.set_on
```

The only change in all of this is the ROUTE statement; now isActive is routed to set_on. When we load the world, the light is on – that's as it should be. But when we click on the light, we can now turn it on and off.

Our final example will be to make a light switch separate from the light. We'll use the Box underneath the light, and we'll have to introduce a new node, Group, which will attach the Box to the TouchSensor; we'll place both in the Group node's children field. Here's the syntax for the Group node:

```
      # Definition of the Group node
```

```
	Group {
		children []	# MFNode, mult. values
		bboxCenter	# SFVec3f
		bboxSize		# SFVec3f
	}
```

The Group node does nothing more than collect nodes together, which – as we can see from this example – is sometimes very useful. Here we attach the TouchSensor to the Box, to switch the PointLight on and off – and we initialize the PointLight to off.

```
	#VRML V2.0 utf8
	# This is the fourteenth example on lights
	# 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 { }
                    }
                ]
        }
        # Route between the SENSOR eventOut isOver
# and the LIGHT eventIn set_on.
ROUTE SENSOR.isActive TO LIGHT.set_on
```

Now, when we click on the box, it acts like a light switch and turns the light on.

The light switch, while interactive, can hardly be called useful – you need to keep you mouse on it all the time. In a later chapter we'll figure out how to make the switch "stick" in the on and off positions, so we can make a real switch. But now, it's time to start a little spinning, because, baby, the Earth is gonna move…