Chapter 15: What's in a Name?

Cameras, Lights, Action!

Having made it this far through the text, you're probably wondering when we'll actually get to the interactive features of VRML, the ones that make it so incredibly cool. We've been picking up a few of those concepts all throughout these chapters, but it makes a lot of sense to have a full understanding of the visible qualities of objects before you learn to manipulate those qualities interactively. That way you'll have some idea of what you're doing when you're doing it, rather than just parroting examples from these pages.

If you've been faithful to following through the examples we've covered on shapes, materials, complex shapes, texture maps, and so forth, you should now have a good grasp of those basics – enough for us to move onto more advanced topics. These next chapters define the essential ideas at the heart of interactivity in VRML. We'll explore them by expanding on our "stage" metaphor in some very tangible ways; this chapter discusses how to use "cameras" in VRML – viewpoints that direct the user to look at particular objects from particular angles; and the next chapter discusses lights. Once we've covered lights and cameras, you – the director – can call "action" and watch the play begin.

The Name of the Nodes

Each VRML node has a unique name that identifies its function; Shape, Sphere, ImageTexture and Material all make their function explicit from their name. These names might be considered as the proper name for these nodes, or even the family name. Any node in VRML can also have a given name, much as you have both a given name and a family name. Once a node has been given a name, it can be uniquely identified from among its peers.

How does this work? VRML has a keyword – that is, a special, reserved word – known as DEF. It's always spelled out in capital letters. When this word is placed before a node's definition, the next word following the DEF becomes the given name of the node. Here's a Material node that's been given a name that corresponds to its function:

```
# Example of the DEF keyword
# Gives name "BLUE" to Material node
DEF BLUE Material {
      diffuseColor 0 0 1
}
```

The given name, BLUE, is given in capital letters, by convention; there's no rule for this, but it makes it much easier to identify given names of fields from their proper names – which can make your VRML much easier to read – especially when someone else reads it. Throughout the rest of this book, uppercase letters will identify nodes with a given name.

Reduce, ReUSE, Recycle

There are several reasons to give names to nodes, beyond identifying nodes uniquely; quite often we want to *instance* a previously defined node. Instancing means "to make a reference to or copy of an existing object". In other words, when you DEF a node, you can reuse it, again and again, simply by referring to it by its given name. When you do this, you need to add the VRML keyword USE before the given name – this lets the browser know that you're about to instance a node from its name.

For our first example of how this works, let's DEF a Shape that creates a blue Sphere, and then USE it inside of a Transform node, creating two objects from the same definition:

```
	#VRML V2.0 utf8
	# This is the first example on naming nodes
	# We name this Shape so we can reuse it
	DEF BLUE_SPHERE Shape {
		appearance Appearance {
			material Material {
				diffuseColor 0 0 1 # blue
			}
		}
		geometry Sphere { }
	}
	# This Transform has the DEF node inside it
Transform {
		children [
			USE BLUE_SPHERE  # puts Sphere here, too
		]
		translation 0 2 0  # above
	}
```

We should see two blue Sphere nodes, one atop the other.

It's also possible to predefine certain nodes, such as Material nodes, so that they can be used as required, rather than being defined every single time. Here we define the three basic colors, then use them with three nodes of different colors:

```
	#VRML V2.0 utf8
	# This is the second example on naming nodes
	# Define a red material – causes small error
	DEF RED Material {
		diffuseColor 1 0 0
	}
	# Defines a green material – causes small error
	DEF GREEN Material {
		diffuseColor 0 1 0
	}
	# Defines a blue material – causes small error
	DEF BLUE Material {
		diffuseColor 0 0 1
	}
	# Create Red Box using DEF
```

```
      Shape {
            appearance Appearance {
                  material USE RED # Red color
            }
            geometry Box { }
      }
      # Transforms for green sphere and blue cone
Transform {
            children [
            Shape {
                        appearance Appearance {
                              material USE GREEN # Green color
                        }
                        geometry Sphere { }
                  }
                  # Cumulative transform blue Cone
                  Transform {
                        children [
                        Shape {
                                    appearance Appearance {
                                          material USE BLUE
                                    }
                                    geometry Cone { }
                              }
                        ]
                        translation 2 2 0
                  }
            ]
            translation 0 2 0  # above
      }
```

We can see three objects, which have been created by DEF Material nodes.

NOTE: You will encounter some errors when you load this example – that won't effect the results of the example. But you should always use a Material node inside of an Appearance node; if you don't, you may get these errors.

Hiding the Truth

Since we must define the Material nodes within Appearance nodes before can use them somewhere else, this is a better version of the previous example:

```
      #VRML V2.0 utf8
      # This is the third example on naming nodes
      # The Switch node allows you to select
      # A visible shape from a large set of shapes.
      # When whichChoice –1, none are visible.
      Switch {
            choice [
                  # Shape that's not really used
                  Shape {
                        appearance Appearance {
                              material DEF RED Material {
                                    diffuseColor 1 0 0
```

```
                        }
                    }
                    geometry Box { }
                }
                # Shape that's not really used
                Shape {
                    appearance Appearance {
                        material DEF GREEN Material {
                            diffuseColor 0 1 0
                        }
                    }
                    geometry Box { }
                }
                # Shape that's not really used
                Shape {
                    appearance Appearance {
                        material DEF BLUE Material {
                            diffuseColor 0 0 1
                        }
                    }
                    geometry Box { }
                }
            ]
            whichChoice –1  # selects none of the objects
        }
        # Create Red Box using DEF
        Shape {
            appearance Appearance {
                material USE RED # Red color
            }
            geometry Box { }
        }
        # Transforms for green sphere and blue cone
Transform {
            children [
            Shape {
                    appearance Appearance {
                        material USE GREEN # Green color
                    }
                    geometry Sphere { }
                }
                # Cumulative transform blue Cone
                Transform {
                    children [
                    Shape {
                            appearance Appearance {
                                material USE BLUE
                            }
                            geometry Cone { }
                        }
                    ]
                    translation 2 2 0
                }
            ]
            translation 0 2 0  # above
        }
```

We're seeing a new node here, the Switch node. Here's its definition:

```
# Definition of Switch node
Switch {
      choice [ ]  # MFNode, mult. values
      whichChoice # SFInt32
}
```

The Switch node allows you to have several representations of the same visible object, like the LOD node, but it allows you to select which of the representations is visible. In the example given above, three Shape nodes – that is, three representations – reside within the choice field of the Shape node. These representations are given index values – again, starting from zero, so the first Shape is index 0, the second, index 1, and the third, index 2. The value in the whichChoice represents the index value of the visible Shape; if the value is –1, none of the nodes within the choice field are visible. That's the case in the example above; although all of the Shape nodes within the choice field of the Switch have been defined, none of them can be seen. This allows us to DEF the RED, GREEN, and BLUE Material nodes associated with the Shape nodes, without causing any errors.

If this seems a little roundabout, that's only because we needed a quick-and-dirty way to define nodes without making them visible; in a real VRML file, you'd simple DEF the first red Material node with the given name RED, then continue to use it throughout the world, and do the same for any other materials. This is a very small world, so we had to go to some lengths to demonstrate how this works. We'll learn more about the Switch node later on, when we learn how to change the value in the whichChoice field interactively – from within the VRML world.

It All Depends on Your Point-of-View

Until this point, we've relied on the VRML browser to do its best job placing a camera into the world for us; it tries to figure out what objects are where, and puts you down somewhere in the "middle" of the world. That isn't always what you want – in fact, it rarely is. For the simple worlds we've covered so far, it's been more than adequate, but as we get into more sophisticated simulations, we'll want to have finer control on where we get "dropped" into cyberspace. That control comes through the VRML Viewpoint node, which looks like this:

```
# Definition of Viewpoint node
Viewpoint {
      fieldOfView             # SFFloat
      jump                    # SFBool
      orientation       # SFRotation
      position                # SFVec3f
      description             # SFString
}
```

Just like a camera lens, the Viewpoint node allows you to define a field-of-view (FOV) for the world. A wide field of view gives everything a fish-eye look, while a narrow field of view gives it a telephoto-lens feel. The default value for the fieldOfView field, given

in radians, has a default value of 0.785398 radians, or 45 degrees. That's an average value, and should give you good results, in most cases. The largest possible value for fieldOfView is 3.14 radians, or 180 degrees – anything more and you'd be able to see behind your head!

The position and orientation fields allow you to set the camera's axes of translation and axes of rotation – where it is and what it's looking at. Let's look at a few examples to see how this works. If we look back the fourth example from chapter 9, we'll see a central cube that's surrounded by six colored spheres.

Let's say we want to setup a Viewpoint so that we look down on the cube from above. We'd want to keep the same fieldOfView, but we'd want a position slightly elevated in Y, while we'd want to keep to the origin for X and Z. For the orientation field, we want to pitch downward 90 degrees in, that is, rotate (with a negative value) 1.78 radians in X. Putting it all together, here's how it looks:

```
#VRML V2.0 utf8
# This is the fourth example on naming nodes
# Here's the Viewpoint node
Viewpoint {
      position 0 30 0   # slightly above
      orientation 1 0 0 -1.78  # pitch down 90 degrees
}
# This is the cyan Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0 1 1
            }
      }
      geometry Box { }
}
# This Sphere is to the right and green
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 3 0 0 # move slightly to the right
}
# This Sphere is to the left and blue
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
```

```
                              diffuseColor 0 0 1 # blue
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation -3 0 0 # move slightly to the left
}
# This Sphere is to the front and red
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 0 0 # red
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 0 3 # move slightly to the front
}
# This Sphere is to the back and yellow
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 1 0 # yellow
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 0 -3 # move slightly to the back
}
# This Sphere is to the top and white
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 1 1 # white
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 3 0 # move slightly to the top
}
# This Sphere is to the bottom and gray
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
```

```
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0.5 0.5 0.5
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 –3 0 # move slightly to the bottom
}
```

Now we can see the view from the top.

Can’t Get There from Here

What happens if we move around? Can we get back to our initial entry point? Not unless we reload the scene – but if we move again, we’re lost again. We need to give the user some control over the camera, so they can come back to the view we want them to have. That’s what the Viewpoint node’s description field is for. It allows you to attach a text string to the Viewpoint, with which the user can identify the camera. Our camera gives us a view from the top, so let’s change the previous example just a bit:

```
#VRML V2.0 utf8
# This is the fifth example on naming nodes
# Here’s the Viewpoint node
Viewpoint {
      position 0 30 0   # slightly above
      orientation 1 0 0 -1.78  # pitch down 90 degrees
      description "View from Above"
}
# …file goes on after this…
```

When we reenter the world, it looks the same, until we take a peek at the Viewpoints menu. The text within the description field of the Viewpoint node has now become an identifying menu item in the Viewpoints menu. What does it do? If we zoom off into the empty black, far away from the model, we can select the “View From Above” viewpoint and immediately find ourselves back at the point where we entered the model. Neat, huh?

The Three-Camera Shoot

It gets even better when we define multiple cameras. Here we’ll add another Viewpoint, which creates a view from underneath:

```
#VRML V2.0 utf8
# This is the sixth example on naming nodes
# Here’s the Viewpoint node
DEF TOP Viewpoint {
      position 0 30 0   # slightly above
      orientation 1 0 0 -1.78  # pitch down 90 degrees
```

```
      description "View from Above"
}
# Here's the Viewpoint node
DEF BOTTOM Viewpoint {
      position 0 -30 0  # slightly below
      orientation 1 0 0 1.78  # pitch up 90 degrees
      description "View from Below"
}
# …file goes on after this…
```

We used DEF to give the two Viewpoint nodes names, so that we could tell them part quickly. When we enter the world, we find ourselves looking at the TOP viewpoint, but we can see that the Viewpoints menu has two entries.

When we select "View from Below", we find ourselves looking up from underneath.

You see that as we moved from one camera to another, the camera animated the motion – that is, it looked as though the camera was actually moving through cyberspace as it moved from one Viewpoint to the next. This is known as a camera "jump", and if you don't want the animation, you can turn if off by setting the jump field of the Viewpoint node to FALSE (the default is TRUE). In this example, we've disabled camera jumps:

```
#VRML V2.0 utf8
# This is the seventh example on naming nodes
# Here's the Viewpoint node
DEF TOP Viewpoint {
      position 0 30 0   # slightly above
      orientation 1 0 0 -1.78  # pitch down 90 degrees
      description "View from Above"
      jump FALSE  # no animation
}
# Here's the Viewpoint node
DEF BOTTOM Viewpoint {
      position 0 -30 0  # slightly below
      orientation 1 0 0 1.78  # pitch up 90 degrees
      description "View from Below"
      jump FALSE  # no animation
}
# …file goes on after this…
```

Now, as you see when you move between viewpoints, when you load the world, you transition directly from one view to another.

The entry viewpoint is determined by the first Viewpoint node that the browser encounters. If we wanted the view from below to be the entry view, we could reverse the viewpoints, like this:

```
#VRML V2.0 utf8
# This is the eighth example on naming nodes
# Here's the Viewpoint node
DEF BOTTOM Viewpoint {
      position 0 -30 0  # slightly below
      orientation 1 0 0 1.78  # pitch up 90 degrees
```

```
      description "View from Below"
}
# Here's the Viewpoint node
DEF TOP Viewpoint {
      position 0 30 0   # slightly above
      orientation 1 0 0 -1.78  # pitch down 90 degrees
      description "View from Above"
}
# …file goes on after this…
```

Now we enter with the BOTTOM Viewpoint.

For the final example, let's add a third camera, for a side view of the world:

```
#VRML V2.0 utf8
# This is the ninth example on naming nodes
# Here's the Viewpoint node
DEF TOP Viewpoint {
      position 0 30 0   # slightly above
      orientation 1 0 0 -1.78  # pitch down 90 degrees
      description "View from Above"
}
# Here's the Viewpoint node
DEF BOTTOM Viewpoint {
      position 0 -30 0  # slightly below
      orientation 1 0 0 1.78  # pitch up 90 degrees
      description "View from Below"
}
# Here's the Viewpoint node
DEF SIDE Viewpoint {
      position 30 0 0   # slightly to the right
      orientation 0 1 0 1.78  # yaw 90 degrees
description "View from the Side"
}
# …file goes on after this…
```

You can add as many Viewpoint nodes to a world as you'd like; each of them – when given a name in their description field – will show up under the Viewpoints menu. As a working example, why not extend this until you have all six sides of the cube covered? Then, like a Hollywood director, you'll be a master of the camera!