Chapter 7: Flagrant Materialism

A Shallow Subject?

Every visible item in the virtual world is composed of two types of information. *Geometry* information speaks to the form of an object – cubic, spherical, etc., while *material* information covers the surface qualities of those forms – color, shininess, glow, etc. Together, they provide the necessary set of definitions to create a visible entity. So far, we've only discussed the geometric forms of objects, and those only just barely. Before we go any further, we must complete our definition of the visible object, adding surface to form. To do that, we have to talk about the appearances of things. This chapter is all about surface appearances – if you'll pardon the pun. In the virtual world, surfaces are reality, as objects are only skin-deep.

Shape and Appearance

As stated in the last chapter, the Shape node provides a way to bind the form of an object to its surface qualities. We explored the use of the geometry field in the Shape node – and used several of VRML's built-in shapes to create basic object forms. The Shape node has another field, appearance, which is where the surface qualities can be applied to a form specified in the geometry field. In its pure form, the Shape node looks like this:

```
Shape {                 # definition of Shape node
      appearance  # field, takes SFNode
                        # you'd likely have
                        # an Appearance node here
      geometry          # field, takes SFNode
                        # you'd have some form info here
}
```

Just as the geometry field takes another node as its value – that's what an SFNode field type specifies – the appearance field uses a node as input. In this case, however, the appearance field nearly always has an Appearance node as its input. The Appearance node has several fields to handle the various surface qualities an object can have:

```
Appearance {            # definition of Appearance node
      material          # field, takes SFNode
      texture           # field, takes SFNode
      textureTransform # field, takes SFNode
}
```

These three fields – material, texture and textureTranform – can be used to define a very wide range of visible surface qualities for any given form. The fields texture and textureTransform, used for texture maps, will be covered in a later chapter. For the moment, we'll focus on the material field. The material field takes an SFNode as an input, and - once again - the material field nearly always takes a Material node as its input. The Material node – the real focus of this chapter – defines a coating, almost like

a “paint” that’s applied to a form. This coating has a number of qualities, accessible through the fields in the Material node:

```
Material {        # definition of Material node
      ambientIntensity  # SFFloat (range 0 -> 1.0)
      diffuseColor            # SFColor
      emissiveColor           # SFColor
      shininess               # SFFloat (range 0 -> 1.0)
      specularColor           # SFColor
      transparency            # SFFloat (range 0 -> 1.0)

}
```

Colors, Colors – Everywhere!

You can see that there are three fields that provide color information; how can that be? How can an object be three colors at once? Well, in the lingo of computer graphics, all objects really do possess three colors. *Diffuse colors* are said to be the colors that objects reflect when light is shined on them. For example, an American flag has diffuse colors of red, white and blue. *Emissive colors* are the colors that an object emits, as if it were glowing. The Sun emits a bright yellow – something we’ll learn to duplicate in a moment. Finally, *specular colors* are the colors of the reflected highlights of an object. My hair, for instance, has red highlights, even though it’s dark brown. Every object’s surface is some combination of diffuse, emissive and specular colors; the correct arrangement can mean the difference – in appearance - between a shiny metal and a light-absorbing velvet.

Diffusing Controversy

There’s a lot of fields here, and each of them are important, so we’ll start with the most important first – that would be diffuseColor. In the most absolute sense, the diffuseColor of an object is *the* color of an object – that is, the color that’s reflected toward you when a light shines on the object. It has a data type of SFColor, which means it need a red, green, blue color triplet supplied to it. For example, if we were to define a green sphere in VRML 2.0, this is how we’d do it:

```
#VRML V2.0 utf8
# This is the first example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 1 0 # green
            }
      }
      geometry Sphere { radius 1 }  # Create form
}
```

Notice that the appearance field has an Appearance node inside that, with a material field with a Material node inside of that. It’s this *nesting* capability which gives VRML some of its unique strength. If we look at it in a browser, we’ll see a green sphere.

NOTE: Unlike the last chapter, these examples are just as they appear. You can type them in yourself or use the CD-ROM based example; it won't matter at all.

The values for the SFColor data type must be between 0 and 1.0 – anything else is strictly forbidden, and may just crash your browser. If we wanted to create a cyan Box, we'd give equal weight to both green and blue color values, like so:

```
#VRML V2.0 utf8
# This is the second example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 1 1 # green AND blue
            }
      }
      geometry Box { size 1 1 1 }   # Create form
}
```

The combination of all colors at their maximum value – that is, at 1.0 – creates white, while the combination of all colors at their minimum value of zero, creates black. Here's a recipe for a black Cone, a witch's cap:

```
#VRML V2.0 utf8
# This is the third example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 0 0 # no colors, black
            }
      }
      geometry Cone { } # Create form, use defaults
}
```

And as you can see, it's doesn't look like it's there at all.

But we could turn the Cone gray by giving each of these colors a value of 0.5:

```
#VRML V2.0 utf8
# This is the fourth example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0.5 0.5 0.5 # gray
            }
      }
      geometry Cone { } # Create form, use defaults
}
```

Now that it's gray, we can see it.

And let's say that we wanted to create a white Cylinder – perhaps for a column in a building. We'd do that like this:

```
#VRML V2.0 utf8
# This is the fifth example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 1 1 1  # white
            }
      }
      geometry Cylinder { }   # use defaults
}
```

Which would create something that looks quite a bit like a column for a building.

What happens if we don't provide a value for the diffuseColor field? The default values are 0.8 0.8 0.8 – something between gray and white. You've seen this color before; all of the models in the last chapter used this default coloring.

Each of the three values in the SFColor triplet can be set to any value between 0 and 1.0 – which means that a very wide range of color values can be created, at least as many as can be shown on your average computer display. Be a little inventive – some of the colors that can be created with diffuseColor are very subtle, others very dramatic. But you'll never know unless you play a bit…

That Glowing Feeling

The major shortcoming of diffuseColor is that it's dull, it's meant to reflect light. Obviously, that'd be inappropriate for a model of something glowing, like a light bulb, a lava lamp, or a star. For these situations – and they're not as rare as you might think – the Material node defines the emissiveColor field. Once again, emissiveColor field takes an SFColor triplet as its input value. Let's say, for starters, that we wanted to create a big glowing, yellow ball, rather like our own Sun – we'd do that by defining an emissiveColor of 1 1 0 (red plus green equals yellow). But first, and very importantly, we must turn off any diffuseColor for the object – that is, set it to black. An object should not glow and reflect light at the same time. That said, here's how the example might look:

```
#VRML V2.0 utf8
# This is the sixth example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 0 0 # no diffuseColor
                  emissiveColor 1 1 0 # glow yellow
            }
      }
      geometry Sphere { radius 1 }  # Create form
}
```

You'll see that this model looks very different from the earlier ones – this one really does seem to glow.

Or, if we wanted a Box that glowed with a dim red glow, it might look like this:

```
#VRML V2.0 utf8
# This is the seventh example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 0 0 # no diffuseColor
                  emissiveColor 0.25 0 0 # glow faint red
            }
      }
      geometry Box { size 1 1 1 }   # Create form
}
```

This gentle glow is only just visible, but still looks substantially different than if we had used diffuseColor to generate a faint red coloring.

It almost looks like an ember, cooling, doesn't it? That's precisely the kind of range of effects that become available to you when you use emissiveColor.

If you don't define the emissiveColor field when you create a Material node, its SFColor triplet is set to 0 0 0, so the surface won't have any emissive properties unless you explicitly define them.

Shiny Happy Martian Apples

The two fields shininess and specularColor are intimately related, both having to do with the specifics of how a surface reflects light back at you. An easy way to think about this is to you the example of the apple in the orchard. While growing on a tree, the apple is rained on, sprayed by pesticides (hopefully, only lightly), windblown, and so forth. While it's on the tree it's probably a pretty dull thing. Now suppose you come along, on a bright sunny day, and pick that apple. You hold it up, and you see that it's reflecting some light back at you. Most of the apple is red, but there's a corner which is rather more white, where it's actually reflecting a dim reflection of the Sun's image back at you. If you polish the apple, by rubbing it with a cloth, that spot of reflection, which was vague just a moment ago, will be a tight, bright point. The tightness of that point is the *shininess* of the apple, and this quality is represented by the shininess field in the Material node.

On the other hand, let's say that this is a Martian apple, which has got blue highlights all the way through it (helping it to absorb the weaker sunshine out Mars' way). The pinpoint of reflected light will be tight, but it might be bluish – because that's the natural highlight color of the object. This is known as the *specular color* of the object, and is represented by the specularColor field in the Material node.

Let's say we wanted to model a Sphere based upon the characteristics of the Martian apple. We'd want to give the object a red surface using diffuseColor, then we'd want to give it a blue highlight using specularColor. It might look like this:

```
#VRML V2.0 utf8
# This is the eighth example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0.75 0.1 0.1 # redish
                  specularColor 0 0 0.75 # blue highlight
            }
      }
      geometry Sphere { radius 1 }  # Create form
}
```

When we pop it into the VRML browsers (make sure your headlight is on), you can see that in the center of the Sphere - where the light is most directly reflecting back toward you and your headlight – there's a definite blue aspect to this very curiously colored shape. It's somewhat diffuse because we haven't yet specified a value for the object's shininess – so it takes a default value at a rather dull 0.2. If we tighten that up significantly, we'll see some changes:

```
#VRML V2.0 utf8
# This is the ninth example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0.75 0.1 0.1 # redish
                  specularColor 0 0 0.75 # blue highlight
                  shininess 0.75    # pretty shiny
            }
      }
      geometry Sphere { radius 1 }  # Create form
}
```

This looks somewhat similar, but the center is might tighter, and much more focused.

Metals and metallic surfaces also have all sorts of reflective qualities that can be created through the judicious use of the specularColor and shininess fields. For example, an aluminum plate might be defined with a diffuse color of dark gray, with high shininess and bright specular highlights:

```
#VRML V2.0 utf8
# This is the tenth example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0.35 0.35 0.35 # dark gray
                  specularColor 1 1 1 # bright
                  shininess 1 # pretty shiny
            }
      }
      geometry Box { size 10 0.01 10 }    # Create form
}
```

When you pop it into the browser, you'll need to spin it around a bit before you can see it. But when you do, you'll see that it looks a lot like sheet metal.

If you don't need any highlights on an object, you can leave the specularColor field undefined; it's default value is 0 0 0; like emissiveColor, you don't need to define it unless you use it.

Utterly Transparent

The last of the fields in the Material node is the most transparent – literally! The transparency field controls the "clarity" of the surface of the object. A value of 1.0 is completely transparent, while a value of 0 – the default – is completely opaque. Anything else in the range is translucent, to varying degrees.

Although any value between zero and one is legal in the transparency field, in computer graphics it's the convention to use values in quarter-step increments; 0, 0.25, 0.5, 0.75 and 1.0. These should be enough for most situations – but don't be afraid to experiment with other values.

Here's how we'd define a blue Cone that's 50% transparent:

```
#VRML V2.0 utf8
# This is the eleventh example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 0 1 # blue
                  transparency 0.5 # halfway transparent
            }
      }
      geometry Cone { } # use defaults
}
```

We'll see that it's translucent when we put it in a browser.

It's difficult to tell how translucent the Cone really is; after all, there isn't anything else in the world to see. If you're not using fancy 3D graphics hardware, chances are that your browser is using something called *screen-door transparency*. This means the object almost looks as if it's been put through a screen to make it transparent. It's a technique that saves time, but you'll often find that transparency slows your worlds down – sometimes quite a bit.

If we jump ahead just a little bit – using concepts we'll cover in our next chapter – let's put a green Sphere behind the blue Cone, and we'll get a better sense for how transparent the Cone really is:

```
#VRML V2.0 utf8
# This is the twelfth example on Materials
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 0 1 # blue
                  transparency 0.5 # halfway transparent
            }
```

```
        }
        geometry Cone { } # use defaults
    }
    # We'll cover this in the next chapter
    Transform { # Just getting your appetites whet
        children [
            Shape {     # Create a green sphere
                appearance Appearance {
                    material Material {
                        diffuseColor 0 1 0
                    }
                }
                geometry Sphere { radius 1 }
            }
        ]
        translation 0 0 –10 # position behind Cone
    }
```

Now you can see that the Cone is translucent; it's possible to see the green Sphere that's been placed behind it. The transparency field is great for creating glass effects, of course, but also can be used to create pseudo-underwater environments, atmospheric effects, and other types of "shimmers".

Moving Right Along…

We've just covered the basics of surfaces, how they're created and used. There's a lot hear to play with – as you'll see when we get further into the book. Almost any surface in the real world can be recreated with some fidelity by a careful use of the Material node and its fields.

But it's time to be moving along. You can now create objects – both forms and materials, but you don't yet know how to place them. In our next chapter you'll learn that there's a place for everything, and everything in its place…