Chapter 6: At Play in the Fields of the Node

An Introduction to VRML

First and foremost, VRML is a computer language. While it shares a modest similarity to other computer languages, such as *Java* or "C," VRML is specifically designed to be an excellent language for interactive computer graphics – virtual reality. For this reason, VRML has a whole host of built-in features that make it very easy to create rich worlds – as you'll see.

The ancestry of VRML is the biggest reason why it's such a strong graphics language. VRML was originally adapted from a language developed at Silicon Graphics, Inc. This predecessor to VRML, *Open Inventor*, was itself developed as an all-encompassing graphics language. The developers of Open Inventor gleaned the best parts of all of the graphics languages available and synthesized them, giving Open Inventor a large amount of expressiveness and flexibility, while keeping it easy to learn and understand.

Although VRML started as a descendant of Open Inventor, it quickly outgrew this heritage. After the introduction of Java, it became clear that both languages would need to grow *together* - as a programming language Java provides great flexibility, but is very difficult to learn; VRML as a *scene description language*, is much easier to learn, but, in some ways, less flexible. The combination of VRML and Java overcomes the limitations of both, and – together in VRML 2.0 – forms the basis for a complete environment for the creation of virtual worlds.

However, this is an introductory text – there's no reason that you should be required to take a course in object-oriented programming just to learn VRML. And Java is an object-oriented language, something that can take people years to master. But don't despair! Java has an easy-to-understand cousin, *JavaScript*, which is also well-integrated into VRML. You can learn JavaScript in a few hours – instead of a few months – and, although it lacks a few of the very advanced features of Java, it's perfectly well suited to everything we'll be covering in the next chapters.

On the other hand, if you're already proficient in Java, and want to learn how to work with VRML and Java, be sure to pick up Roger Lea's excellent *Java and VRML for 3D Worlds*, also from New Riders Publishing. It will be a very good complement to what we're about to cover.

Laying out the Plan

Over the next five chapters, we'll lay out the basic structure of VRML's visible qualities, starting with the essence of VRML as a language, and moving on to the essential elements of that language – shapes and surface materials. Once those basics have been

covered, we'll deal with the orientation of those objects in space and their placement, followed by a discussion of more complex shapes. Finally we'll cover texture maps – essentially the equivalent of wallpaper in cyberspace. By the end of these five chapters – the foundation of the visible aspects of VRML - you'll be able to create any VRML form, simple or complex.

Heading Up the Cast

In any definition of a VRML world, the very first thing that must occur in the file which defines that world is the VRML file header. The header is basically a declaration to the computer which says, "This is a legal VRML file, honestly." The header is a line that looks precisely like this:

```
#VRML V2.0 utf8
```

What does it all mean? The pound symbol indicates a *comment* – we'll cover that next. "VRML" indicates the language (yeah, we know that), while "V2.0" identifies this as the second versions of VRML – the one which includes support for Java, JavaScript, and many other cool features we'll cover here. Finally "utf8" indicates that VRML 2.0 supports the International Standards Organization's Universal Text Format, in its 8-bit form. ISO's UTF-8 is a way to specify non-Roman sets of characters, such as the ones that make up Russian, Japanese, Chinese or Korean. This means that when you use text in your virtual worlds – and yes, you can use text – you aren't confined to the character sets normally associated with European languages.

Running Commentary

One of the most useful things you can remember to do while you're learning to program in VRML is to *document* your program by adding comments to the program. Comments are normal text, which begin with a pound symbol "#" and continue to the end of the line. You can put anything you like inside a comment, any note to yourself – useful if you're returning to a program after some time away – or any comment that might be useful for anyone else who comes along to look at your VRML code.

Here's an example of a VRML file that's legal, but doesn't do anything:

```
#VRML V2.0 utf8
# This is a comment line – it begins with the pound symbol
# It continues to the end of the line
# Which is why we needed another one
# You can have as many comment lines as you might like
# Just so long as every one begins with a pound symbol
# You can also have blank lines in your VRML programs.
# Blank lines are ignored.
```

As you can see the program begins with the VRML file header, that makes it legal VRML 2.0. Following that are a number of comment lines. It's generally a good idea to

create a block of comments at the top of your VRML program, which tell you what the program does, when it was created, who created it, when it was changed, and so forth.

The more you comment your programs now, the less you'll have to regret later. Trust me.

Atomic Theory

The foundation of VRML, its basic, atomic element, is known as the *node*. Anything that happens inside of VRML happens within a node; shapes are defined, colors are assigned, and interactivity detailed – all within the node. Nodes have names, which identify what they do with a common sense name like Box, Sphere, Coordinate and Shape.

By convention in VRML, the name of a node always begins with an uppercase letter, like a proper name, which is followed by a space and then an opening brace. Between the opening brace and a closing brace come the definition of the node's *fields*, which we'll cover in detail in just a moment. A node will always look something like this:

```
                    # An example of a "fake" node.
Node {              # The name of the node and opening brace,
                    # with lots of interesting fields in here,
}                   # And a closing brace,
                    # Which concludes this example.
```

There are fifty-five basic nodes in VRML 2.0, from these fifty-five building blocks you can build worlds of infinite interactivity.

Fields of Dreams

What goes on inside a node? Well, that depends upon what the node is. Some nodes have lots of information within them, while some have practically none at all. Any information inside of a node is placed within *fields* inside the node. These fields have names, as do nodes, but field names begin with a lower-case letter. Most field names are incredibly sensible. For example, the Sphere node has a single field, radius, which specifies the radius of the sphere. (Does this make sense or what?) That node might look something like this:

```
Sphere {            # Define a Sphere node
      radius 1.0    # A radius of one VRML unit
}                   # And that's all, folks
```

As you can see from this very simple example, the radius field has been supplied a single value, the number 1.0, which specifies a radius of 1.0 VRML units.

NOTE: While units in the cyberspace of VRML don't really mean anything in the real world, by convention, **a unit in VRML is equal to one meter in the real world**. It's arbitrary, but that's the way it is.

Type Casting

How do you know what goes into a particular field? You can't always tell from the field's name. But every node has a *definition*, which specifies all of its fields and the *data types* for each field. The data type defines the kind of information that goes into the field. There are different kinds of information, such as character information – that is, alphabetic information – or numeric information, or color information, and so forth.

Data types have names, and always begin with "SF" or "MF" – the difference will be discussed later. Don't get scared off by the "techie" sounding names of the fields; it's very much like labels on a spice rack. Here's a table of the basic data types and what they specify:

SFBool, MFBool	A value of either TRUE or FALSE. Comes from the word "boolean", named after British mathematician George Boole, who invented binary logic.
SFFloat, MFFloat	A "floating-point" value. In computer lingo, a floating-point number is a number of *any* value that might contain a decimal point. Examples of floating point numbers are: 1 0.02 -3011
SFColor, MFColor	A color "triplet". The triplet defines values for the red, green, and blue portions of the color, which are floating point values – but must be between zero and one. For example, the triplet which would define a pure red color would be 1 0 0, while white – which is all colors together – would be 1 1 1. On the other hand, black would be 0 0 0. Because fractional combinations are possible, such as gray at 0.5 0.5 0.5, an infinite range of color can be created.
SFInt32, MFInt32	An integer – that is, a number with no

	decimal part, such as 2, 32, or –53. This integer has a maximum and a minimum range, it can't be any bigger than 2,147,483,647 or less than –2,147,483,648.
SFString, MFString	A character string, inside quotes. Some examples: "This is a string" "Here's another one." "Say, \"Goodnight!\"" When you want to have a quote inside of a string, you need to put a backslash "\" character before it. That tells the computer that the quote is inside the string, rather than ending it.
SFVec3f, MFVec3f	A three-dimensional *vector*, really just three values grouped together. A we discussed in the 3D primer, all three dimensional points have three values, for x, y, and z, so this vector is composed of three floating-point values, with any range. Some examples would be: 0 99 2.202125 -16.1 0.001 15 1 3 65 Notice that there are no commas between the numbers, just a space.
SFVec2f, MFVec2f	A two-dimensional vector, actually just two values grouped together. This is just like the 3D vector, but, instead of x, y, and z values, the vector only has x and y values, representing a two dimensional surface, such as a screen or a page. Some examples would be: 12 15.2 0 –1

	0.01 98750
SFNode, MFNode	A node, here being defined as the content of a field. In this way nodes can be *nested*, placed one within another, almost like virtual Russian dolls. This capability adds enormous flexibility to VRML, as you'll see when we cover the Shape node.

This is only a partial list of the data types which might go into a field, but these will be all we'll be using for the next several chapters. Each of these basic types has a sister type which begins with the prefix "MF" rather than "SF". These prefixes determine how many values of the type may be provided within the field. Any field that has a data type that begins with "SF" specifies a *single field* value, that is, just one value of the type should be supplied. However, if the data type begins with "MF", the field specifies a *multiple field* value. A multiple field value begins with a bracket, and contains as many field values as are required, then closes with a bracket.

For example, an MFInt32 would define a field with multiple SFInt32 values, like this:

```
[ 32, 10000, -98, 7658, 0 ]  # MFInt32
```

And an MFColor field might look something like this:

```
[ 0 0 0, 1 0.5 0, 0.76 0.85 1, 0.98 0 1 ] # MFColor
```

The Shape of Things to Come

With those basics behind us, we're ready to get down to business.

In the beginning is the simplest VRML file, which contains nothing but the header. If you loaded that into a VRML browser, you'd see nothing at all. The header doesn't create anything, so of course nothing is visible. In order to create a visible object in VRML, you must use the Shape node. The Shape node connects the geometry of an object – that is, its contour – with the surface appearance of the object. Remember, in the primer we learned that objects are defined from points, and the surfaces of those objects are defined separately from the points that compose them. The Shape node is a way to connect appearance with form. 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
      }
```

The appearance and geometry fields, supplied with nodes of their own, define the visible characteristics and form of a visible object. We'll leave the appearance field alone – that's the subject of the next chapter – and concern ourselves with some of the built-in forms of VRML.

This Baby's Loaded with Features!

Just as an automobile is often loaded with "standard" features, VRML defines four built-in forms. These forms are the most commons ones found in 3D, hence, they were made essential parts of the language. The simplest and most pure of these is the Sphere, which has a very simple definition, as we've seen:

```
      Sphere {          # definition of the Sphere node
            radius      # SFFloat, defaults to 1.0
      }
```

The Sphere node has just a single field, radius. If you define a Sphere node, and don't specify the radius field, it uses a default value of 1.0. Most fields have default values, and if you're happy with a field's default value, there's no need to specify that field when defining a node. The node:

```
      Sphere {          # no field, uses default
      }
```

is the same as:

```
      Sphere {          # defines the field, default value
            radius 1.0
      }
```

You should be very careful when you assume that the default behavior of a field is what you expect it to be; chances are you'll be right, but it never hurts to check – and you can check the VRML 2.0 specification given in an appendix of this book if you're unsure.

To put the Sphere node together with the Shape node – or rather, to put the Sphere node within the Shape node, to produce a visible Sphere, the complete VRML file would look something like this:

```
      #VRML V2.0 utf8
      # This is the first visible example
      # A visible Sphere
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Sphere {                # we define a sphere
      radius 1.0  # default
}                       # end Sphere
      }                 # end Shape, end file
```

WARNING: This is a simplification of the actual VRML code. Don't expect to type this in verbatim, and have it work. However, the examples on the CD-ROM do work.

When you pop that into a VRML browser (either Netscape/Cosmo Player or Internet Explorer/Worldview will wok fine) you'll a sphere. As you can see by spinning it (use the "Spin" control on Cosmo or the "Study" control on Worldview) it's a true sphere; difficult to tell which end is up.

Next, the Box node, which, as befits its name, describes a box. The definition is almost as simple as for the Sphere:

```
      Box {       # definition of the Box node
            size        # SFVec3f, defaults to 2.0 2.0 2.0
      }
```

In its default state, the Box node actually defines a cube of one unit per side. The field size takes 3 values, for the length, height and depth. The VRML for that might look like this:

```
      #VRML V2.0 utf8
      # This is the second visible example
      # A visible Box
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Box {                   # we define a Box
      size 1.0 1.0 1.0  # our cube
}                       # end Box
      }                 # end Shape, end file
```

When you pop this one into a VRML browser, you'll a cube. But it doesn't have to be. Each of the three values in the size field can be widely different. Let's say we wanted to make a floor – common enough in virtual reality. We'd want something wide and deep, but not very tall. Let's say it's 10 meters wide, by 10 meters deep by 10 centimeters tall. That would look like this:

```
      #VRML V2.0 utf8
      # This is the third visible example
      # A visible Box floor
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Box {                   # we define a Box
      size 10 .01 10.0  # floor-shaped
}                       # end Box
      }                 # end Shape, end file
```

As you see, this really does look like a floor.

You can set the values in the size field of the Box node to any reasonable values, and create a very wide variety of shapes.

Next, the Cone node creates the shape we're familiar with from ice cream, witches' hats and, oh yeah, the VRML logo. It's got four fields, which allow us to play a little bit with its shape:

```
      Cone {                        # definition of the Cone node
            bottomRadius            # SFFloat, default 1.0
            height                  # SFFloat, default 2.0
            side                    # SFBool, default TRUE
            bottom                  # SFBool, default TRUE
      }
```

Why so many fields? The first two, bottomRadius and height actually determine the dimensions, while side and bottom turn on or off the two parts of the Cone. A basic Cone description would look something like this:

```
      #VRML V2.0 utf8
      # This is the fourth visible example
      # A visible Cone
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Cone {                  # we define a Cone
      bottomRadius 1.0 # same as default
      height 2.0  # same as default
}                       # end Cone
      }                 # end Shape, end file
```

This will create pretty much what we'd expect to see.

The side and bottom fields come into play when we want to make a Cone with only one of its parts. The part that we don't want – say the bottom – we turn off by setting the field value to FALSE. Here's the same example again, with the bottom removed:

```
      #VRML V2.0 utf8
      # This is the fifth visible example
      # A visible Cone
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Cone {                  # we define a Cone
      bottomRadius 1.0 # same as default
      height 2.0  # same as default
      bottom FALSE      # No bottom on cone
}                       # end Cone
      }                 # end Shape, end file
```

When we examine it, we'll see that, yes, indeed, it is bottomless.

If we do the opposite, and leave the bottom on while removing the side, the VRML will look like this:

```
      #VRML V2.0 utf8
      # This is the sixth visible example
      # A visible Cone
      Shape {           # define the Shape node
```

```
            geometry    # inside the geometry field
Cone {                  # we define a Cone
      bottomRadius 1.0 # same as default
      height 2.0  # same as default
      side FALSE  # No side on cone
}                       # end Cone
      }                 # end Shape, end file
```

It'll look very different indeed. In fact, you can't see it unless you spin it a bit.

Our final built-in shape is the Cylinder. The Cylinder has a definition very much like the Cone, with the addition of one more field:

```
      Cylinder {        # definition of the Cylinder node
            bottom                  # SFBool, default TRUE
            height                  # SFFloat, default 2.0
            radius                  # SFFloat, default 1.0
            side                    # SFBool, default TRUE
            top                     # SFBool, default TRUE
      }
```

The basic Cylinder definition might look like this:

```
      #VRML V2.0 utf8
      # This is the seventh visible example
      # A visible Cylinder
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Cylinder {        # we define a Cylinder
      height 2.0  # same as default
      radius 1.0  # same as default
}                       # end Cylinder
      }                 # end Shape, end file
```

Which produces a thoroughly ordinary looking object.

However, as with the Cone, you can turn off the parts of the Cylinder with the bottom, top and side fields. For example, here's the same Cylinder, but with no top or bottom:

```
      #VRML V2.0 utf8
      # This is the eighth visible example
      # A visible Cylinder
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Cylinder {        # we define a Cylinder
      height 2.0  # same as default
      radius 1.0  # same as default
      top FALSE         # no top
      bottom FALSE      # no bottom
}                       # end Cylinder
      }                 # end Shape, end file
```

So yes, you can see right through it.

In a more perverse scenario, you could leave the ends on and get rid of the body of the Cylinder:

```
      #VRML V2.0 utf8
      # This is the ninth visible example
      # A visible Cylinder
      Shape {           # define the Shape node
            geometry    # inside the geometry field
Cylinder {        # we define a Cylinder
      height 2.0  # same as default
      radius 1.0  # same as default
      side FALSE  # no side? Weird!
}                       # end Cylinder
      }                 # end Shape, end file
```

And there it is – but remember you'll only ever see the top or the bottom, never both at the same time.

Those are the basic built-in shapes in VRML, and the basic syntax for the Shape node which must be used with them. But you must be getting just a little bit tired at looking at the same, dull, white objects every time. Why are they this color? Come and see…