Chapter 13: Excellent References

Pastiche

The best Web sites integrate content from many different places into a cohesive whole; the power of the Web lies in its ability to acquire things by reference. You want a picture on your pages – you don't need a copy of it, you only need to point to it. While this raises all sorts of issues about who owns what on the Web – and where it can be used – the Web has also become the ultimate engine of pastiche. Some of its corners, like the acerbic *Suck* (www.suck.com), specialize in appropriation, critiquing the whole of the Web – sometimes even all of human culture – with little bits stolen from the whole cloth.

Think about the "Looks best with…" browser buttons you see on so many Web pages; these buttons contain art from other Web sites; sometimes the art resides on the local Web site, but other times it appears by reference. In fact, all images on all Web pages have been brought in as references to the documents that contain the image data; none of it resides within the HTML document.

Why do this? It's very difficult to define an image within a text document; like trying to mix oil and water. Inclusion by reference solves the problem nicely, blending images and text together without forcing them to mix unnaturally in the same document.

We do something very similar in VRML when we reference texture map images or movies using the url field of the ImageTexture and MovieTexture nodes; we point to the image, and the browser loads it – but we don't actually have to put the image inside the VRML file. It's enough to tell the browser how to get the image. But we've gone even further in VRML; it's possible to load a VRML world from within another VRML world – through a mechanism known as *inline file inclusion*.

It's Important to Feel Included

Inline file inclusion creates a single VRML world from multiple component VRML worlds; each of these must be complete, legal VRML worlds on their own. The "main" world gathers up the inline worlds into a coherent whole – but each constituent world is a whole world in its own right. The Inline node of VRML allows you to define the connection between a world and its components:

```
Inline {                # definition of Inline node
      url []            # MFString, mult. values
      bboxCenter  # SFVec3f
      bboxSize          # SFVec3f
}
```

Let's go through a simple example of how this works. We'll start off with a world that creates a green sphere:

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

This gives us our expected green Sphere; it also implies that this file is a complete stand-alone VRML world.

Now let's create a world that has nothing but an Inline node which references our first example:

```
#VRML V2.0 utf8
# This is the second example on inline worlds
Inline {
url [ "14_1.wrl" ] # reference first example
}
```

When it loads, we see exactly the same thing.

You may have noticed – briefly – that a wire frame of a cube appeared within the browser before the object popped up. This wire frame is known as a *bounding box*; this box lets you know that something will be popping up in place of the box in just a moment – as soon as the referenced object gets fetched, parsed and rendered. If the object has a complicated, long definition, the bounding box could remain for some time – perhaps even a few minutes – so it can be an important signal to you of the impending appearance of new objects in a VRML world.

The referenced VRML world can be of any complexity, from the simple to the dense; in the following example we reference the 80-sided sphere we created in chapter 10 – for this chapter we've renamed it eighty_sides.wrl – and find that it looks remarkably similar to the previous example:

```
#VRML V2.0 utf8
# This is the third example on inline worlds
Inline {
url [ "eighty_sides.wrl" ] # from ch. 10
}
```

Here we see that it takes a bit longer for the model to load – so we see the bounding box a bit longer as well – but soon the shape is loaded.

You can put the Inline node anywhere you might put a VRML object definition; that means we can place Inline nodes inside the children fields of Transform and Anchor nodes. Here we take our complex object and link it to the VRML Repository:

```
#VRML V2.0 utf8
# This is the fourth example on inline worlds
Anchor {
      children [ # list of children
            Inline {  # inlined child
            url [ "eighty_sides.wrl" ]
      }
]
url [ "http://www.sdsc.edu/vrml/" ] # Repository
}
```

It's the same object, now linked to the Repository.

Not In My Backyard

If once is great, why not do it again? The Inline node allows you to place an object within a world several times (there are other ways to do this, too, which we'll cover in a later chapter). Let's say that we wanted to have a world with three of these eighty-sided spheres. We'd need to put at least two of them inside of Transform nodes, so it might look like this:

```
      #VRML V2.0 utf8
      # This is the fifth example on inline worlds
      # First object definition
      Inline {  # inlined world
      url [ "eighty_sides.wrl" ]
}
      # Second and third objects
      Transform {
            children [ # list of children
                  Inline {  # inlined child
                  url [ "eighty_sides.wrl" ]
            }
            # Third object
            Transform {
                  children [
                              Inline {  # inlined child
                              url [ "eighty_sides.wrl" ]
                        }
                  ]
                  translation 10 0 0
            }
      ]
      translation 10 0 0
      }
```

This creates three objects which look the same, spread out legthwise.

We used Transform nodes to keep the objects distinct; but what happens when you load a world with its own Transform nodes? They become active in the local coordinate space of the world they get loaded into – so if they get loaded into a Transform node, they take on all of the cumulative transformations of the world they're loaded into plus whatever

transformations have been defined local to their own world. Here's a simple world that creates a purple Sphere that's a bit "up in the air", above the origin of the Y axis:

```
#VRML V2.0 utf8
# This is the sixth example on inline worlds
# Purple Cube floating above Y origin
Transform {
      children [ # list of children
            Shape {  # visible form
            appearance Appearance {
                  material Material {
                        diffuseColor 1 0 1
                  }
            }
            geometry Sphere { }
      }
]
translation 0 2 0
}
```

But let's Inline it into a world with a blue Cone at the origin:

```
#VRML V2.0 utf8
# This is the seventh example on inline worlds
# Blue Cone floating at origin
Shape {  # visible form
appearance Appearance {
      material Material {
            diffuseColor 0 0 1
      }
}
geometry Cone { }
}
      # First object definition
      Inline {  # inlined world
      url [ "14_6.wrl" ]
}
```

You may have seen the bounding box show up – very quickly – in the midst of the Cone. Why did it pop up there, and not where the Sphere later appeared? The Inline node in this example is in the global coordinate space – outside of any Transform nodes – so the browser assumed that the Inline object would pop up at the origin; but the object, inside of its own local Transform node, didn't do that. So the effect, rather than being helpful, looks rather clumsy.

How do we get around this? By using the bboxCenter and bboxSize fields of the Inline node. These fields allow you to explicitly set the center point and size of the bounding box generated when the browser loads an Inline world. In the example above, we know that the center of bounding box will need to be 2 units in Y higher than the origin of the coordinate system – because the local coordinate system of the sixth example creates a translation of two units. Our amended example would look like this:

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

```
      # This is the eighth example on inline worlds
      # Blue Cone floating at origin
      Shape {  # visible form
      appearance Appearance {
            material Material {
                  diffuseColor 0 0 1
            }
      }
      geometry Cone { }
}
      # First object definition
      Inline {  # inlined world
      url [ "14_6.wrl" ]
      bboxCenter 0 2 0 # move it up a bit
}
```

Now when the scene loads, the bounding box is in the correct place.

But the bounding box isn't quite the same size as the Sphere; we'd need to modify the bboxSize field a bit so that we get a better fit:

```
      #VRML V2.0 utf8
      # This is the ninth example on inline worlds
      # Blue Cone floating at origin
      Shape {  # visible form
      appearance Appearance {
            material Material {
                  diffuseColor 0 0 1
            }
      }
      geometry Cone { }
}
      # First object definition
      Inline {  # inlined world
      url [ "14_6.wrl" ]
      bboxCenter 0 2 0 # move it up a bit
      bboxSize 1.414 1.414 1.414 # correct for sphere
}
```

Now we get a bounding box that correctly reflects the position and size of the world referenced and loaded through the Inline node.

Lazy on the Details

When used in conjunction with the LOD node, the Inline node allows you do take advantage of something known as *lazy loading*. Lazy loading means that a browser won't load an Inline object until it has become visible; if that Inline node resides inside of the level field of an LOD node, the Inline must come into range before it gets loaded by the browser. Why is that important? Complex objects have detailed – and large – definitions, that can quickly bog down a modem connection. When LOD and Inline work together to create lazy loading, you only load those versions of an object that you can see.

As you get closer to the object – and pass a transition point between levels of detail – you'll load the more complicated – and bandwidth-intensive – version.

Here's the final example from chapter 12, broken into four separate files, starting with the most complex version of the object:

```
#VRML V2.0 utf8
# This is the tenth example on level-of-detail
Shape {           #  Create a visible shape
      appearance Appearance {
            texture ImageTexture {
                  url [ "worldmap.jpg" ]
            }
      }
      geometry Sphere { }
}
```

Moving to the simpler:

```
#VRML V2.0 utf8
# This is the eleventh example on level-of-detail
Shape {           #  Create a visible shape
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 1 1
            }
      }
      geometry Sphere { }
}
```

And then the simplest:

```
#VRML V2.0 utf8
# This is the twelfth example on level-of-detail
Shape {           #  Create a visible shape
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 1 1
            }
      }
      geometry Box { }
}
```

Here's the main world that brings them all in, under an LOD node:

```
#VRML V2.0 utf8
# This is the thirteenth example on level-of-detail
LOD {
      level [           # three representations
            # the most complicated goes first
            Inline {  # inlined world
            url [ "14_10.wrl" ]
}
            # followed by the less complicated
            Inline {  # inlined world
```

```
                    url [ "14_11.wrl" ]
}
                    # followed by the simplest
                    Inline {  # inlined world
                    url [ "14_12.wrl" ]
}
            ] # end list of representations
            range [ 50, 200 ] # switch at 50 and 200 units
      }
```

When we load the world, we see the most complicated object, because we're rather close to it. However, as we back away, we see the LOD transitions accompanied by Inline fetches.

Ready, Willing, and Able

That concludes the portion of this book that concerns itself with the visible features of VRML; a few things we have yet to cover – text, cameras and lights – we'll pick up in the next chapters. But you've already learned as much as what could be done with VRML 1.0; everything from here on in is new territory – the undiscovered country of interactivity. But before we proceed, let's take at look at how four women have fundamentally redefined our expectations for interactivity in general, and virtual reality in specific…