Chapter 10: Wallpapering Cyberspace

Covering up the Sins of Omission

People often complain that virtual worlds don't look real; they feel artificial, radically simplified, almost caricatures of their real-world counterparts. The world-at-large, crowded with shapes, figures and colors, presents a challenge to the virtual world creator. How can we get the intensity of detail in the real into the virtual? If we modeled every feature of the real world, the nooks and crannies of every surface, the gentle shadings of textiles, and the glow of muted lights, we soon find the computer utterly overloaded; real-time computer graphics – like VRML – need to be simple; that simplicity ensures their speed.

These virtual sins of omission – and to err on the side of performance is hardly a sin at all – have an antidote in the *texture map*. The texture map, rather like wallpaper in a decrepit building, can transform the ugliest of virtual spaces into a unique and aesthetic place. Although none of us has seen a wall painted with an intricate floral pattern, most of us have seen flower-flocked wallpaper adorning even the humblest of homes. A cheap solution to the eternal war between beauty and utility, wallpaper buys us a lot for very little. And – with a little ingenuity – a wallpaper pattern can become an element of design, carefully tiled to set off other elements of a space's design. All of this, and much more, can be done with the texture map; it's the foundation of virtual objects that look real.

*DOOM*ed?

Computer games have done more to make people aware of the incredible potential of 3D technology than any other type of application. In 1994, when the folks from id Software introduced *DOOM*, gaming changed forever; thumb-twitch titles would have to treat the user to realistic, blazingly fast 3D worlds - nothing less would do. Many folks noted that DOOM ran fast even on slow computers. How could this be? The secret lay in DOOM's clever use of texture maps to create the "feeling" of being in a realistic space. All of the scenes in DOOM are very simple – no more than about twenty polygons can be seen at any one time – but these worlds, richly textured, communicate an impressive feeling of place, an incredible example of an effective trade-off between detail and performance.

Can VRML create the kinds of environments you see in DOOM? Absolutely! Can VRML browsers get the same kind of speed you see in DOOM? That depends on a whole host of variables; however, one company, Newfire, has released a VRML browser plug-in – *Torch* - designed around the DOOM gaming engine. In simplistic, heavily textured worlds similar to DOOM, it boasts incredible performance – a perfect foundation for Web-based VRML gaming.

So let's learn how we can make DOOM-like worlds – and other complex objects – using just a few new nodes.

The Appearance of Wallpaper

We've been using the Appearance node consistently since we introduced it in chapter 8; the Shape node requires it as the value in its appearance field. Once again, here's the node and its fields:

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

So far, we've been using the material field of the Appearance node, ignoring the texture and textureTransform fields. We've been "painting" objects, giving them surface qualities through the values provided in a Material node supplied as the value to the material field. Now we want to apply images – texture maps - to these forms, rather than surface colors, so we'll now use the texture field of the Appearance node, which takes as its input value, one of two nodes – ImageTexture or MovieTexture.

The ImageTexture node applies a still image to a form, while the MovieTexture node allows you to take an MPEG movie and use that as "moving wallpaper". We'll discuss MovieTexture a little further along, once we've covered the basics with ImageTexture. Here's the node's definition:

```
ImageTexture {          # definition of ImageTexture node
      url []            # MFString, mult. values
      repeatS           # SFBool
      repeatT           # SFBool
}
```

Of these fields, the only one we need right now – to start off with a few examples – is the url field. We'll see the url field in many different VRML nodes, so let's spend a moment discussing how to use it.

The URL of VRML

The Universal Resource Locator, or URL, as it's more commonly known, defines a unique path across the Internet to a document somewhere within it. It's broken up into three parts: access method, host name, and file path and name.

The first part of the URL defines the *access method* which should be used to fetch the information. While "http:" – the protocol of the Web – occurs with greatest frequency, others, including "ftp:", "gopher:" and "wais:" can also be found. This means you have a couple of different ways to get to information across the Internet. No web server where your documents are? Use file transfer protocol – FTP – instead. It's just as reliable; before the Web, FTP soaked up most of the Internet's bandwidth. It's also even more ubiquitous than the Web; many sites that don't offer Web access will offer FTP access – opening up another world of documents.

The middle part of the URL defines the *host name* of the machine where the document resides. This hostname always begins with a double forward-slash, separating it from the access method. For instance, my home page resides on www.hyperreal.com, so the host name portion of a URL to my home page would be "//www.hyperreal.com".

The tail end of the URL defines the *path* and *name* of the document. This part of the URL is separated from the host name by a single forward-slash. It can be arbitrarily long or short – in fact, it doesn't have to be there at all. The path to my own home page is ~mpesce (it means user "mpesce"), while my home page is named "index.html". So the complete path and name to my home page is "/~mpesce/index.html"; note that another single-slash comes between the path and the document name.

When you put them all together – access method, host name, path and document name - you get a *fully-formed* URL. That means the URL is explicit – it defines one and only one way to get there from here. But it's not always necessary – or even a good idea – to give a fully-formed URL every time you supply a URL within a VRML world. The *relative* URL allows you to "assume" a lot of information about the URL, so that you don't have to keep track of the exact path to a document, its hostname, and so forth – which is really quite useful, because these things have a tendency to change when you're not looking – breaking all of your URLs.

The relative URL is simplicity itself. Let's say that you've loaded a VRML world from the following URL: http://www.fakeurl.com/test/myworld.wrl. All relative URLs within the VRML world myworld.wrl would take their access method, host name and path from the URL used to load myworld.wrl. So if we have a relative URL, that is, a url field with the value "mybox.jpg" (a JPEG image), that would be the same as if we had typed in http://www.fakeurl.com/test/mybox.jpg – but without all of that typing.

NOTE: The relative URL implies that a file is on the same host and in the same path as the file it's relative to. If that isn't so, you can't use a relative URL; you must give a fully-formed URL.

The url field is always defined as a data type of MFString. Why have multiple values in a url field? It's a fail-safe mechanism that's essential to VRML 2.0. The first URL given in the url field is the *preferred* URL – the browser will look here first. If the browser can't locate the document using the preferred URL, the browser will drop back to the next URL in the list; if that fails, it will try the next, and so on – until it either succeeds or runs out of URLs. That means that if you have a mission-critical VRML world, you can keep primary and secondary (even tertiary) copies of all objects referenced by URL; if the network goes down, or perhaps the Pacific Rim is clogging all of your circuits, you'll still be in business.

You don't need to have multiple URLs in the url field; you must have at least one, but any others are not required – it's entirely up to you. Here's some sample values for the url field, showing you how this works:

```
# This is just one URL
url [ "http://www.primaryurl.com/test/test.jpg" ]
```

```
# The backup points to a machine in Australia
url [ "http://www.primaryurl.com/test/test.jpg",
      "http://www.secondaryurl.au.co/test/test.jpg" ]
# And finally, backups in Australia and France
url [ "http://www.primaryurl.com/test/test.jpg",
      "http://www.secondaryurl.au.co/test/test.jpg",
      "http://www.tertiaryurl.fr.co/test/test.jpg" ]
```

Notice that each of the strings is separated by commas from each preceding string.

It's All About Image

VRML has built-in support for two types of image files, JPEG – a very common format, and the still-rare PNG. Most image programs, including Adobe *PhotoShop*, will create JPEG images for you – may JPEG converters exist as well, and many are free. JPEGs are compact, and rich in color, but they lack *transparency*. Transparency allows the surface underneath a texture map to appear as part of the surface – which can be a very powerful way to build composite images from a combination of surfaces and textures. GIF format does support transparency, but it isn't required to be supported in a VRML browser. PNG – the successor to the ubiquitous GIF format – does include support for transparency, but it's not yet wildly popular, even though your browser supports it.

That said, many browsers do support GIF image types, and some support TIFF and BMP images, particularly browsers running under Windows 95 or NT. The safe bets are JPEG and PNG; beyond that, you're risking incompatibilities between your images and someone else's browser.

Making the World Round…and Square

Each of the built-in shapes in VRML has a *default wrap*; that is, a predefined way that a texture map will drape itself over the form of the built-in shape. Let's being with Sphere node. It has a wrap that favors images that look like maps! For example, open worldmap.jpg.

The file worldmap.jpg is a JPEG image, a Mercator projection of the Earth's surface – we've all seen it before on classroom walls, in *National Geographic*, and on the news. But the amazing thing about the default wrap for the Sphere node is that it takes a

Mercator projection and wraps it into something that removes the distortion inherent in that projection. Here's the VRML for that:

```
      #VRML V2.0 utf8
      # This is the first example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
}
            }
            geometry Sphere { }     # Create form, use defaults
      }
```

Now let's take a peek at our model of the world. As you can now see, texture maps are a great way to buy a lot of visual richness for very little programming; this is just seven lines of VRML, yet it does something valuable!

The Box has a very different type of default wrap; instead of wrapping the whole image across all six faces, it wraps the image six times – once for each face. The VRML looks almost exactly the same:

```
      #VRML V2.0 utf8
      # This is the second example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
}
            }
            geometry Box { }  # Create form, use defaults
      }
```

Now we have a cube with a map on every side.

This could be a lot more useful – on a cube – if the texture map looked like brick, or stone. Then it could be used as flooring, or a wall – the beginnings of a DOOM-like environment. Here's a bit of brick:

Let's see if we can make a brick floor by making a Box that's long, wide, and thin:

```
      #VRML V2.0 utf8
      # This is the third example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "brick.jpg" ]
```

```
}
            }
            geometry Box { size 10 0.01 10 }    # floor shape
      }
```

And we should now see something suitable for our own dungeon (remember to spin in a bit when you enter the world; it's so thin you won't see anything right away):

Making World Soda

It's wonderfully easy to make items like soda cans using texture maps with the Cylinder node. Here's the basic formula to make "World Soda", which uses the worldmap.jpg image:

```
      #VRML V2.0 utf8
      # This is the fourth example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
}
            }
            geometry Cylinder { radius 0.5 }    # soda can
      }
```

This looks pretty cool, but the top and bottom of the soda can are covered with our image – and that looks rather unprofessional.

How do we get rid of this texturing on the top and bottom of the soda can? During our initial discussion of the Cylinder node, we covered the top, side and bottom fields; if these were set to FALSE, they weren't drawn. So the secret is to create the Cylinder twice; once with the texture map, but top and bottom FALSE, then a second time, with side FALSE but creating an aluminum like finish. Here's how that looks:

```
      #VRML V2.0 utf8
      # This is the fifth example on textures
      # Create the side of the can
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
}
            }
            geometry Cylinder { radius 0.5 # soda can
                  top FALSE         # no top
                  bottom FALSE   # or bottom
}
      }
      # Create the top and bottom of the can
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  material Material {     # Create material
                        diffuseColor 0.5 0.5 0.5 # gray
```

```
                        shininess 0.9 # very shiny
                        specularColor 1 1 1 # reflects white
}
            }
            geometry Cylinder { radius 0.5 # soda can
                  side FALSE  # no side
}
      }
```

Now we have something that looks a lot more like a soda can.

All it needs is a pull-cap – but you could supply that, too. Take an image of a pull-tap, and texture map it onto the top of the can.

Wrapping Up a Mess

So far these default wraps have worked out just fine; we've gotten something interesting and useful every time. But, once you move to more complex forms, texture mapping gets a lot harder to do well. How would you wrap that world map to a teapot? Or a telephone? It's not exactly straightforward, even for very simple forms. For example, let's apply that world map to the pyramid that we created in the last chapter. At its simplest, it might look like this:

```
      #VRML V2.0 utf8
      # This is the sixth example on texture maps
      Shape {
            appearance Appearance {
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
                  }
            }
            geometry IndexedFaceSet {     # complex shape!
coord Coordinate {            # Coordinate node
                        point [           # MFVec3f, mult. Values
                              0 0 0,      # index zero
                              1 0 0,      # index one
                              1 0 -1,     # index two
                              0 0 -1,     # index three
                              0.5 0.5 -0.5 # index four
                        ]
                  }
                  coordIndex [ 0, 3, 2, 1, -1, # bottom down
                                    0, 1, 4, -1, # front
2, 3, 4, -1,  # back
1, 2, 4, -1, # right
3, 0, 4, ] # left
                  solid TRUE  # this is a solid shape
            }
      }
```

Now take a look at it. In this case, the browser has made the correct assumptions, for the top of the pyramid, but take a look at the base. The texture mapping on the underside of the pyramid is reversed right-left from what we might expect. Without adequate

guidance, the browser presumed this to be the correct wrap for the object. How do we fix this? By using something known as *texture coordinates*. Texture coordinates are the "virtual tacks" that allow you to have precise control over which parts of an image are attached to which surfaces of an object. Using texture coordinates, you define the portions of the image that are to be attached to a surface; the browser uses this information instead of its own assumptions, and the results are picture-perfect. Here's the definition of the TextureCoordinate node:

```
TextureCoordinate {     # def. of TextureCoordinate
      point []          # MFVec2f, mult. values
}
```

The values defines in the point field of the TextureCoordinate node have a data type of MFVec2f, that is, they're two-dimensional points. Why? Although texture coordinates are used in a three-dimensional world, images themselves are two-dimensional objects; slicing them up into sections only requires two values, known as *S* and *T*. (I don't think anyone knows why they're called S and T.)

Any image, no matter how big or how small, how wide or how tall, is defined as having a width, that is, S, of 1.0, and a height, or T, of 1.0. That means we don't have to fiddle with pixel values that may be nearly impossible to determine – we can just take the whole size of an image, and work from there.

The lower left hand corner of the image is the origin of the texture coordinate system, having a value of (0, 0), while the upper right hand corner describes the full extent of the image, with a value of (1, 1). Everything area in the image falls between these two values, so if we wanted to describe a set of four texture coordinate values that described the entire area of the image, the TextureCoordinate node would look like this:

```
TextureCoordinate {     # whole image
      point [ 0 0,      # lower left corner
              1 0,   # lower right corner
              1 1,      # upper right corner
              0 1 ]  # upper left corner
}
```

Once we've defined the TextureCoordinate, where does it get used? Well, there are a few fields in the IndexedFaceSet node we haven't covered:

```
IndexedFaceSet {  # IndexedFaceSet definition
      color             # SFNode
      coord             # SFNode
      normal                  # SFNode
      texCoord                # SFNode
      ccw                     # SFBool
      colorIndex []           # MFInt32
      colorPerVertex          # SFBool
      convex                  # SFBool
      coordIndex []           # MFInt32
      creaseAngle       # SFFloat
      normalIndex []    # MFInt32
```

```
        normalPerVertex   # SFBool
        solid             # SFBool
        texCoordIndex []  # MFInt32
    }
```

The texCoord and texCoordIndex fields work together to produce the texture wrapping for a complex object. A TextureCoordinate node is supplied as the value for the texCoord field, and – as with point values in the Coordinate node supplied to the coord field – the values in the point field of the TextureCoordinate node have reference index numbers, which, once again, begin with zero. That means that our TextureCoordinate node really looks like this:

```
    TextureCoordinate {     # whole image
        point [ 0 0,        # lower left corner, index zero
                1 0,    # lower right corner, index one
                1 1,      # upper right corner, index two
                0 1 ]   # upper left corner, index three
    }
```

The texCoordIndex field defines the one-to-one mapping between the 3D points used to define surfaces – given in the coordIndex field – and the 2D points used to map images onto those surfaces. Before we wade in hip-deep, we'll start with a very simple shape – the square from the beginning of the last lesson. But we'll map it with this TextureCoordinate node. That might look like this:

```
    #VRML V2.0 utf8
    # This is the seventh example on texture maps
    Shape {
        appearance Appearance {
            texture ImageTexture {
                url [ "worldmap.jpg" ]
            }
        }
        geometry IndexedFaceSet {
            coord Coordinate {            # define points
                point [
                    0 0 0,      # index zero
                    1 0 0,      # index one
                    1 1 0,      # index two
                    0 1 0 # index three
                ]
            }
            coordIndex [ 0, 1, 2, 3 ] # make square
            texCoord TextureCoordinate{ #  whole image
        point [ 0 0,        # index zero
                           1 0,    # index one
                           1 1,      # index two
                           0 1 ]   # index three
            }
            texCoordIndex [ 0, 1, 2, 3 ] # map square
        }
    }
```

We should see a square with the entire image mapped across it.

But if we change the ordering of the values in the texCoordIndex field – to a clockwise ordering rather than counter-clockwise, we'll reverse the image. Here's the code:

```
#VRML V2.0 utf8
# This is the eighth example on texture maps
Shape {
      appearance Appearance {
            texture ImageTexture {
                  url [ "worldmap.jpg" ]
            }
      }
      geometry IndexedFaceSet {
            coord Coordinate {            # define points
                  point [
                        0 0 0,      # index zero
                        1 0 0,      # index one
                        1 1 0,      # index two
                        0 1 0 # index three
                  ]
            }
            coordIndex [ 0, 1, 2, 3 ] # make square
            texCoord TextureCoordinate{ #  whole image
      point [ 0 0,      # index zero
                           1 0,   # index one
                           1 1,     # index two
                           0 1 ]  # index three
            }
            texCoordIndex [ 1, 0, 3, 2 ] # rev. square
      }
}
```

And we can see that the image is now reversed. It's reversed because the texCoordIndex field specifies that the lower right point of the image is attached to the lower left point of the surface, the lower left point of the image to the lower right point of the surface, and so on. We could easily flip the image over:

```
#VRML V2.0 utf8
# This is the ninth example on texture maps
Shape {
      appearance Appearance {
            texture ImageTexture {
                  url [ "worldmap.jpg" ]
            }
      }
      geometry IndexedFaceSet {
            coord Coordinate {            # define points
                  point [
                        0 0 0,      # index zero
                        1 0 0,      # index one
                        1 1 0,      # index two
                        0 1 0 # index three
                  ]
            }
            coordIndex [ 0, 1, 2, 3 ] # make square
            texCoord TextureCoordinate{ #  whole image
```

```
            point [ 0 0,      # index zero
                                 1 0,   # index one
                                 1 1,     # index two
                                 0 1 ]  # index three
                  }
                  texCoordIndex [ 3, 2, 1, 0 ] # flip square
            }
      }
```

By attaching the bottom points of the image to the top points of the surface, we get an upside-down image.

But the fun's only just beginning…

Slicing the Pie

We've been working with whole images; what if we want just a part of the image to map onto a surface? All we need to do is use a different set of texture coordinates, and slice the image into any sections that are useful to us. For example, the North American continent occupies the upper left-hand quadrant of the map. That's the area where S is between 0 and 0.5 (halfway), and T is between 0.5 and 1. That would give us the following TextureCoordinate node:

```
TextureCoordinate {     # upper left-hand side
      point [ 0 0.5,    # lower left corner
               0.5 0.5, # lower right corner,
               0.5 1,   # upper right corner
               0 1 ]  # upper left corner
}
```

When we integrate that into the code from example seven:

```
#VRML V2.0 utf8

# This is the tenth example on texture maps

Shape {

      appearance Appearance {

            texture ImageTexture {

                  url [ "worldmap.jpg" ]

            }

      }

      geometry IndexedFaceSet {

            coord Coordinate {            # define points

                  point [
```

```
                        0 0 0,      # index zero
                        1 0 0,      # index one
                        1 1 0,      # index two
                        0 1 0 # index three
                  ]
            }
            coordIndex [ 0, 1, 2, 3 ] # make square
            texCoord TextureCoordinate {  # upper lt.
      point [ 0 0.5,    # lower left corner
               0.5 0.5, # lower right corner,
               0.5 1,   # upper right corner
               0 1 ]  # upper left corner
}
            texCoordIndex [ 0, 1, 2, 3 ] # map square
      }
}
```

Which gives us just what we were looking for, the upper left hand side.

Still, we've got a bit from South America and Africa; let's adjust the TextureCoordinate values once more, and zoom in a bit:

```
#VRML V2.0 utf8
# This is the eleventh example on texture maps
Shape {
      appearance Appearance {
            texture ImageTexture {
                  url [ "worldmap.jpg" ]
            }
      }
      geometry IndexedFaceSet {
            coord Coordinate {            # define points
                  point [
                        0 0 0,      # index zero
                        1 0 0,      # index one
                        1 1 0,      # index two
                        0 1 0 # index three
                  ]
            }
            coordIndex [ 0, 1, 2, 3 ] # make square
```

```
                  texCoord TextureCoordinate {  # upper lt.
            point [ 0 0.6,    # lower left corner
                     0.4 0.6, # lower right corner,
                     0.4 1,   # upper right corner
                     0 1 ]  # upper left corner
      }
                  texCoordIndex [ 0, 1, 2, 3 ] # map square
            }
      }
```

And we've zoomed in quite a bit.

So let's return to our example with the pyramid. If we want to rely entirely on our own wraps, so that we can give explicit instructions to the browser on how to place the map on that object, we'd have to define five TextureCoordinate points, corresponding to the five points of the pyramid:

```
      TextureCoordinate {     # pyramid points
            point [ 0 0,      # lower left corner, index zero
                     1 0,   # lower right corner, index one
                     1 1,     # upper right corner, index two
                     0 1,   # upper left corner, index three
                     0.5 0.5 # center, index four
            ]
      }
```

Now let's return to example six, but this time, for every surface we define in the coordIndex field, we'll define a wrap for it in the texCoordIndex field:

```
      #VRML V2.0 utf8
      # This is the twelfth example on texture maps
      Shape {
            appearance Appearance {
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
                  }
            }
            geometry IndexedFaceSet {     # complex shape!
coord Coordinate {            # Coordinate node
                        point [           # MFVec3f, mult. Values
                              0 0 0,      # index zero
                              1 0 0,      # index one
                              1 0 -1,     # index two
                              0 0 -1,     # index three
                              0.5 0.5 -0.5 # index four
                        ]
                  }
                  coordIndex [ 0, 3, 2, 1, -1, # bottom down
                                    0, 1, 4, -1, # front
2, 3, 4, -1,  # back
1, 2, 4, -1, # right
3, 0, 4, ] # left
                  solid TRUE  # this is a solid shape
                  texCoord    TextureCoordinate {     # pyramid
            point [ 0 0,      # index zero
```

```
                  1 0,   # index one
                  1 1,     # index two
                  0 1,  # index three
                  0.5 0.5 # center, index four
            ]
                  }
                  texCoordIndex [ 1, 2, 3, 0, -1, # bottom
                                  0, 1, 4, -1, # front
2, 3, 4, -1,  # back
1, 2, 4, -1, # right
3, 0, 4, ] # left
            }
      }
```

Note the correspondence between the coordIndex and texCoordIndex fields. For every surface defined in coordIndex, there's a wrap defined in the texCoordIndex – all wraps are terminated with a –1, just as all surfaces are in coordIndex. The texCoordIndex field is almost a perfect carbon copy of the coordIndex field – with the exception of the first line, which defines the wrap for the bottom of the pyramid. Here we've change the wrap ordering, so that the bottom side is oriented correctly.

Another thing we've done was to cut the image into triangular sections for the sides of the pyramid; that's just as legal as any rectangular shape. In fact, you can cut an image into a very complex shape using texture coordinates, then lay that down precisely on any surface – but that's the domain of modeling tools or computer-generated VRML, because it gets very complicated very quickly.

Going Hollywood

Back at the opening to this chapter we learned that the images we attach to surfaces don't have to be static. It's possible to use MPEG movies as texture maps; these movies – well, they move. To specify an MPEG movie, you must use the MovieTexture node in place of the ImageTexture node. Here's the definition of MovieTexture:

```
      MovieTexture {          # definition of MovieTexture node
            loop              # SFBool
            speed       # SFFloat
            startTime         # SFTime
            stopTime          # SFTime
            url []            # MFString, mult. values
            repeatS           # SFBool
            repeatT           # SFBool
      }
```

We're getting a little ahead of ourselves here, so we won't cover the startTime or stopTime fields in much detail – as we come to program our worlds for interactivity, you'll come to understand how they're used. We'll learn how to setup a movie texture map so that it begins playing when we load the VRML world into our browser; for that, stopTime is always set to 0, while startTime is set to 1.

The speed field tells the browser how quickly you'd like to run the movie. The field defaults to a value of 1, or normal speed, so – unless you want some fancy effects – you can leave that alone. However, if you do want to watch the movie go by in slow-motion, you can set it to some number less than one but greater than zero. For example, a value of 0.5 would run the movie at half-speed. The opposite is also true; if speed is set to 2, then the movie runs at twice its normal speed.

The url field should contain the URL of an MPEG movie file – and MPEG is the only type supported in VRML 2.0. Some browsers may support other movie types – like QuickTime or AVI, but you're on your own if you use anything but MPEG.

Here's our basic Box, with an MPEG from NASA of Hurricane Andrew:

```
      #VRML V2.0 utf8
      # This is the thirteenth example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture MovieTexture {  # Create texture
                        loop TRUE         # play over and again
                        stopTime 0
                        startTime 1
                        url [ "Andrew.mpeg" ]
}
            }
            geometry Box { }  # Create form, use defaults
      }
```

Not only does it move, but it repeats over and over. That's because we set the loop field to a value of TRUE. But best of all, the same movie is being shown on all six sides of the Box, because of its default wrap.

You can slice and dice these MPEG movies just as if they were normal images; here's example twelve, which presents the pyramid correctly texture mapped, but this time, using the movie:

```
      #VRML V2.0 utf8
      # This is the fourteenth example on texture maps
      Shape {
            appearance Appearance {
                  texture MovieTexture {  # Create texture
                        loop TRUE         # play over and again
                        stopTime 0
                        startTime 1
                        url [ "Andrew.mpeg" ]
}
            }
            geometry IndexedFaceSet {     # complex shape!
coord Coordinate {            # Coordinate node
                        point [           # MFVec3f, mult. Values
                              0 0 0,      # index zero
                              1 0 0,      # index one
                              1 0 -1,     # index two
                              0 0 -1,     # index three
```

```
                              0.5 0.5 -0.5 # index four
                        ]
                  }
                  coordIndex [ 0, 3, 2, 1, -1, # bottom down
                                    0, 1, 4, -1, # front
2, 3, 4, -1,  # back
1, 2, 4, -1, # right
3, 0, 4, ] # left
                  solid TRUE  # this is a solid shape
                  texCoord    TextureCoordinate {     # pyramid
            point [ 0 0,      # index zero
                     1 0,   # index one
                     1 1,     # index two
                     0 1,  # index three
                     0.5 0.5 # center, index four
            ]
                  }
                  texCoordIndex [ 1, 2, 3, 0, -1, # bottom
                                    0, 1, 4, -1, # front
2, 3, 4, -1,  # back
1, 2, 4, -1, # right
3, 0, 4, ] # left
            }
      }
```

As you can see, movies behave just like images.

I Seem to be Repeating Myself

Let's go back to our DOOM example again. To create the beginnings of our dungeon, we'd want to have a brick floor stretching out for a hundred meters in every direction. Here's how we'd begin:

```
      #VRML V2.0 utf8
      # This is the fifteenth example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "brick.jpg" ]
}
            }
            geometry Box { size 100 0.01 100 }  # big floor
      }
```

When we look at it (and remember to turn, so that it comes into view), we find, to our horror, that the brick pattern has grown to cover the entire surface of the Box.

This is ridiculous, these bricks must be tens of meters across! Why did this happen? The browser dutifully stretched the texture map to cover the entire surface of the Box – in other words, one side of the Box equals the entire area of the image. So no matter how big – or how small – we make the Box, the texture will cover it perfect.

To get around this, we need to work with the last of the three fields of the Appearance node, the textureTransform field. This field takes as its input the TextureTransform node, which has the following definition:

```
      TextureTransform {            # TextureTransform node
            center            # SFVec2f
            rotation          # SFFloat
            scale       # SFVec2f
            translation # SFVec2f
      }
```

Here we can see several of the fields we saw in the Transform node; this time, though, they're used for manipulating two-dimensional images, not three dimensional forms. In order to change the size of a texture we'd use the scale field, just as we used the scale field in the Transform node to alter the size of objects. However, the scale field in the TextureTransform node works in the reverse way from its counterpart**. In order to make a texture smaller, you must increase the scale value.** For example, a scale field value of 2 2 will make the texture twice as small in S and twice as small in T; in the same area, four times as many textures can be applied to a surface.

So let's go back to our DOOM example again, and make the texture five times smaller in both width and height; that'll mean that twenty-five textures can fit into the area of just one before:

```
      #VRML V2.0 utf8
      # This is the sixteenth example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "brick.jpg" ]
}
textureTransform TextureTransform {
      scale 5 5 # 1/5 width, 1/5 height
}
            }
            geometry Box { size 100 0.01 100 }  # big floor
      }
```

The texture map has repeated itself five times in both dimensions. Using the scale field, you could get any number of repetitions in any dimension; here we'll get more repetitions in width than in height:

```
      #VRML V2.0 utf8
      # This is the seventeenth example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "brick.jpg" ]
}
textureTransform TextureTransform {
      scale 5 2 # 1/5 width, 1/2 height
}
            }
```

```
            geometry Box { size 100 0.01 100 }  # big floor
      }
```

And you'll a pattern that squeezes the bricks more than a bit.

You can use the rotation field of the TextureTransform node to spin the texture map. As always the rotation value is given in radians, so you may need the Magic Radians & Degrees Calculator handy. Let's say that we want to rotate the bricks by 15 degrees. This is how the VRML would look:

```
      #VRML V2.0 utf8
      # This is the eighteenth example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "brick.jpg" ]
}
textureTransform TextureTransform {
      rotation 0.2633 # fifteen degrees
      scale 5 2 # 1/5 wide, ½ tall
}
            }
            geometry Box { size 100 0.01 100 }  # big floor
      }
```

We can see the spin on the bricks quite clearly.

Finally, you can use the translation field of the TextureTransform node to move the origin of the image's coordinate system to another point on the image. Why would you do this? Perhaps to focus on particular features, or perhaps to give a "scrolling" effect to your images. Here we take our world map image, and, by setting the S value in the translation field to 0.5, we can make the mid-line of the image its starting point:

```
      #VRML V2.0 utf8
      # This is the nineteenth example on textures
      Shape {           #  Create a visible shape
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
}
textureTransform TextureTransform {
      translation 0.5 0 # halfway across
}
            }
            geometry Box { }  # big floor
      }
```

This repositions the map. This cuts off just a little bit of Africa, not exactly clean work. But a value of 0.45 for S in the translation field works just great:

```
      #VRML V2.0 utf8
      # This is the twentieth example on textures
      Shape {           #  Create a visible shape
```

```
            appearance Appearance { # Create appearance
                  texture ImageTexture {  # Create texture
                        url [ "worldmap.jpg" ]
}
textureTransform TextureTransform {
      translation 0.45 0 # halfway across
}
            }
            geometry Box { }  # big floor
      }
```

Graphics Gods

While it might be going a bit far to say that you're a graphics god, at this point let me tell you something important – **you've mastered all of the key concepts in computer graphics**; shapes, materials, complex surfaces and texture maps are the foundation of any understanding of computer graphics. What we're about to cover – level of detail, anchors to the Web, inline worlds and the like – these are features that are specific to VRML.

So you should be proud of yourself – you're now a certified graphics god in the making!