Chapter 9: A Face Only a Mother Could Love

Getting to the Point

We've gone just about as far as we can with our understanding of VRML's built-in shapes. But the world-at-large has a whole host of shapes that don't neatly decompose into some combination of Sphere, Box, Cube and Cylinder nodes. While that might be fine for a simple automobile – or even a drastically reduced human figure – these solids haven't enough character to begin to model the complexity of the real world. The objects that surround us have complex surfaces, curving this way and that, with holes and serrated edges and textures that give them their unique, sensual qualities.

How do we get these forms into the virtual world?

If you recall from the 3D Graphics Primer in chapter 4, all objects in the virtual world are composed of *points*; these points are "wired" together into *frames*, and then these frames are given *surfaces*. The built-in shapes condense this step-by-step process into a single node name, hiding all of this complexity from you. That makes them easy to learn and very useful. It also makes them inflexible; you can't get a Box to be anything other than a six-sided figure with parallel and perpendicular faces, no matter how hard you try. That's just what a Box is – to change it, you'd have to be able to modify the points that make up the Box, and they're not available for modification.

To define our own, arbitrary shapes, we'd need a node that would allow us to define the points that make up that shape. The Coordinate node in VRML is just that node:

```
Coordinate {            # Coordinate node definition
      point []          # MFVec3f, mult. Values
}
```

The Coordinate node is just about the simplest node in VRML; its one field, point, contains a list of all of the points to be defined in the node. How does it work? Well, let's say that we wanted to create a single four-sided polygon – a square. It'd require that four points be defined. That shape might be defined in VRML by a Coordinate node that looked something like this:

```
      #VRML V2.0 utf8
      # This is the first example on complex shapes
Coordinate {            # Coordinate node example
            point [           # MFVec3f, mult. Values
                  0 0 0,      # First point
                  1 0 0,      # Second point
                  1 1 0,      # Third point
                  0 1 0 # Fourth and final point
            ]
```

```
}
```

Each of the four points defined in the point field is on its own line, each one – except for the last one – is followed by a comma, to separate it from the other points. We could have defined them all on the same line, so long as we had separated them with commas. So what happens if we pop it into a browser:

Nothing! Why is that? There are at least two reasons. First, we haven't put this inside of a Shape node, so it won't be visible. And next, although we've defined the points, we haven't connected them together, to define a visible surface. While the Coordinate node allows us to create points, we need to use another node, IndexedFaceSet, to turn those points into visible surfaces. IndexedFaceSet is one of the most common nodes in VRML, used for all irregular surfaces, and it's loaded with fields:

```
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
}
```

Although there are lots of fields here, only two are of any importance to the creation of faces – the coord and coordIndex fields. The coord field has a data type of SFNode, which is meant – in most situations – to take a Coordinate node as its value. So that's where we'd put the points that we'd defined. But where do we "wire" them together into a face? That happens in the coordIndex field – on the basis of *index values*.

What are index values? They're numbers – really, references - that are automatically attached to the points that you define in the point field of the Coordinate node. These reference numbers start with the first defined point, and continue through to the last point. However, because computer graphics is the domain of computer programmers, **these reference numbers begin at zero**, rather than at one; it seems that computers like to begin counting with zero just as much as humans like to begin counting with the number one. Here's that Coordinate node again, with its implied index values:

```
Coordinate {            # Coordinate node
      point [           # MFVec3f, mult. Values
            0 0 0,      # First point, index zero
            1 0 0,      # Second point, index one
            1 1 0,      # Third point, index two
            0 1 0 # Fourth point, index three
      ]
```

```
}
```

The thing to remember is that index values go from zero to one less than the number of defined points. In this case, with four points, the index values go from zero to three. These index values are used in the coordIndex field to create faces – literally "connecting the dots" to create polygon faces. Here's the completed example – inside of a Shape node – using the Coordinate and IndexedFaceSet nodes:

```
#VRML V2.0 utf8
# This is the second example on complex shapes
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 0 1 # purple
            }
      }
      geometry IndexedFaceSet {     # complex shape!
            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 ] # connect points!
      }
}
```

Instead of supplying a built-in shape to the geometry field of the Shape node, we define a complex form using IndexedFaceSet. Within the IndexedFaceSet node, we define our four points by supplying the Coordinate node as the value for the coord field, and then – most importantly – we "connect the dots" in the coordIndex field, by listing all four point index values as the four points that define a surface. We should see a purple square.

Now we're getting somewhere! This is the basic technique which is used to create arbitrary shapes in VRML.

Normal to a Fault

It's important to note that we can't "connect the dots" in any random order. The way the points are connected – in the coordIndex field – determines both the form of the surface and the direction of its *normal*. The normal – you'll remember from the 3D Graphics Primer – is an imaginary arrow which determines if a given surface reflects light or lets light pass through it. If a surface reflects light, it's visible; if it doesn't, then it's quite invisible. Surfaces in the virtual world have two sides, but they only have one visible side – that is, one side where the normal is pointing away from the surface.

When we "connect the dots", we establish the surface normal; in VRML the surface normal is pointing away from the surface whose points have been connected in a counter-clockwise fashion. If you take a look at how we defined the coordIndex in the previous

example, you'll see that it describes a counter-clockwise direction. If we take our previous example and reverse the direction of the points in the coordIndex field, it'd look like this:

```
#VRML V2.0 utf8
# This is the third example on complex shapes
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 0 1 # purple
            }
      }
      geometry IndexedFaceSet {     # complex shape!
            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, 3, 2, 1 ] # reverse points
      }
}
```

Now, when we pop it into the VRML browser, the face is pointing away from us.

So we can't see anything. But if we use one of the object rotation tools, such as "Study", or "Spin", we can rotate the surface so that it becomes visible to us.

It's not that it's not there; it's that it's not visible.

If we wanted to create a form that was visible from both sides, we'd need to define two distinct surfaces from the same set of points. This is easy to do, because coordIndex is an MFInt32 field; we can have as many numbers as we need, and can even reuse numbers. But we need to be able to tell the computer when we've finished defining one surface and have begun defining another. In this case, we use the index value of –1 to indicate that we've finished defining one surface, and are moving on to another. So our example – defining a face visible from both sides – would look like this:

```
#VRML V2.0 utf8
# This is the fourth example on complex shapes
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 0 1 # purple
            }
      }
      geometry IndexedFaceSet {     # complex shape!
            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, -1, # front face
0, 3, 2, 1    # back face
                  ]
            }
      }
```

In the coordIndex field, we define the front face, end the face with a –1, then move on to define the back face. This should give us an object with two visible sides – but you'll need to spin it around to see both faces.

In most situations – particularly dealing with solid objects – it isn't necessary to give surfaces two sides. However, some objects, such as signs and billboards – need to be visible both front and back. In situations like that, you'd use this technique.

Pyramid Power

Now let's define a rather more complex shape, and see how to create entirely enclosed shapes using the basic VRML nodes. We're going to create a pyramid, both because it's a cool shape – at least, the Egyptians thought so – and because it requires that we define only five points, an economy that will make this example pretty easy to understand.

That'd mean we'd have the following Coordinate node:

```
      Coordinate {            # Coordinate node
            point [           # MFVec3f, mult. Values
                  0 0 0,      # First point, index zero
                  1 0 0,      # Second point, index one
                  1 0 -1,     # Third point, index two
                  0 0 -1,     # Fourth point, index three
                  0.5 0.5 -0.5 # Fifth point, index four
            ]
      }
```

This is just an modification of the design for the square from our previous examples. In this case, we'll put the square "underneath" the pyramid along the x and z axes. We'll build this example up in stages, beginning with this square:

```
      #VRML V2.0 utf8
      # This is the fifth example on complex shapes
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 0.5 0.5 0.5 # gray
                  }
            }
            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 ] # bottom down
      }
}
```

This should give us the gray bottom of the pyramid, pointing down – because we want all the visible faces of the pyramid to be on the outside, that is, facing outward.

Now let's define the front of the pyramid. We'll be using point indexes zero, one and four, and we must connect them in that way – counter-clockwise – so that they're visible to us:

```
#VRML V2.0 utf8
# This is the sixth example on complex shapes
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 0.5 0.5 # gray
            }
      }
      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 ] # front
      }
}
```

We should now have something that looks like a pyramid from the front.

It's a beginning. Now let's add the rear side, using point indexes two, three and four, in that order, to keep it visible from the right side:

```
#VRML V2.0 utf8
# This is the seventh example on complex shapes
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 0.5 0.5 # gray
            }
      }
      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 ] # back
            }
      }
```

This won't look much different till we look at it from above.

Finally, let's add the right and left sides of the pyramid. The right side will use point indexes one, two and four, while the left side will use point indexes three, zero and four:

```
      #VRML V2.0 utf8
      # This is the eighth example on complex shapes
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 0.5 0.5 0.5 # gray
                  }
            }
            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
            }
      }
```

Note that we've included the solid field and set it to TRUE, which may give the browser a clue as to how to draw it more quickly. Whenever you have created closed shapes – solids – you should set the solid field to TRUE.

This now gives us a completed pyramid. It actually looks a great deal like the real thing, doesn't it?

Candy-Colored Pyramids

One of the constraints in the Shape node is that only one surface can be applied to an object – no matter how complex that surface is. The IndexedFaceSet node provides a way to work around this, with the color and colorIndex fields. These allow you to specify a different color for every surface you define in the coordIndex field of the node.

To begin with, the color field has a data type of SFNode, and in most cases wants to have a Color node provided to it. Here's the very simple definition of the Color node:

```
Color {           # Color node definition
      color []    # MFColor, mult. Values
}
```

Essentially, the Color node provides a way for you to define a list of colors – as many or as few as you might like. These colors – just like the points in the Coordinate node – have reference index values, once again starting from zero.

Let's say that we want to color every side of they pyramid a different color – that would be five color definitions. Here's how that might look in a Color node:

```
Color {           # Color node
      color [           # MFColor, mult. Values
            1 1 1,      # First color, index zero
            1 0 0,      # Second color, index one
            1 1 0,      # Third color, index two
            0 1 0,      # Fourth color, index three
            0 0 1 # Fifth color, index four

      ]
}
```

The colorIndex field explicitly maps a color to a surface, it can do this because the surfaces created in the coordIndex field also have reference index values. The first surface created is surface zero, and so on. That means that the bottom of the pyramid – the first surface created – is surface zero, while the front is surface one, the back surface two, the right is surface three, and the left surface four. So let's add these details to our existing example, and add color to the sides of the pyramid:

```
#VRML V2.0 utf8
# This is the ninth example on complex shapes
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0.5 0.5 0.5 # gray
            }
      }
      geometry IndexedFaceSet {     # complex shape!
      color Color {           # Color node
                  color [           # MFColor
                        1 1 1,      # index zero
                        1 0 0,      # index one
```

```
                              1 1 0,      # index two
                              0 1 0,      # index three
                              0 0 1 # index four
                        ]
                  }
                  colorIndex [ 0, 1, 2, 3, 4 ] # all surfaces
                  colorPerVertex FALSE # Must be set FALSE!
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
            }
      }
```

The one other field that must be present is the colorPerVertex field. There's two types of coloring that you can do with the color and colorIndex fields; face coloring – which we're doing here – and *vertex* coloring, which is much more complicated (and correspondingly subtle) but beyond the scope of this book. So, we need to set that field to FALSE, to ensure that we get face coloring. Now we should see a brighly-colored pyramid.

It's not necessary to have as many colors defined as surfaces; you can reuse colors over and over again on a form to create a complex patterns of colors. In this example, we'll reuse green and blue colors on all five sides of the pyramid:

```
      #VRML V2.0 utf8
      # This is the tenth example on complex shapes
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 0.5 0.5 0.5 # gray
                  }
            }
            geometry IndexedFaceSet {     # complex shape!
            color Color {           # Color node
                        color [           # MFColor
                              0 1 0,      # index zero
                              0 0 1 # index one
                        ]
                  }
                  colorIndex [ 1, 0, 0, 1, 1 ] # all surfaces
                  colorPerVertex FALSE # Must be set FALSE!
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
            }
      }
```

As you can see, the colors are repeated across the object.

So, even if you have a limited palette of colors, you can have absolute control over the colorings of complex objects.

Way Too Complex

Although we've been talking about how the Coordinate and IndexedFaceSet nodes are useful for creating very complex objects, we've created only very simple objects in this chapter. Why is that? One word: complexity. As we start adding points and faces it gets very hard to keep track of everything. This is where computers come in; they're ever so much better at keeping track of the kinds of bookkeeping that we do when we create a complex model – so, while we might not create a complex model by hand using IndexedFaceSet, we might program a computer to do so, or use a modeling tool that does.

However, it'd be a mistake if you didn't have a peek at what a complex VRML model looks like. This one – an 80-sided sphere – is still very simple, but it goes a long way to demonstrating the kind of complexity you can expect from other models:

```
#VRML V2.0 utf8
# This is the eleventh example on complex forms
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0.2 1 0.2
            }
      }
      geometry IndexedFaceSet { # faces: 80
            coord Coordinate {
                  point [
                        4.19952 1.92726 -2.73438,
                        5.39063 2.26563 -2.73438,
                        4.69051 1.83293 -3.86719,
                        6.58173 1.92726 -2.73438,
                        6.09074 1.83293 -3.86719,
                        5.39063 1.19111 -4.66163,
                        3.5577 1.13281 -3.43449,
```

```
                4.25781 0.700117 -4.5673,
                3.46337 0 -3.92548,
                4.25781 -0.700117 -4.5673,
                5.39063 0 -5,
                5.39063 -1.19111 -4.66163,
                6.52344 0.700117 -4.5673,
                7.31788 0 -3.92548,
                6.52344 -0.700117 -4.5673,
                7.22355 1.13281 -3.43449,
                7.65625 0 -2.73438,
                7.22355 1.13281 -2.03426,
                7.31788 0 -1.54327,
                7.22355 -1.13281 -3.43449,
                7.22355 -1.13281 -2.03426,
                6.58173 -1.92726 -2.73438,
                6.09074 -1.83293 -3.86719,
                4.69051 -1.83293 -3.86719,
                5.39063 -2.26563 -2.73438,
                4.19952 -1.92726 -2.73438,
                4.69051 -1.83293 -1.60156,
                6.09074 -1.83293 -1.60156,
                5.39063 -1.19111 -0.807119,
                6.52344 -0.700117 -0.901446,
                5.39063 0 -0.46875,
                6.52344 0.700117 -0.901446,
                5.39063 1.19111 -0.807119,
                4.25781 -0.700117 -0.901446,
                4.25781 0.700117 -0.901446,
                3.46337 0 -1.54327,
                3.5577 1.13281 -2.03426,
                4.69051 1.83293 -1.60156,
                6.09074 1.83293 -1.60156,
                3.125 0 -2.73438,
                3.5577 -1.13281 -3.43449,
                3.5577 -1.13281 -2.03426
        ]
}
coordIndex [
        0, 1, 2, -1,
        3, 4, 1, -1,
        5, 2, 4, -1,
        1, 4, 2, -1,
        0, 2, 6, -1,
        5, 7, 2, -1,
        8, 6, 7, -1,
        2, 7, 6, -1,
        8, 7, 9, -1,
        5, 10, 7, -1,
        11, 9, 10, -1,
        7, 10, 9, -1,
        5, 12, 10, -1,
        13, 14, 12, -1,
        11, 10, 14, -1,
        12, 14, 10, -1,
        5, 4, 12, -1,
        3, 15, 4, -1,
        13, 12, 15, -1,
```

```
4, 15, 12, -1,
13, 15, 16, -1,
3, 17, 15, -1,
18, 16, 17, -1,
15, 17, 16, -1,
13, 16, 19, -1,
18, 20, 16, -1,
21, 19, 20, -1,
16, 20, 19, -1,
11, 14, 22, -1,
13, 19, 14, -1,
21, 22, 19, -1,
14, 19, 22, -1,
11, 22, 23, -1,
21, 24, 22, -1,
25, 23, 24, -1,
22, 24, 23, -1,
25, 24, 26, -1,
21, 27, 24, -1,
28, 26, 27, -1,
24, 27, 26, -1,
28, 27, 29, -1,
21, 20, 27, -1,
18, 29, 20, -1,
27, 20, 29, -1,
28, 29, 30, -1,
18, 31, 29, -1,
32, 30, 31, -1,
29, 31, 30, -1,
28, 30, 33, -1,
32, 34, 30, -1,
35, 33, 34, -1,
30, 34, 33, -1,
35, 34, 36, -1,
32, 37, 34, -1,
0, 36, 37, -1,
34, 37, 36, -1,
0, 37, 1, -1,
32, 38, 37, -1,
3, 1, 38, -1,
37, 38, 1, -1,
32, 31, 38, -1,
18, 17, 31, -1,
3, 38, 17, -1,
31, 17, 38, -1,
8, 39, 6, -1,
35, 36, 39, -1,
0, 6, 36, -1,
39, 36, 6, -1,
8, 40, 39, -1,
25, 41, 40, -1,
35, 39, 41, -1,
40, 41, 39, -1,
8, 9, 40, -1,
11, 23, 9, -1,
25, 40, 23, -1,
9, 23, 40, -1,
```

```
                  28, 33, 26, -1,
                  35, 41, 33, -1,
                  25, 26, 41, -1,
                  33, 41, 26, -1
            ]
      }
}
```

All of this just to make a rather primitive spherical shape.

As you get braver – and begin to take a peek "under the hood" of VRML worlds, you'll find that most worlds are mostly made up of IndexedFaceSet nodes, with their associated Coordinate nodes and coordIndex field values. It's the basis for all forms that aren't built-in.

The Silver Lining

Of course, you don't need to create solid forms using points; it's just as easy to create outlines, real "frames" using VRML. A companion node to IndexedFaceSet, IndexedLineSet, can be used to create "wire" frames rather than surfaces. The node's definition looks like a simplified version of the IndexedFaceSet:

```
      IndexedLineSet {  # IndexedLineSet definition
            color             # SFNode
            coord             # SFNode
            colorIndex []           # MFInt32
            colorPerVertex          # SFBool
            coordIndex []           # MFInt32
      }
```

These are the five fields we've already covered in this chapter – and they work just the same way here; however, this node does not create surfaces, rather, it literally "connects the dots" between points, to create frames. Here's the first pyramid example, as an IndexedLineSet:

```
      #VRML V2.0 utf8
      # This is the twelfth example on complex shapes
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 0.5 0.5 0.5 # gray
                  }
            }
            geometry IndexedLineSet {     # 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
            }
      }
```

This creates the same form, but it's only the outline of the pyramid.

Once again, we can use the color and colorIndex fields to change the color of the wiring on each of the sides of this pyramid. Let's adapt the tenth example for IndexedLineSet:

```
      #VRML V2.0 utf8
      # This is the thirteenth example on complex shapes
      Shape {
            appearance Appearance {
                  material Material {
                        diffuseColor 0.5 0.5 0.5 # gray
                  }
            }
            geometry IndexedLineSet {     # complex shape!
            color Color {           # Color node
                        color [           # MFColor
                              0 1 0,      # index zero
                              0 0 1 # index one
                        ]
                  }
                  colorIndex [ 1, 0, 0, 1, 1 ] # all surfaces
                  colorPerVertex FALSE # Must be set FALSE!
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
            }
      }
```

Some of the wires are green, an others are blue.

Wrapping It Up

That's about all we need to cover on complex shapes; the techniques you've learned here will show up again and again as we progress into deeper understanding of the features of

VRML, for most objects are composed of these simple primitives. But now it's time to learn how to wallpaper cyberspace…