Chapter 8: Put That There

Transporter Failures?

One of the most horrible accidents that can ever happen in Star Trek's mythical future of incredible technology, a transporter failure defines *nasty*. One moment you're fine, the next you're interpenetrating someone else's flesh and bones – and, well, it isn't very pretty.

Yet, that's exactly where we are right now in our own study of VRML. Objects in the VRML world are drawn from a center point known as the *origin*, the point in space where the x, y, and z coordinate axes all have their zero value. Every object we draw – a Box, Sphere, Cone, Cylinder, or anything else – gets drawn at the center of the virtual universe. So what do we get when we create multiple objects? A mess:

```
#VRML V2.0 utf8
# This is the first example on Transforms
# Everything drawn from the origin, all jumbled up!
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 1 0 # green
            }
      }
      geometry Sphere { radius 1.25 }     # Create form
}
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 1 0 0 # red
            }
      }
      geometry Box { size 2 2 1 }   # Create form
}
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 0 0 1 # blue
            }
      }
      geometry Cone { bottomRadius 1.25 } # Create form
}
Shape {           #  Create a visible shape
      appearance Appearance { # Create appearance
            material Material {     # Create material
                  diffuseColor 1 1 1 # white
            }
      }
      geometry Cylinder { }   # use defaults
}
```

This world creates all four built-in VRML solids, in different colors, with slightly different sizes – designed to accentuate the problem. All together, it looks more than a little confusing.

The origin point of the virtual world is in the midst of all of these objects, so each of them crowds into every other. How can we avoid this mess? In addition to a universal (or *global*) coordinate system, you can have *local* coordinate systems – as many as you like. Each local coordinate system has its own origin, and all drawing within a local coordinate system takes place from its origin, not the global origin. To create local coordinate systems, we need to become familiar with the Transform node.

Transforming Yourself

The Transform node encompasses a whole range of functionality needed to position and orient objects in the virtual world. It's the only node that allows you to create a local coordinate system; once the local coordinate system has been created, you can continue creating visible objects without getting those unsightly interpenetrating shapes. Here's the definition of the Transform node and its fields:

```
Transform { # definition of Transform node
      center                  # SFVec3f
      children []       # MFNode, takes mult. Nodes
      rotation                # SFRotation – defined below
      scale             # SFVec3f
      scaleOrientation  # SFRotation – defined below
      translation       # SFVec3f
      bboxCenter        # SFVec3f
      bboxSize                # SFVec3f
}
```

Here, for the first time, we see an MFNode data type, in the children field. The children field is at the heart of the Transform node; it contains a list of all nodes – that is, all objects – which are to be affected by the Transform node. All of the other field data types we've seen before, except for SFRotation, which we'll cover separately – and at some length – a little further along in this chapter.

Lost In Translation

The most straightforward of all of the fields in the Transform node is the translation field. The x, y, z value supplied as the SFVec3f value to this field becomes the origin point of the local coordinate system. For example, let's say that we wanted to define a green Sphere which is slightly to the right of the global origin point of the world. Here's one way we might do it:

```
#VRML V2.0 utf8
# This is the second example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
```

```
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 3 0 0 # move a bit in x
}
```

In this example, you can begin to see how the children field is used. The Shape node – which defines the green Sphere – is entirely inside the children field. The translation field will move the local origin point three units (meters) to the right of the global origin point in x, none in y or z.

Doesn't look different, does it? Why is that? Well, VRML browsers are smart – in some ways, a bit too smart. The browser doesn't necessarily point itself at the origin of the world. Instead, it points itself at whatever objects might be visible in the world. That way, you're much less likely to be looking at empty space when you enter a virtual world. However, that makes it impossible to tell that we've offset the local origin of the Sphere by three units.

On the other hand, if we add an object – say a red Box – and define it at the global origin, we should be able to see some difference:

```
#VRML V2.0 utf8
# This is the third example on Transforms
# This is the red Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 0 0
            }
      }
      geometry Box { }
}
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 3 0 0 # move slightly to the right
}
```

Now we can see two entirely distinct objects.

That's it – translation is the essential concept behind placement of objects in the virtual world. Moving on to a slightly more complicated example, let's work all of the possible directions – that is, six (plus and minus in three dimensions) – and create a range of Sphere nodes surrounding a central Box. For clarity's sake, we'll give each Sphere a different color:

```
#VRML V2.0 utf8
# This is the fourth example on Transforms
# This is the cyan Box
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 0 1 1
            }
      }
      geometry Box { }
}
# This Sphere is to the right and green
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 3 0 0 # move slightly to the right
}
# This Sphere is to the left and blue
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 0 1 # blue
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation -3 0 0 # move slightly to the left
}
# This Sphere is to the front and red
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 0 0 # red
                        }
```

```
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 0 3 # move slightly to the front
}
# This Sphere is to the back and yellow
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 1 0 # yellow
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 0 -3 # move slightly to the back
}
# This Sphere is to the top and white
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 1 1 # white
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 3 0 # move slightly to the top
}
# This Sphere is to the bottom and gray
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0.5 0.5 0.5
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 0 –3 0 # move slightly to the bottom
}
```

We should see a central cube surrounded by six differently colored spheres.

In each of these translation fields, we've only moved along one axis at a time – the other two values in the field have been zero. This doesn't need to be – it's possible to move

along multiple axes simultaneously. Let's say that I'd like to take the current example and add another sphere, this one in front of, to the right of, and above the cube. We'd add the following Transform node to the existing file:

```
# This is the fifth example on Transforms
# It's incomplete on its own
# We add this to the existing file
# A purple sphere that's above, right and in front
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 0 1 # purple
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      translation 3 3 3 # move front, right, and up
}
```

And we'll now have a purple Sphere in front, to the right and above.

The value supplied to the translation field can be *any* valid value – there is no range specified or implied. And because they're floating point numbers, those values can be very big (or very small) indeed.

Nesting Syndrome

One of the most common places you'll find a Transform node is within the children field of another Transform node. Transform nodes are – in graphics lingo – *cumulative*; this means that each Transform adds its own particular values to any Transform nodes that might be inside of it.

Why would you want to put Transform nodes within one another? Generally, it's to denote something known as a *parent-child relationship*. The easiest way to think about this might be to look at your arm, hand and fingers. Your arm is the *parent* in this relationship, because if you move your arm, your hand's going to follow. Whereas, when your hand – the *child* in this relationship - moves, your arm doesn't. The same parent-child relationship holds between your hand and your fingers; wherever your hand goes, your fingers follow, but crossing your fingers doesn't move your hand.

Say that we wanted to recreate a version of the VRML logo – which is a red cube, green sphere and blue cone – using parent-child relationships. There's all spread out in a line, splaying out on the x axis, so it might look like this:

```
#VRML V2.0 utf8
# This is the sixth example on Transforms
# This is the red Box (cube, actually)
```

```
Shape {
      appearance Appearance {
            material Material {
                  diffuseColor 1 0 0 # red
            }
      }
      geometry Box { }
}
# An example of nested transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
            # Transform inherits the parent translation
            Transform {
                  children [  # open list of nodes
                        # define blue cone
                        Shape {
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 0 0 1
                                    }
                              }
                              geometry Cone {}
                        }
                  ] # End list of children
                  translation 3 0 0 # move a bit right
            }
      ] # closes the list of children nodes
      translation 3 0 0 # move slightly to the right
}
```

Now we should see all three figures, cube, sphere, cone, from left to right.

Even though each Transform node only moved each local coordinate system three units in x from the previous coordinate system, the inner Transform node for the blue cone inherited the three unit movement from its parent, the green sphere, and, adding onto its own three unit movement, had a total of six units of movement from the global origin where the red cube sits.

It’s possible to have an indefinite number of nested Transform nodes, each of them inheriting the cumulative qualities of all the Transform nodes that are its ancestors. This means that you can build very complex shapes out of very simple forms. For example, a human hand could be constructed from a simple set of Box nodes that have parent-child relationships between the forearm (one Box), the hand (another Box) and the fingers (five more boxes).

Sense and Scalability

Did you ever wish that you could shrink yourself down to sub-miniature proportions? Or perhaps grow to giant's size? Such things are possible in the virtual world – using the Transform node. One of the other fields of that node, the scale field, allows you to take any set of objects within the children field of the Transform node, and make them larger or smaller, along any axis.

This effect is most noticeable with Sphere nodes – and it's very hard to get any other way. Here's an example of a Sphere node that's uniformly scaled to twice its normal size:

```
#VRML V2.0 utf8
# This is the seventh example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      scale 2 2 2 # double the size of the object
}
```

Once again, when this example is viewed, the browser is being too smart for its own good. We've made the object twice as big in every dimension – by supplying the SFVec3f value of 2 2 2 to the scale field – but the browser has simply backed off a bit, leaving the object looking exactly the same size.

We need to try some non-uniform scaling – that is, scaling the object differing degrees in different dimensions. Let's make the object half as tall, twice as wide, and no more deep:

```
#VRML V2.0 utf8
# This is the eighth example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
      ] # closes the list of children nodes
      scale 2 0.5 1 # double width, half height
```

```
}
```

This time, the scaling is much more visible. Sort of looks like a dirigible, doesn't it? Long and somewhat deep, but flattened on top – the scale field of the Transform node is a very easy way to make a rather complicated shape.

As was true with the translation field, the scale field is also cumulative in nested Transform nodes. In this extension of the previous example, we'll see two spheres, green and blue; the blue one is a child of the green, and inherits its scale, and adds some of its own:

```
#VRML V2.0 utf8
# This is the ninth example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0 # green
                        }
                  }
                  geometry Sphere { }     # Create form
            }
            # child Transform node for blue sphere
            Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 0 0 1
                                    }
                              }
                              geometry Sphere { }
                        }
                  ]
                  scale 0.5 1 0.5 # half as wide and deep
                  translation 3 0 0 # move it away a bit
            }
      ] # closes the list of children nodes
      scale 2 0.5 1 # double width, half height
}
```

We used the translation and scale fields together within the inner Transform node, to move the inner Shape away from the outer one, and, as you can see, the inner Shape did inherit the qualities of outer Transform, except that it's half as wide and half as deep.

It's even possible to scale objects back to their normal dimensions. In this final example on the scale field, we nest another Transform node inside the inner Transform node, and create a red Sphere which will be scaled to half width, returning it to normal dimensions:

```
#VRML V2.0 utf8
# This is the tenth example on Transforms
Transform { # Open the Transform node
```

```
            children [  # open list of nodes
                  # This node is within the children field
      Shape {           #  Create a visible shape
                        appearance Appearance {
                              material Material {
                                    diffuseColor 0 1 0 # green
                              }
                        }
                        geometry Sphere { }     # Create form
                  }
                  # child Transform node for blue sphere
                  Transform {
                        children [
                              Shape {
                                    appearance Appearance {
                                          material Material {
                                                diffuseColor 0 0 1
                                          }
                                    }
                                    geometry Sphere { }
                              }
                              Transform {
                                    children [
                                          Shape {
                                                appearance Appearance {
                                                      material Material
{

      diffuseColor 1 0 0
                                                }
                                          }
                                          geometry Sphere { }
                                    }
                              ]
                              scale 0.5 1 1
                              translation 3 0 0
                        }
                  ]
                  scale 0.5 1 0.5 # half as wide and deep
                  translation 3 0 0 # move it away a bit
            }
      ] # closes the list of children nodes
      scale 2 0.5 1 # double width, half height
}
```

Which gives us three shapes, the last one a perfect red sphere. It's important to notice that the red and blue spheres are twice as close together as the blue and green spheres. Why is this? After all, the Transform node for both indicates that each should be 3 units away from one another. But that's before the scale field values have been figured in. In the Transform node for the red Sphere, the x scale has been cut in half – so any movements along x are also cut in half. Keep this in mind when using the fields of the Transform node together, or when they're nested – because the value in one might have an affect on another.

You'll sometimes find you'll need the Transform node when importing objects that have been created by other people; your idea of a mailbox for the front of your virtual yard may be ten thousand times smaller than how someone else envisions it. When that happens, you can put the object inside a Transform node and use the scale field to bring it down (or up) to size.

Sit on it and Rotate

Along with translation and scale, the rotation field is the last of the key fields in the Transform node. The data type for the rotation field is SFRotation, which he haven't seen before. The SFRotation data type can be thought of as a combination of two other data types - an SFVec3f value triplet, and an SFFloat value. Rotation of an object actually has four basic components.

The *axes of rotation* are related to the *axes of translation*; that is, pitch, yaw and roll are rotations around the x, y, and z axes, respectively. It's not enough to specify which axis – or axes – to rotate around, you also need to specify how far around the axis to rotate. In our day to day life, we specify degrees of rotation in degrees – we talk about "doing a one-eighty," when we talk about turning ourselves around, or "doing a three-sixty," when we're going in circles. This refers to the fact that there are three hundred and sixty degrees in the full rotation of a circle.

Programmers, scientists and other technical types rarely use degrees – as it turns out, the math is very clumsy. Instead they rely on another mathematical convenience, the *radian*. The radian is derived from the relationship between the radius of a circle and its circumference, that is, the circumference of a circle is 2 times its radius. The radian then, is a unit of rotation, and there are 2 radians in a full circle. A rough approximation for this is that 6.28 radians (2 x 3.14) equals 360 degrees. Rotations in VRML are *always* given in radians, never in degrees.

Does this mean you're going to have to do conversions all the time while you work through this book? Not if I can help it. After all, this is an age of computers, and computers are very good at the kinds of nuisance calculations - like conversions between degrees and radians – that we'd rather avoid. For that reason, I've put the *Magic Radians & Degrees Calculator* on the CD-ROM. If you open the file radian.html in a JavaScript-capable browser (either Microsoft *Internet Explorer* or Netscape *Navigator* will be fine), you'll see this calculator.

The calculator has a bit of explanation about radians – always good for review – and a JavaScript applet which allows you to convert between radians and degrees, or vice versa. For example, if you type the number 360 into the box, then hit the button labeled "Deg2Rad", you'll get the number 6.28 out. Put 6.28 into the box, and hit the button labeled "Rad2Deg" and you'll get 360 – almost. As I said, this is all approximate, but it's more than close enough for any work we'll be doing here. You may want to bookmark the calculator as we work through this book – then you'll have easy access.

The SFRotation data type specifies the axes of rotation and the total angle of that rotation, in radians. Let's try it with a very simple example. While the Box shape is very useful, it always comes up face-on to you, which makes it impossible to distinguish if it is a real Box or just a flat surface. However, if we could yaw it – say, about 30 degrees – we'd be able to see another side, and know it's a Box. Here's how that might look:

```
#VRML V2.0 utf8
# This is the eleventh example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 0 0 # red
                        }
                  }
                  geometry Box { }  # Create form
            }
      ] # closes the list of children nodes
      rotation 0 1 0 0.5233 # yaw 30 degrees
}
```

Now let's pop that into a browser. We can clearly see two sides now, and know that it's a Box, not just a flat surface.

The rotation field specifies that the Box is only to be rotated around the y axis – that is, yawed – and that it's to be rotated by 0.523 radians, or 30 degrees (yes, I did use the calculator to get this figure). The other axes *must* be set to zero if they're not to be used.

Another built-in shape that it's easy to mistake is the Cone. It's hard to tell if one is looking at it head on or from slightly above. Let's pitch the Cone upward just a bit, perhaps 15 degrees – this is negative pitch, by the way – so that we can see underneath it:

```
#VRML V2.0 utf8
# This is the twelfth example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 0 1 # blue
                        }
                  }
                  geometry Cone { } # Create form
            }
      ] # closes the list of children nodes
      rotation 1 0 0 –0.26167 # pitch -15 degrees
}
```

We should be able to see underneath the Cone now.

As we see here, radian values can be either positive or negative, but they must have a absolute value less than 2 . A negative radian value indicates rotation in a reverse (counter-clockwise) direction along the axis.

Next, let's roll a Cylinder node 90 degrees:

```
#VRML V2.0 utf8
# This is the thirteenth example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 1 1 # white
                        }
                  }
                  geometry Cylinder { }   # Create form
            }
      ] # closes the list of children nodes
      rotation 0 0 1 1.57  # roll 90 degrees
}
```

We'll see a Cylinder on its side.

The Combination Dip, Tuck and Roll

It's possible to use the various rotations together, to create a rotation that has components of pitch, yaw and roll in it. However, this isn't trivial, and involves a fair bit of algebra. You see, a rotation on any one axis affects rotations on any of the other axes. For this reason, you need to *normalize* the value in the SFVec3f portion of the SFRotation data type. The process of normalization adjusts for the effect of combined rotations. The formulas to normalize the values in SFVec3f are very complicated. But – again, to save you from unnecessary work – we have the *Magic Vector Normalizer*. Load up normalizer.html on your JavaScript capable browser, and here's what you'll see:

All you need to do to use the normalizer is to tell it which axes you'll be rotating around, and it'll do the rest of the work. For example, let's say we want to take that Box, and both yaw it and pitch it 30 degrees. That would look like this:

```
#VRML V2.0 utf8
# This is the fourteenth example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 0 0 # red
                        }
                  }
                  geometry Box { }  # Create form
```

```
            }
      ] # closes the list of children nodes
      rotation 0.7071 0.7071 0 0.5233 # yaw and pitch
}
```

How did we get the numbers in the rotation field? From the calculator. Since we wanted equal rotation about the x and y axes, we type a 1 into both fields of the calculator, leaving the z field at zero. Upon pressing the button labeled "Do it!", we see that both the X Norm and Y Norm fields have a values of 0.7071. That's what we put into the rotation field. Once again we used the radian calculator to find that 30 degrees is 0.5233 radians. When we look at the Box, we'll see that it has acquired rotation about both axes.

With the dual rotation, three sides are visible. Let's try something quite a bit more complicated now. Let's pitch and yaw the Box by 30 degrees, but let's roll it by 60 degrees. To do that we'd put 0.5 into the X and Y values of the calculator, and 1.0 into the Z value of the calculator – because we're only pitching and yawing by half as much of a rotation as we will be rolling, 30 degrees versus 60. We get normalized results of 0.4082, 0.4082 and 0.8165, respectively. We use 1.047 for the radians value – that's 60 degrees. This is how it looks:

```
#VRML V2.0 utf8
# This is the fifteenth example on Transforms
Transform { # Open the Transform node
      children [  # open list of nodes
            # This node is within the children field
Shape {           #  Create a visible shape
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 0 0 # red
                        }
                  }
                  geometry Box { }  # Create form
            }
      ] # closes the list of children nodes
      rotation 0.4082 0.4082 0.8165 0.5233 # everything
}
```

We should see a Box that looks like it's tumbling through space.

All Together Now

For one final example, let's put together everything we've learned in this lesson about the Transform node and its translation, scale and rotation fields. Let's do a bang-up job of recreating the VRML logo in all of its glory, with a red Box that's pitched and yawed, a green Sphere that's scaled slightly larger, and a translucent blue Cone that pitched up so that you can see its bottom. First we begin with the Box:

```
#VRML V2.0 utf8
# This is the sixteenth example on transforms
# This is a red Box, pitched and rolled
# So it sits in its own Transform node
```

```
Transform {
      children [
            Shape {     # create red Box
                  appearance Appearance {
                        material Material {
                              diffuseColor 1 0 0 # red
                        }
                  }
                  geometry Box { }
            }
      ]
      rotation 0.7071 0.7071 0 0.5233 # pitch, yaw
}
```

One that's defined, we'll open up a parent Transform node for all of the subsequent shapes, so that they'll be drawing away from the global origin. Then we create the green Sphere and blue Cone, within this Transform, in Transforms of their own:

```
Transform {       # Move away from origin
      children [
            Transform { # inherits parent move
                  children [
                        Shape {     # create green Sphere
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 0 1 0
                                    }
                              }
                              geometry Sphere { }
                        }
                  ]
                  scale 1.25 1.25 1.25 # slightly big
            }
            Transform { # inherits parent move
                  children [
                        Transform {
                              children [
                                    Shape {     # blue cone
                                          appearance Appearance {
                                                material Material
{

      diffuseColor 0 0 1

      transparency 0.5
                                          }
                                    }
                                    geometry Cone { }
                              }
                        ]
                        rotation 1 0 0 –0.2633
                  }
            ]
            translation 3 0 0 # further away
      }
]
```

```
        translation 3 0 0 # slightly away
    }
```

We'll see a world where each item is subtly different.

Bored with Boxes?

We're just about at our limit of what we can do with simple shapes – we've explored an entire range of surfaces, positions and orientations for these basic objects. What do we do if we want to create a complex shape, something that we can't get with our basic built-in forms? That's what the next chapter is about, where things get a little less formula and a little bit more complex…