Chapter 17: Did the World Move for You, Too?

Timely Ideas

While we've been talking about VRML as a language for three-dimensional computer graphics, in practice it's actually a four-dimensional language. The concept of time (since Einstein, the fourth dimension) becomes essential as we discuss things like animation; animation is a form of interactivity that takes place through time. Realistic virtual worlds change through time, show wear and tear, or growth, or motion.

In the real world it is surprisingly difficult to measure time – it has to be gauged against some physical process, such as the movement of the sun, the flow of water, or the swing of a pendulum. The idea that time must *flow* to be measured has been incorporated into a VRML node known as TimeSensor. The TimeSensor node is the clock, stopwatch and timer for VRML, and – with both fields and events – here's how it looks:

```
TimeSensor {            # Definition of TimeSensor
      cycleInterval           # SFTime, exposedField
      enabled                 # SFBool, exposedField
      loop                    # SFBool, exposedField
      startTime               # SFTime, exposedField
      stopTime                # SFTime, exposedField
      isActive                # SFBool, eventOut
      cycleTime               # SFTime, eventOut
      fraction_changed  # SFFloat, eventOut
      time                    # SFTime, eventOut
}
```

We see a new field data type being used here, SFTime. It's actually a floating-point number (like SFFloat), but expressed with greater precision, representing the number of seconds that have elapsed since 1 January 1970, at midnight, Greenwich Mean Time – something that's also known as "*the Epoch*". Now we're well past that time, and given that there are 31,536,000 seconds in a year, we're probably up into numbers in the range of 851,472,000 seconds past the Epoch – almost a billion seconds. (Bet you didn't think you'd live to see a billion of anything, but you have.)

It's not so much important to know the exact value in an SFTime field as it is to know whether a particular time value is greater or less than another. The two TimeSensor fields startTime and stopTime rely on this comparison. A TimeSensor will start when the *world time* – that is, the actual time – is equal to or greater than the value in the startTime field; a TimeSensor will stop when the world time is equal to or greater than the value in the stopTime field. Both fields have a default value of 0, so a TimeSensor will not start on its own with its default values. However, there's a bit of a trick you can use to get a TimeSensor to start as soon as the world is loaded; set the value of startTime a value greater than or equal to stopTime, and set loop to TRUE.

The enabled field must be TRUE before the TimeSensor can become active; a value of FALSE turns the TimeSensor off if it's running, or prevents it from becoming active. The cycleInterval field sets the duration of the timer. This is a value in seconds, not the number of seconds past the Epoch, so for a 10 second countdown, the field value is a sensible 10. It has a default value of 1, that is, a second. If you want the TimeSensor to restart after the cycleInterval has been completed, set the loop field to TRUE; otherwise, the TimeSensor will run just once. The loop field marks the difference between a timer and a clock; both of them count time, but the timer counts down and is done, while the clock counts on and on and on.

When the TimeSensor is active, it sends emits a whole cluster of eventOut messages. The isActive eventOut emits TRUE when the TimeSensor begins running and FALSE when it stops. The cycleTime eventOut emits the current world time at the start of the TimeSensor cycle, and each time it loops, if it loops. The time eventOut sends a constant stream of messages with the current world time.

Perhaps the most important of the eventOut messages sent by the TimeSensor is fraction_changed. The value emitted by fraction_changed is an SFFloat, and is a number in the range of 0 to 1.0, indicating how much of the overall TimeSensor cycleInterval has been completed. For example, if we setup a timer with a cycleInterval value of 30, so that it cycles every half minute, and then start the timer, we'll immediately get a fraction_changed eventOut with a value of 0 – because the cycle has just started. After a second, we'll be getting a fraction_changed with a value of 0.03333334, or 1/30. After six seconds, we'll be getting a fraction_changed value of 0.2, after 15 seconds a value of 0.5, and, after 30 seconds, a value of 1.0. Then – if loop is TRUE – fraction_changed will sweep back around to 0, as the TimeSensor resets and begins counting up once again.

The messages sent by fraction_changed are the key to understanding animations in VRML. These little "slices" of time can be used to direct the movement of objects in space, the playback of sounds, lights, and so on.

Interpreting Interpolators

Classical animation, done with a pen and ink, developed from frame-by-frame animation; every figure, drawn into every frame, would be moved – subtly – from frame to frame. Through this gentle technique the illusion of motion would appear. While it's possible to do this in computer graphics – and even in VRML – a lot of the "grunt work" involved in laboriously animating a character can be given to the computer. The computer can take *key frame* information, that is, the snapshot position of an object at two points in time, and then calculate all of the intermediate motion required to move through space from the first point to the second point. This calculation of intermediate motion is more commonly known as *interpolation*; the computer uses mathematics to guess (interpolate) smooth motion from two snapshots.

VRML has a class of nodes which handle interpolation; all are designed to be used in conjunction with the TimeSensor node to produce smooth motion from key frame

information. Each of the interpolator nodes has two fields for storing the key frame information and the data associated with the key frames. Here's the definition of the PositionInterpolator node, which moves an translates an object between two positions:

```
PositionInterpolator {  # Definition
      key []                  # MFFloat, exposedField
      keyValue []       # MFVec3f, exposedField
      set_fraction            # SFFloat, eventIn
      value_changed           # SFVec3f, eventOut
}
```

The key field contains the timing of the "snapshots" of the key frames – and there must always be at least two values in the key field; the browser won't be able to calculate intermediate values unless you give it at least two points to work with. The total timing in the key field goes from 0 to 1.0, in other words, from the start of the animation through till the end of the animation.

The keyValue field contains a list of the translation values for the key frames. For instance, if started we wanted to translate an object from its origin to a point two units above, the first value in the keyValue field would be 0 0 0, and the second value would be 0 2 0. These values will get routed into the translation field of a Transform node.

The set_fraction eventIn normally receives input from the fraction_changed eventOut of an associated TimeSensor node; as the TimeSensor goes through its cycle, it continuously sends fraction_changed eventOut messages to the set_fraction eventIn. That's how animations can be controlled by TimeSensors.

Finally, value_changed continuously emits the results of the interpolator; as set_fraction receives different inputs, value_changed transmit different outputs. The eventOut transmits a message with a SFVec3f data type, the same data type as the translation field of the Transform node; most often that's where this message will be routed.

A basic example will help to clarify all of the this conceptual groundwork on TimeSensor and PositionInterpolator. We'll create a green Sphere and slowly "float" it upward, over a ten-second period of time. We'll start the TimeSensor automatically, and we'll put the Shape node for the Sphere inside of a Transform node, so that we can route information from the PositionInterpolator to it, and change its translation. Here's how it looks:

```
#VRML V2.0 utf8
# This is the first example on interpolators
#Here's the transform for green Sphere
DEF SPHERE_XFORM Transform {
      children [
            # Define green sphere
            Shape {
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0
                        }
                  }
                  geometry Sphere { }
```

```
            }
      ]
}
# Create an auto-starting, ten second timer
DEF TIMER TimeSensor {
      loop TRUE
      cycleInterval 10  # ten second timer
      startTime   1     # automatic begin
      stopTime    0     # automatic begin
}
# Move the Sphere from its origin, up 10 units.
DEF SPHERE_MOVE PositionInterpolator {
      key [ 0, 1 ]      # start, stop keyframes
      keyValue [ 0 0 0, 0 10 0 ]    # start, stop position values
}
# First we ROUTE the TimeSensor into the PositionInterpolator
ROUTE TIMER.fraction_changed TO SPHERE_MOVE.set_fraction
# Last we ROUTE the PositionInterpolator into the Transform
ROUTE SPHERE_MOVE.value_changed TO SPHERE_XFORM.set_translation
```

We didn't use any of the fields in the Transform node – other than children – but we need it just the same, because the PositionInterpolator will be sending messages to it, changing the value of the translation field as the TimeSensor runs. We should now see a green Sphere that rises upward at one unit per second.

You may find that the green Sphere rolls off the top of the screen, so you'll need to pull back a bit to get the grand view.

While a good start, this animation looks rather crude; at the end of the cycle the Sphere disappears and reappears at its start point. If we change the values in the key and keyValue fields of the PositionInterpolator node, we can get the Sphere to loop up and down. We'd do this by adding a key frame for the mid-way point in the animation, and ending the animation where it starts, at the origin. That might look like this:

```
#VRML V2.0 utf8
# This is the second example on interpolators
#Here's the transform for green Sphere
DEF SPHERE_XFORM Transform {
      children [
            # Define green sphere
            Shape {
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0
                        }
                  }
                  geometry Sphere { }
            }
      ]
}
# Create an auto-starting, ten second timer
DEF TIMER TimeSensor {
      loop TRUE
      cycleInterval 10  # ten second timer
```

```
      startTime   1     # automatic begin
      stopTime    0     # automatic begin
}
# Move the Sphere from its origin, up 10 units.
DEF SPHERE_MOVE PositionInterpolator {
      key [ 0, 0.5, 1 ] # start, stop keyframes
      keyValue [ 0 0 0, 0 10 0, 0 0 0 ]   # start, midway, stop
}
# First we ROUTE the TimeSensor into the PositionInterpolator
ROUTE TIMER.fraction_changed TO SPHERE_MOVE.set_fraction
# Last we ROUTE the PositionInterpolator into the Transform
ROUTE SPHERE_MOVE.value_changed TO SPHERE_XFORM.set_translation
```

Now the Sphere moves smoothly both up and down.

Feeling Gravity's Pull

Although interpolators can create linear motions – such as our last example - they can also create examples of acceleration and deceleration; they can be used to model things like objects falling to the ground, bouncing, and so on. All we need do is to add the appropriate key frame information, and the interpolator will do the rest. While the interpolator always moves linearly, we can "bunch" interpolator points together to give the feel of acceleration.

An object's acceleration when falling is governed by the law of falling bodies; on Earth's surface an object falls at a rate of 9.8 meters/second squared. In the first second, a body falls 9.8 meters; in the first half second, it falls one-quarter that distance, or 2.4 meters. In the first quarter second, it falls one-sixteenth that distance, or 0.6125 meters. That's enough information for us to construct a reasonably realistic PositionInterpolator. Here again, the same example, but adjusted for gravity:

```
#VRML V2.0 utf8
# This is the third example on interpolators
#Here's the transform for green Sphere
DEF SPHERE_XFORM Transform {
      children [
            # Define green sphere
            Shape {
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0
                        }
                  }
                  geometry Sphere { }
            }
      ]
}
# Create an auto-starting, ten second timer
DEF TIMER TimeSensor {
      loop TRUE
      cycleInterval 10  # ten second timer
      startTime   1     # automatic begin
      stopTime    0     # automatic begin
```

```
}
# Move the Sphere from its origin, up 10 units.
# We're using gravity-based readings
DEF SPHERE_MOVE PositionInterpolator {
      key [ 0, 0.25, 0.375, 0.5, 0.625, 0.75,  1 ]    # many keys
      keyValue [ 0 0 0, # origin
                  0 7.5 0, # fast as it climbs
                  0 9.38 0, # slows close to top
                  0 10 0,  # at zenith
                  0 9.38 0,
                  0 7.5 0,
                  0 0 0 ]
}
# First we ROUTE the TimeSensor into the PositionInterpolator
ROUTE TIMER.fraction_changed TO SPHERE_MOVE.set_fraction
# Last we ROUTE the PositionInterpolator into the Transform
ROUTE SPHERE_MOVE.value_changed TO SPHERE_XFORM.set_translation
```

Do we get realistic motion? It's hardly perfect, that's for sure, but it's starting to look realistic. The more points you add to the interpolator, the greater the realism; instead of seven key frames, you might want to try using twenty-five. This is something that computers handle with great efficiency, while we humans get bored crunching that many numbers!

With a small change, you can give the Sphere some thrust outward in the X axis, and now you've got something that looks like a cannon ball being launched:

```
#VRML V2.0 utf8
# This is the fourth example on interpolators
#Here's the transform for green Sphere
DEF SPHERE_XFORM Transform {
      children [
            # Define green sphere
            Shape {
                  appearance Appearance {
                        material Material {
                              diffuseColor 0 1 0
                        }
                  }
                  geometry Sphere { }
            }
      ]
}
# Create an auto-starting, ten second timer
DEF TIMER TimeSensor {
      loop TRUE
      cycleInterval 5   # five second timer
      startTime   1     # automatic begin
      stopTime    0     # automatic begin
}
# Move the Sphere from its origin, up 10 units.
# We're using gravity-based readings
DEF SPHERE_MOVE PositionInterpolator {
      key [ 0, 0.25, 0.375, 0.5, 0.625, 0.75,  1 ]    # many keys
      keyValue [ 0 0 0, # origin
```

```
                    25 7.5 0, # fast as it climbs
                    37.5 9.38 0, # slows close to top
                    50 10 0,  # at zenith
                    62.5 9.38 0,
                    75 7.5 0,
                    100 0 0 ]
}
# First we ROUTE the TimeSensor into the PositionInterpolator
ROUTE TIMER.fraction_changed TO SPHERE_MOVE.set_fraction
# Last we ROUTE the PositionInterpolator into the Transform
ROUTE SPHERE_MOVE.value_changed TO SPHERE_XFORM.set_translation
```

We've given the ball a completely uniform thrust in X, so we should simply see it move out, and away from the origin, then immediately come back and do it again. It does look like a cannon ball – captured in slow-motion.

You Say You Want a Revolution?

The OrientationInterpolator node allows you to animate the orientation of an object, its pitch, yaw and roll. The node is almost identical to the PositionInterpolator:

```
OrientationInterpolator {     # Definition
      key []                  # MFFloat, exposedField
      keyValue []       # MFRotation, exposedField
      set_fraction            # SFFloat, eventIn
      value_changed           # SFVec3f, eventOut
}
```

Again, the key field contains the timing of the "snapshots" of the key frames. For instance, if started we wanted to rotate an object through a full circle of yaw, we'd need to define three keys, of 0, 0.5, and 1 – with rotations in a full circle, you need to define a mid-point that describes the direction of the rotation.

.The keyValue field contains a list of the rotation values for the key frames. In a rotation through a full circle of yaw, we'd need a starting SFRotation of 0 1 0 0, indicating a rotation around the Y axis, of 0 radians – zero degrees. We'd then establish a mid-point at 180 degrees or 3.14 radians, with a SFRotation of 0 1 0 3.14. We'd complete the rotation by using a SFRotation of 0 1 0 6.28, indicating a full 360 degrees of yaw.

The set_fraction eventIn and value_changed eventOut behave just as they do for the PositionInterpolator. The data type of value_changed is SFRotation; usually, this event is routed into the rotation field of a Transform node.

Let's start with a basic example, adapted from our work on texture maps in chapter 11. Let's take our model of the Earth and make it rotate. To do this we need to put a Transform node around the Shape node which creates the Earth – so that we can route changes in rotation values to it. We need a TimeSensor – we'll let it cycle every sixty seconds – and we'll need an OrientationInterpolator that will run through a full 360 degrees of rotation. Here's how that might look:

```
      #VRML V2.0 utf8
# This is the fifth example on interpolators
DEF EARTH_XFORM Transform {
      children [
                  # Create semi-realistic model of Earth
Shape {           #  Create a visible shape
                        appearance Appearance {
                              texture ImageTexture { # texture
                                    url [ "worldmap.jpg" ]
                              }
                        }
                        geometry Sphere { }     # Make Earth!
}
            ]
      }
      # Create an auto-starting, sixty second timer
      DEF TIMER TimeSensor {
            loop TRUE
            cycleInterval 60  # sixty-second timer
            startTime   1     # automatic begin
            stopTime    0     # automatic begin
      }
      # Move the Sphere through 360 degrees of rotation
      DEF SPHERE_ROTATE OrientationInterpolator {
            key [ 0, 0.5, 1 ] # start, stop keyframes
            keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
      }
      # First we ROUTE the TimeSensor into the OrientationInterpolator
      ROUTE TIMER.fraction_changed TO SPHERE_ROTATE.set_fraction
      # Last we ROUTE the OrientationInterpolator into the Transform
      ROUTE SPHERE_ROTATE.value_changed TO EARTH_XFORM.set_rotation
```

The TimeSensor generates a stream of fraction_changed events, which get routed into the OrientationInterpolator; the OrientationInterpolator calculates a new rotation value from the key frame information, and that information is routed to the rotation field of the Transform node surrounding the Earth model. In the browser, the world spins, making a full revolution once a minute.

Now let's try something a little bit more complicated; let's try to model the Earth and the Moon together. The Earth rotates every 24 hours – every minute in our model – while the Moon circles the Earth every 29.25 days, or, in this model, every 1755 seconds. In our last example we created the rotation of the Earth, so now we need to create the rotation of the Moon – at a point some distance away from the Earth. That means we must put the Moon inside of two Transform nodes, the outermost of which handles the rotation, and inside that we nest another Transform node, which positions the Moon away from the Earth. This nesting of Transform nodes ensures that the Moon circles the Earth.

The model might look like this:

```
#VRML V2.0 utf8
# This is the sixth example on interpolators
DEF EARTH_XFORM Transform {
      children [
```

```
                  # Create semi-realistic model of Earth
Shape {           #  Create a visible shape
                        appearance Appearance {
                              texture ImageTexture { # texture
                                    url [ "worldmap.jpg" ]
                              }
                        }
                        geometry Sphere { }     # Make Earth!
}
            ]
      }
# Create the Moon in nested transforms
# This transform handles the changing rotation
DEF MOON_XFORM Transform {
      children [
                  # This transform handles the constant translation
                  Transform {
                        children [
                              # Create fake model of the Moon
Shape {
                                    appearance Appearance {
                              material Material {
                                          diffuseColor 0.5 0.5
0.5
                                    }
                              }
                              geometry Sphere { radius 0.3 } #
small
}
                        ]
                        translation 0 0 10 # out in front
                  }
            ]
      }
      # Create an auto-starting, sixty second timer
      DEF EARTH_TIMER TimeSensor {
            loop TRUE
            cycleInterval 60  # sixty-second timer
            startTime   1     # automatic begin
            stopTime    0     # automatic begin
      }
      # Create an auto-starting, 1755 second timer
      DEF MOON_TIMER TimeSensor {
            loop TRUE
            cycleInterval 1755      # 29.5 days
            startTime   1     # automatic begin
            stopTime    0     # automatic begin
      }
      # Move the Earth through 360 degrees of rotation
      DEF SPHERE_ROTATE OrientationInterpolator {
            key [ 0, 0.5, 1 ] # start, stop keyframes
            keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
      }
      # Move the Moon through 360 degrees of orbit
      DEF MOON_ORBIT OrientationInterpolator {
            key [ 0, 0.5, 1 ] # start, stop keyframes
            keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
```

```
}
# First we ROUTE the earth timer into the OrientationInterpolator
ROUTE EARTH_TIMER.fraction_changed TO SPHERE_ROTATE.set_fraction
# next we ROUTE the OrientationInterpolator into the Transform
ROUTE SPHERE_ROTATE.value_changed TO EARTH_XFORM.set_rotation
# then we ROUTE the moon timer into the OrientationInterpolator
ROUTE MOON_TIMER.fraction_changed TO MOON_ORBIT.set_fraction
# last we ROUTE the OrientationInterpolator into the Transform
ROUTE MOON_ORBIT.value_changed TO MOON_XFORM.set_rotation
```

We've doubled the number of TimeSensor nodes and ROUTES that we've used; we need separate a TimeSensor for the Earth and the Moon, and we need to two routes for each pair of TimeSensor and OrientationInterpolator nodes.

The Moon is rotating rather slowly with respect to the Earth, so you may have to watch it for a bit before you see it changing – and it works best if you shift your viewpoint to look at both bodies from above.

The Many-Colored Moon

One of the more playful of the interpolator nodes is the ColorInterpolator. As the name implies, the ColorInterpolator allows you to cycle through a range of SFColor values, based on key frame information. Here's how it looks:

```
ColorInterpolator {     # Definition
      key []                  # MFFloat, exposedField
      keyValue []       # MFColor, exposedField
      set_fraction            # SFFloat, eventIn
      value_changed           # SFVec3f, eventOut
}
```

Look familiar? The key field contains the key frame timing, while the keyValue field contains a list of the SFColor values for each key frame. Events set_fraction and value_changed work as expected; the value_changed event sends a message with a data type of SFColor – which quite often gets routed to the diffuseColor field of a Material node.

Let's use the ColorInterpolator with our previous example to make the Moon pass through throbbing red, green and blue colors. We'll use the Earth rotation TimeSensor – you can route an eventOut to multiple eventIn, something known as *fan-out* - and get double-duty from it. Here's how that might look:

```
#VRML V2.0 utf8
# This is the seventh example on interpolators
DEF EARTH_XFORM Transform {
      children [
                  # Create semi-realistic model of Earth
Shape {           #  Create a visible shape
                        appearance Appearance {
                              texture ImageTexture { # texture
                                    url [ "worldmap.jpg" ]
```

```
                        }
                  }
                  geometry Sphere { }     # Make Earth!
}
      ]
}
# Create the Moon in nested transforms
# This transform handles the changing rotation
DEF MOON_XFORM Transform {
      children [
            # This transform handles the constant translation
            Transform {
                  children [
                        # Create fake model of the Moon
Shape {
                              appearance Appearance {
                        material DEF MOON_COLOR Material {
                                    diffuseColor 1 0 0 #
start Red
                                          }
                              }
                              geometry Sphere { radius 0.3 } #
small
}
                  ]
                  translation 0 0 10 # out in front
            }
      ]
}
# Create an auto-starting, sixty second timer
DEF EARTH_TIMER TimeSensor {
      loop TRUE
      cycleInterval 60  # sixty-second timer
      startTime   1     # automatic begin
      stopTime    0     # automatic begin
}
# Create an auto-starting, 1755 second timer
DEF MOON_TIMER TimeSensor {
      loop TRUE
      cycleInterval 1755      # 29.5 days
      startTime   1     # automatic begin
      stopTime    0     # automatic begin
}
# Move the Earth through 360 degrees of rotation
DEF SPHERE_ROTATE OrientationInterpolator {
      key [ 0, 0.5, 1 ] # start, mid, stop keyframes
      keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
}
# Move the Moon through 360 degrees of orbit
DEF MOON_ORBIT OrientationInterpolator {
      key [ 0, 0.5, 1 ] # start, mid, stop keyframes
      keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
}
DEF MOON_PULSATE ColorInterpolator {
      key [ 0, 0.5, 1 ] # start, mid, stop keyframes
      keyValue [ 1 0 0, 0 1 0, 0 0 1 ]
}
```

```
      # First we ROUTE the earth timer into the OrientationInterpolator
      ROUTE EARTH_TIMER.fraction_changed TO SPHERE_ROTATE.set_fraction
      # next we ROUTE the OrientationInterpolator into the Transform
      ROUTE SPHERE_ROTATE.value_changed TO EARTH_XFORM.set_rotation
      # then we ROUTE the moon timer into the OrientationInterpolator
      ROUTE MOON_TIMER.fraction_changed TO MOON_ORBIT.set_fraction
      # then we ROUTE the OrientationInterpolator into the Transform
      ROUTE MOON_ORBIT.value_changed TO MOON_XFORM.set_rotation
      # then we ROUTE the earth timer into the ColorInterpolator
      ROUTE EARTH_TIMER.fraction_changed TO MOON_PULSATE.set_fraction
      # then we ROUTE the ColorInterpolator into the Material
      ROUTE MOON_PULSATE.value_changed TO MOON_COLOR.set_diffuseColor
```

We added a ColorInterpolator, and two more ROUTE statements, which connect the Earth's rotation TimeSensor to the ColorInterpolator, and then take the output of the ColorInterpolator and send it to the Material node associated with the Moon.

The Moon is now pulsating red, green and blue much faster than it orbits the Earth. Because we didn't match the values at the beginning and the end of the animation, the color jumps from blue to red at the end of the cycle. That's easy to fix, though – all you need to do is add more key and keyValue information to the ColorInterpolator node.

The Vanishing Orb

Another of the interpolators, the ScalarInterpolator, animates between a range of SFFloat values. Here's how it looks:

```
      ScalarInterpolator {    # Definition
            key []                  # MFFloat, exposedField
            keyValue []       # MFFloat, exposedField
            set_fraction            # SFFloat, eventIn
            value_changed           # SFVec3f, eventOut
      }
```

This is getting pretty dull, isn't it? The key field contains the key frame timing, while the keyValue field contains a list of the SFFloat values for each key frame. Events set_fraction and value_changed work as expected; the value_changed event sends a message with a data type of SFFloat – which can get routed to any exposedField of data type of SFFloat, such as the Material node's shininess and transparency fields. Let's take the previous example and modify it – removing the ColorInterpolator and adding a ScalarInterpolator which gets routed to the transparency field of the Moon's Material node; that way the Moon will fade out of and back into existence every 60 seconds:

```
#VRML V2.0 utf8
# This is the eighth example on interpolators
DEF EARTH_XFORM Transform {
      children [
                  # Create semi-realistic model of Earth
Shape {           #  Create a visible shape
                        appearance Appearance {
                              texture ImageTexture { # texture
                                    url [ "worldmap.jpg" ]
```

```
                              }
                        }
                        geometry Sphere { }     # Make Earth!
}
            ]
      }
# Create the Moon in nested transforms
# This transform handles the changing rotation
DEF MOON_XFORM Transform {
      children [
                  # This transform handles the constant translation
                  Transform {
                        children [
                              # Create fake model of the Moon
Shape {
                                    appearance Appearance {
                              material DEF MOON_COLOR Material {
                                          diffuseColor 0.5 0.5
0.5 # gray
                                    }
                              }
                              geometry Sphere { radius 0.3 } #
small
}
                        ]
                        translation 0 0 10 # out in front
                  }
            ]
      }
      # Create an auto-starting, sixty second timer
      DEF EARTH_TIMER TimeSensor {
            loop TRUE
            cycleInterval 60  # sixty-second timer
            startTime   1     # automatic begin
            stopTime    0     # automatic begin
      }
      # Create an auto-starting, 1755 second timer
      DEF MOON_TIMER TimeSensor {
            loop TRUE
            cycleInterval 1755      # 29.5 days
            startTime   1     # automatic begin
            stopTime    0     # automatic begin
      }
      # Move the Earth through 360 degrees of rotation
      DEF SPHERE_ROTATE OrientationInterpolator {
            key [ 0, 0.5, 1 ] # start, mid, stop keyframes
            keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
      }
      # Move the Moon through 360 degrees of orbit
      DEF MOON_ORBIT OrientationInterpolator {
            key [ 0, 0.5, 1 ] # start, mid, stop keyframes
            keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]
      }
      # Make the moon go transparent once every 60 seconds
      DEF MOON_VANISH ScalarInterpolator {
            key [ 0, 0.5, 1 ] # start, mid, stop keyframes
            keyValue [ 0, 1, 0 ] # opaque, transparent, opaque
```

```
}
# First we ROUTE the earth timer into the OrientationInterpolator
ROUTE EARTH_TIMER.fraction_changed TO SPHERE_ROTATE.set_fraction
# next we ROUTE the OrientationInterpolator into the Transform
ROUTE SPHERE_ROTATE.value_changed TO EARTH_XFORM.set_rotation
# then we ROUTE the moon timer into the OrientationInterpolator
ROUTE MOON_TIMER.fraction_changed TO MOON_ORBIT.set_fraction
# then we ROUTE the OrientationInterpolator into the Transform
ROUTE MOON_ORBIT.value_changed TO MOON_XFORM.set_rotation
# then we ROUTE the earth timer into the ScalarInterpolator
ROUTE EARTH_TIMER.fraction_changed TO MOON_VANISH.set_fraction
# then we ROUTE the ScalarInterpolator into the Material
ROUTE MOON_VANISH.value_changed TO MOON_COLOR.set_transparency
```

And now, the Moon vanishes at the 30 second mark of every minute, fading slowly out of view, and returning slowly.

Might Morphin' Power Pyramids

One of the most impressive of all of the interpolators is the CoordinateInterpolator; it allows you to change the shape of objects in real-time. We're all familiar with this technique, known as "polymorphic in-betweening", or *morphing*, film director James Cameron used to give the T1000 in *Terminator 2: Judgement Day* its incredible realism.

The CoordinateInterpolator works in conjunction with a Coordinate node, and changes the field point values within the node as the interpolator cycles through its range. Here's how it looks – and yes, it'll seem awfully familiar by now:

```
CoordinateInterpolator {      # Definition
      key []                  # MFFloat, exposedField
      keyValue []       # MFFloat, exposedField
      set_fraction            # SFFloat, eventIn
      value_changed           # SFVec3f, eventOut
}
```

The key field contains the key frame timing, while the keyValue field contains a list of the MFVec3f values for each key frame, that is, the set of points which make up the form of an object. Events set_fraction and value_changed work as expected; the value_changed event sends a message with a data type of MFVec3f; this is usually routed into the point field of a Coordinate node.

Examples with the CoordinateInterpolator can get very complex, so we'll adapt the example of our pyramid from chapter 10, which has only five points. The CoordinateInterpolator will change the value of only one of those points, the top of the pyramid, so that as a TimeSensor cycles through its 10-second cycle, the top of the pyramid will appear to go up and down. Here's how that might look:

```
#VRML V2.0 utf8
# This is the ninth example on interpolators
Shape {
      appearance Appearance {
```

```
                  material Material {
                        diffuseColor 0.5 0.5 0.5 # gray
                  }
            }
            geometry IndexedFaceSet {     # complex shape!
coord DEF MORPH_POINTS 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
            }
      }

      # Create an auto-starting, ten second timer
      DEF MORPH_TIMER TimeSensor {
            loop TRUE
            cycleInterval 10  # sixty-second timer
            startTime   1     # automatic begin
            stopTime    0     # automatic begin
      }

      # This CoordinateInterpolator defines the pyramid
      # Three times, first and last are the same.
      DEF MORPH_MAKER CoordinateInterpolator {
            key [ 0, 0.5, 1 ] # start, mid, end
            keyValue [ 0 0 0, 1 0 0, 1 0 –1, 0 0 –1, 0.5 0.5 -0.5,
                        0 0 0, 1 0 0, 1 0 –1, 0 0 –1, 0.5 2 -0.5,
0 0 0, 1 0 0, 1 0 –1, 0 0 –1, 0.5 0.5 -0.5 ]
      }
      # First we ROUTE the morph timer into the CoordinateInterpolator
      ROUTE MORPH_TIMER.fraction_changed TO MORPH_MAKER.set_fraction
      # next we ROUTE the CoordinateInterpolator into the Coordinate
      ROUTE MORPH_MAKER.value_changed TO MORPH_POINTS.set_point
```

We define a CoordinateInterpolator which describes the pyramid three times; the mid-point key frame elevates the top of the pyramid – the last point - by one-and-a-half units. The output of this CoordinateInterpolator is routed into the Coordinate node within the IndexedFaceSet – effectively changing the form created by the IndexedFaceSet. To drive the whole thing, we also create a self-starting TimeSensor that cycles every ten seconds.

Although it may look much less sophisticated, there's absolutely no difference between this example and the liquid transformations of the T1000!

What Every Director Wants

We've just begun to explore interactivity. Animation with interpolators is an important part of a compelling experience, but – far more important are the story and narrative associated that form a coherent world. In the virtual world as in Hollywood, a good story begins with a good script, and that's what we're going to learn about next – so fasten your seatbelts, we're about to launch into the world of VRML programming with JavaScript…