CHAPTER 13

Writing custom behaviors

Some behaviors automatically execute complex code to modify objects within the scenegraph, so care must be taken to ensure that behavior processing does not bog down application performance. With careful design and knowledge of some of the limitations, behaviors can be a powerful asset in quickly developing or prototyping application logic.

By the end of this chapter you should have a good sense of how to develop your own behaviors. By mixing and matching your own and the built-in behaviors, you should be able to design your application logic within Java 3D's behavior model.

13.1 THE BEHAVIORTEST EXAMPLE

There are occasions when the built-in behaviors do not provide enough functionality to capture the logic of your application. By creating your own classes derived from `Behavior`, you can easily integrate your application logic into Java 3D's behavior processing framework.

The `BehaviorTest` example application uses four behaviors: the built-in `RotationInterpolator` and three custom behaviors of varying complexity: `ObjectSizeBehavior`, `ExplodeBehavior`, and `StretchBehavior`. See figure 13.1.

`ObjectSizeBehavior` is the simplest, and it calculates the smallest `BoundingBox` that encloses a `Shape3D`'s geometry. The `BoundingBox` is recalculated every 20 frames, and the size of the `BoundingBox` is written to standard output. Note that the basic anatomy of a behavior described in section 11.3 is adhered to here.

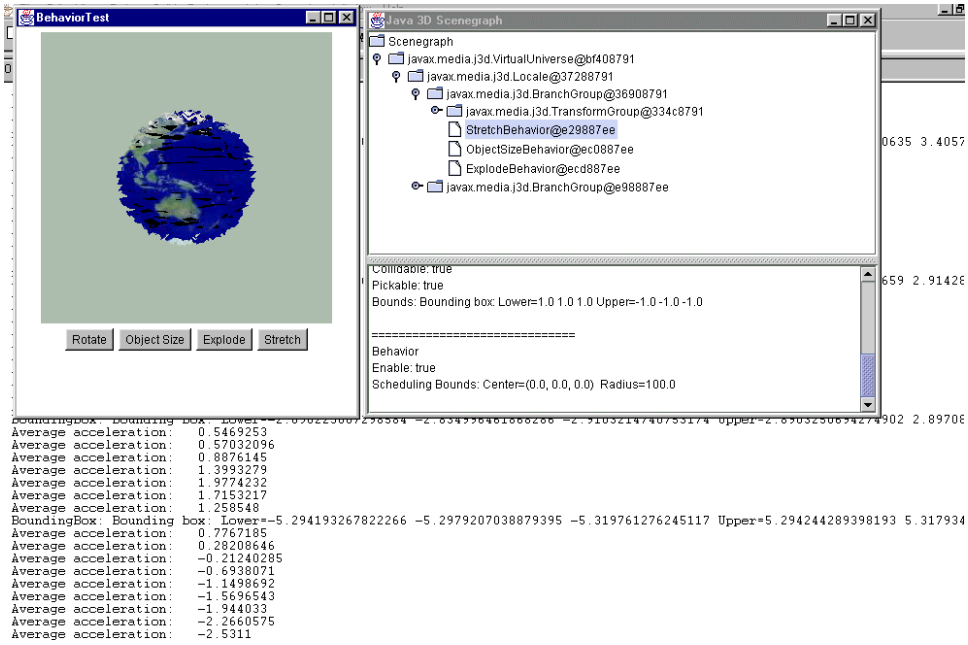

Figure 13.1 The `BehaviorTest` example application. `StretchBehavior` is used to modify the geometry of the `Sphere` after every frame, while `ObjectSizeBehavior` reports the `Bounds` for the object after every 20 frames

`ExplodeBehavior` is more complex. Given a `Shape3D` object, it explodes the object after a specified number of milliseconds by rendering the `Shape3D` as points and modifying the coordinates within the `Shape3D`'s `GeometryArray`. The transparency of the object is gradually increased so that the object fades into the background.

`StretchBehavior` is the most complex of the custom behaviors. It operates upon a specified `GeometryArray` and animates the vertices within the array as if they were weights attached by springs to the origin. `StretchBehavior` listens for key presses and increases the acceleration of each vertex when a key is pressed. The increased acceleration causes the vertices to move away from the origin, which causes an increase in the restraining force from the spring. The vertices oscillate back and forth, finally coming to rest at their original position.

13.2 OBJECTSIZEBEHAVIOR

The `ObjectSizeBehavior` class implements a simple behavior that calculates and prints the size of an object based on the vertices in its `GeometryArray`.

From ObjectSizeBehavior.java

```
class ObjectSizeBehavior extends Behavior
{
//the wake up condition for the behavior
protected WakeupCondition   m_WakeupCondition = null;

//the GeometryArray for the Shape3D that we are querying
protected GeometryArray   m_GeometryArray = null;

//cache some information on the model to save reallocation
protected float[]       m_CoordinateArray = null;

protected BoundingBox    m_BoundingBox = null;
protected Point3d       m_Point = null;;

public ObjectSizeBehavior( GeometryArray geomArray )
{
 //save the GeometryArray that we are modifying
 m_GeometryArray = geomArray;

 //set the capability bits that the behavior requires
 m_GeometryArray.setCapability(
  GeometryArray.ALLOW_COORDINATE_READ );
 m_GeometryArray.setCapability(
  GeometryArray.ALLOW_COUNT_READ );

 //allocate an array for the coordinates
 m_CoordinateArray =
  new float[ 3 * m_GeometryArray.getVertexCount() ];

 //create the BoundingBox used to calculate the size of the object
 m_BoundingBox = new BoundingBox();

 //create a temporary point
 m_Point = new Point3d();

 //create the WakeupCriterion for the behavior
 WakeupCriterion criterionArray[] = new WakeupCriterion[1];
  criterionArray[0] = new WakeupOnElapsedFrames( 20 );

 //save the WakeupCriterion for the behavior
 m_WakeupCondition = new WakeupOr( criterionArray );
}

public void initialize()
{
 //apply the initial WakeupCriterion
 wakeupOn( m_WakeupCondition );
}

public void processStimulus( java.util.Enumeration criteria )
{
 while( criteria.hasMoreElements() )
 {
  WakeupCriterion wakeUp =
```

```
   (WakeupCriterion) criteria.nextElement();

  //every N frames, recalculate the bounds for the points
  //in the GeometryArray
  if( wakeUp instanceof WakeupOnElapsedFrames )
  {
   //get all the coordinates
   m_GeometryArray.getCoordinates( 0, m_CoordinateArray );

   //clear the old BoundingBox
   m_BoundingBox.setLower( 0,0,0 );
   m_BoundingBox.setUpper( 0,0,0 );

   //loop over every vertex and combine with the BoundingBox
   for( int n = 0; n < m_CoordinateArray.length; n+=3 )
   {
    m_Point.x = m_CoordinateArray[n];
    m_Point.y = m_CoordinateArray[n+1];
    m_Point.z = m_CoordinateArray[n+2];

    m_BoundingBox.combine( m_Point );
   }

   System.out.println( "BoundingBox: " + m_BoundingBox );
  }
 }

 //assign the next WakeUpCondition, so we are notified again
 wakeupOn( m_WakeupCondition );
}
}
```

To use the behavior one could write:

```
Sphere sphere = new Sphere( 3, Primitive.GENERATE_NORMALS | Primi-
tive.GENERATE_TEXTURE_COORDS, 32, app );

m_SizeBehavior = new ObjectSizeBehavior( (GeometryArray) sphere.get-
Shape().getGeometry() );
m_SizeBehavior.setSchedulingBounds( getApplicationBounds() );
objRoot.addChild( m_SizeBehavior );
```

This code snippet creates the behavior and passes the geometry for a `Sphere` to the constructor, sets the scheduling bounds for the behavior and adds it to the scenegraph. Do not forget to *add* the behavior to the scenegraph, or it will not get scheduled.

Output from the behavior is simply:

```
Bounding box: Lower=-5.048 -5.044 -5.069 Upper=5.040 5.060 5.069
Bounding box: Lower=-5.048 -5.044 -5.069 Upper=5.040 5.060 5.069
Bounding box: Lower=-5.048 -5.044 -5.069 Upper=5.040 5.060 5.069
```

The behavior verifies the size of the geometry for the `Shape3D` every 20 frames. Note that the behavior follows the general anatomy of a behavior as was described in section 11.3.

When writing a behavior you should be very aware of the computational cost of the processing within the `processStimulus` method and how often the behavior is likely to be invoked. The `ObjectSizeBehavior`'s `processStimulus` method is called once every 20 frames, so any processing that is performed is going to have a fairly big impact on application performance. Whenever possible, avoid creating `Objects` (using the `new` operator) within the `processStimulus` method if it is going to be invoked frequently. Any `Objects` created by the behavior using the `new` operator and not assigned to a member variable will have to be garbage-collected. Not only is creating objects a relatively costly operation, but garbage collection can cause your application to noticeably pause during rendering.

For example, instead of creating a new `BoundingBox`, which would have had size 0, a single `BoundingBox` object was resized using:

```
m_BoundingBox.setLower( 0,0,0 );
m_BoundingBox.setUpper( 0,0,0 );
```

With Java 3D in general, you should avoid burning (allocate, followed by garbage-collect) `Objects` as much as possible, and minimize the work that the garbage collector has to perform.

13.3 EXPLODEBEHAVIOR

The constructor for the `ExplodeBehavior` is as follows:

```
public ExplodeBehavior( Shape3D shape3D,
                        int nElapsedTime, int nNumFrames,
                        ExplosionListener listener )
```

The behavior attaches to the `Shape3D` specified and explodes the object after nElapsedTime milliseconds (figure 13.2). The explosion animation takes nNumFrames to complete, and, once complete, a notification is passed to the caller via an `ExplosionListener` interface method.

To model the simple explosion, the behavior switches the `Shape3D`'s appearance to rendering in points (by modifying the `PolygonAttributes`) and sets the point size (using `PointAttributes`). The transparency of the `Shape3D` is then set using `TransparencyAttributes`. The vertices of the `Shape3D`'s geometry are then moved away from the origin with a slight random bias in the $x+$, $y+$, and $z+$ direction.

The `ExplodeBehavior` moves through the following life cycle:

1. The behavior is created.
2. Initialize is called by Java 3D.
3. `WakeUp` condition is set to be `WakeupOnElapsedTime( n milliseconds )`.
4. `processStimulus` is called after *n* milliseconds.
5. The `Appearance` attributes are modified for the `Shape3D`.
6. The `WakeUp` condition is set to `WakeupOnElapsedFrames( 1 )`.

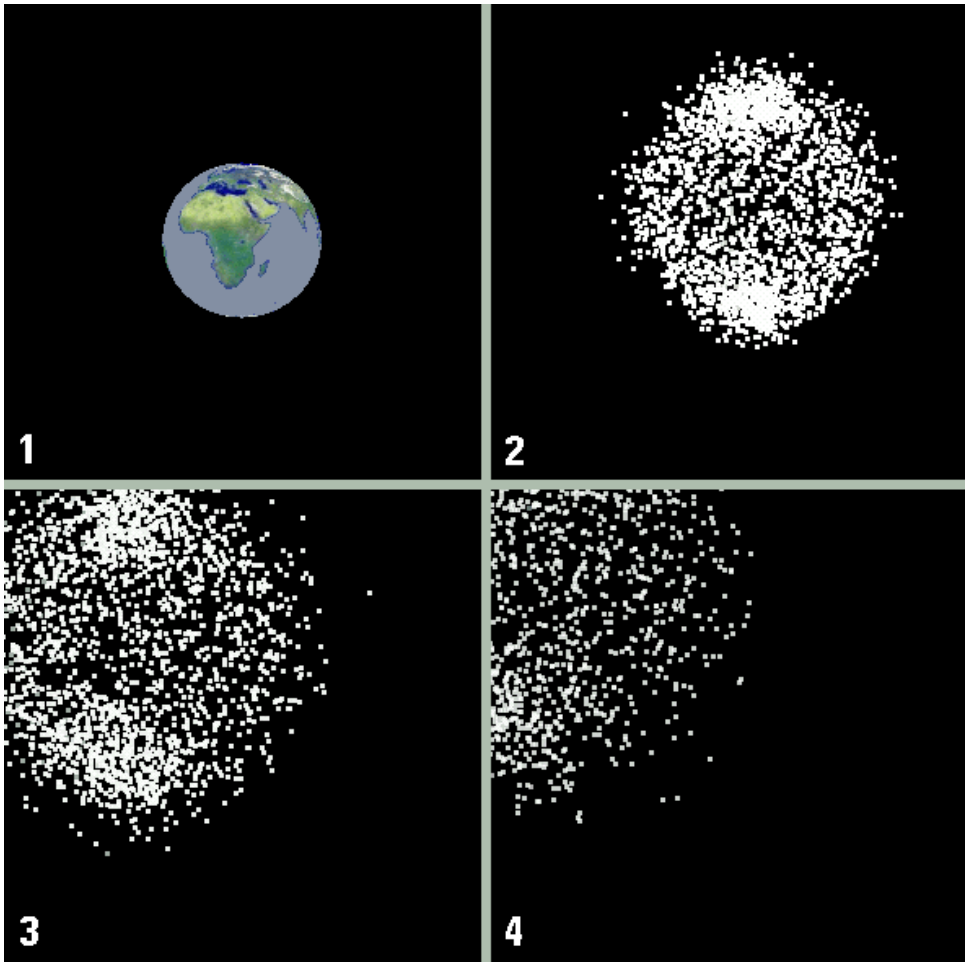

Figure 13.2 The `ExplodeBehavior`: Frame 1, the original `Shape3D`; frames 2–4, some frames of the explosion animation

7 `processStimulus` is called after every frame.

8 The `GeometryArray`'s vertex coordinates is modified.

9 Coordinates are reassigned.

10 If frame number < number of frames for animation

- Set the `WakeUp` condition to `WakeupOnElapsedFrames( 1 )`

11 Else

- Restore the original `Shape3D Appearance` and coordinates.
- Notify the `ExplosionListener` that the behavior is done.
- Call `setEnable( false )` to disabled the behavior.

The `processStimulus` method for the `ExplodeBehavior` is as follows.

From ExplodeBehavior.java

```
public void processStimulus( java.util.Enumeration criteria )
{
 while( criteria.hasMoreElements() )
 {
  WakeupCriterion wakeUp =
    (WakeupCriterion) criteria.nextElement();

  if( wakeUp instanceof WakeupOnElapsedTime )
  {
   //we are starting the explosion,
   //apply the appearance changes we require
   PolygonAttributes polyAttribs =
    new PolygonAttributes( PolygonAttributes.POLYGON_POINT,
                           PolygonAttributes.CULL_NONE, 0 );
   m_Shape3D.getAppearance().setPolygonAttributes( polyAttribs );

   PointAttributes pointAttribs = new PointAttributes( 3, false );
   m_Shape3D.getAppearance().setPointAttributes( pointAttribs );

   m_Shape3D.getAppearance().setTexture( null );

   m_TransparencyAttributes =
    new TransparencyAttributes( TransparencyAttributes.NICEST, 0 );
   m_TransparencyAttributes.setCapability(
     TransparencyAttributes.ALLOW_VALUE_WRITE );
   m_Shape3D.getAppearance().setTransparencyAttributes(
     m_TransparencyAttributes );
  }
  else
  {
   //we are mid explosion, modify the GeometryArray
   m_nFrameNumber++;

   m_GeometryArray.getCoordinates( 0, m_CoordinateArray );

   m_TransparencyAttributes.
    setTransparency( ((float) m_nFrameNumber) /
   ((float) m_nNumFrames) );
   m_Shape3D.getAppearance().
    setTransparencyAttributes( m_TransparencyAttributes );

   for( int n = 0; n < m_CoordinateArray.length; n+=3 )
   {
    m_Vector.x = m_CoordinateArray[n];
    m_Vector.y = m_CoordinateArray[n+1];
    m_Vector.z = m_CoordinateArray[n+2];

    m_Vector.normalize();

    m_CoordinateArray[n] += m_Vector.x * Math.random() +
     Math.random();
```

```
    m_CoordinateArray[n+1] += m_Vector.y * Math.random() +
     Math.random();
    m_CoordinateArray[n+2] += m_Vector.z * Math.random() +
     Math.random();
   }

   //assign the new coordinates
   m_GeometryArray.setCoordinates( 0, m_CoordinateArray );
  }
 }

 if( m_nFrameNumber < m_nNumFrames )
 {
  //assign the next WakeUpCondition, so we are notified again
  wakeupOn( m_FrameWakeupCondition );
 }
 else
 {
  //we are at the end of the explosion
  //reapply the original appearance and GeometryArray coordinates
  setEnable( false );
  m_Shape3D.setAppearance( m_Appearance );

  m_GeometryArray.setCoordinates( 0, m_OriginalCoordinateArray );

  m_OriginalCoordinateArray = null;
  m_GeometryArray = null;
  m_CoordinateArray = null;
  m_TransparencyAttributes = null;

  //if we have a listener notify them that we are done
  if( m_Listener != null )
   wakeupOn( m_Listener.onExplosionFinished( this, m_Shape3D ) );
 }
```

13.4 StretchBehavior

`StretchBehavior` implements a more complex behavior. The behavior modifies the coordinates within a `GeometryArray` based on simulated forces applied to the geometric model. Forces are modeled as springs from the origin to every vertex. Every vertex has a mass and an applied force, and hence an acceleration. Pressing a key will increase the acceleration at each vertex, upsetting the force equilibrium at vertices. The model will then start to oscillate in size under the influence of the springs. Because there are variations in mass between vertices, the model will distort slightly as it oscillates—the heavier vertices displacing less than the lighter ones. A damping effect is modeled by losing a portion of the vertex acceleration after each iteration. See figure 13.3.

NOTE This is a computationally expensive behavior.

`StretchBehavior` responds to two `WakeUp` conditions: after every frame and after a key press. The `WakeUp` conditions for the behavior are specified as follows:

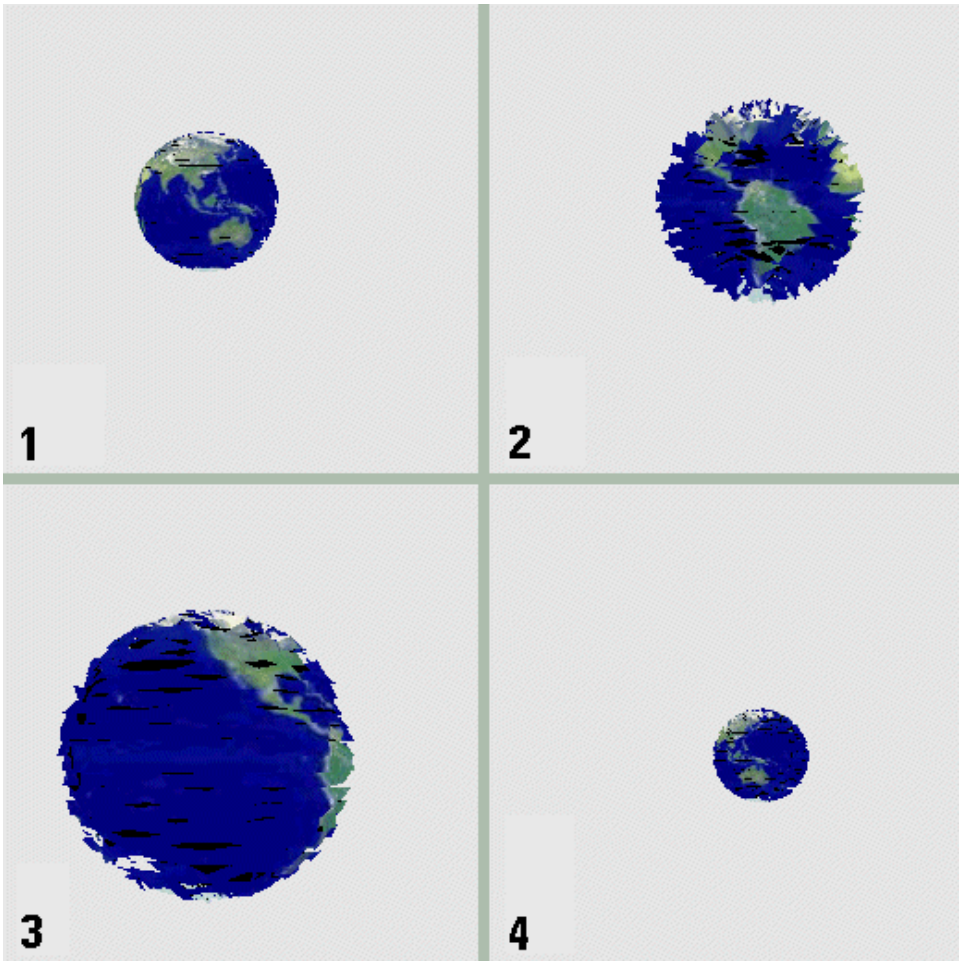

Figure 13.3 The `StretchBehavior`: Frame 1, the original `Shape3D`; frames 2–4, the vertices within the `Shape3D`'s geometry are oscillating as the vertices are affected by the springs from each vertex to the origin. The model is a `Sphere` primitive with an applied texture image. The `Sphere` was created with a resolution value of 32

```
//create the WakeupCriterion for the behavior
WakeupCriterion criterionArray[] = new WakeupCriterion[2];
criterionArray[0] = new WakeupOnAWTEvent( KeyEvent.KEY_PRESSED );
criterionArray[1] = new WakeupOnElapsedFrames( 1 );

//save the WakeupCriterion for the behavior
m_WakeupCondition = new WakeupOr( criterionArray );
```

As usual, the `WakeupCriterion` is passed to the behavior inside the `initialize` method:

```
public void initialize()
{
 //apply the initial WakeupCriterion
 wakeupOn( m_WakeupCondition );
}
```

The `processStimulus` method of the behavior, which is called on every frame and in response to a key press, performs all the basic physics calculations and updates the positions of the coordinates within the `GeometryArray`.

From StretchBehavior.java

```
public void processStimulus( java.util.Enumeration criteria )
{
 //update the positions of the vertices—regardless of criteria
 float elongation = 0;
 float force_spring = 0;
 float force_mass = 0;
 float force_sum = 0;
 float timeFactor = 0.1f;
 float accel_sum = 0;

 //loop over every vertex and calculate its new position
 //based on the sum of forces due to acceleration and the spring
 for( int n = 0; n < m_CoordinateArray.length; n+=3 )
 {
  m_Vector.x = m_CoordinateArray[n];
  m_Vector.y = m_CoordinateArray[n+1];
  m_Vector.z = m_CoordinateArray[n+2];

  //use squared lengths, as sqrt is costly
  elongation = m_LengthArray[n/3] - m_Vector.lengthSquared();

  //Fspring = k*Le
  force_spring = m_kSpringConstant * elongation;
  force_mass = m_AccelerationArray[n/3] * m_MassArray[n/3];

  //calculate resultant force
  force_sum = force_mass + force_spring;

  //a = F/m
  m_AccelerationArray[n/3] = (force_sum / m_MassArray[n/3]) *
   m_kAccelerationLossFactor;
  accel_sum += m_AccelerationArray[n/3];

  m_Vector.normalize();

  //apply a portion of the acceleration as change
  //in coordinate based on the normalized vector
  //from the origin to the vertex
  m_CoordinateArray[n] +=
    m_Vector.x * timeFactor * m_AccelerationArray[n/3];
  m_CoordinateArray[n+1] +=
    m_Vector.y * timeFactor * m_AccelerationArray[n/3];
```

```
 m_CoordinateArray[n+2] +=
   m_Vector.z * timeFactor * m_AccelerationArray[n/3];
}

//assign the new coordinates
m_GeometryArray.setCoordinates( 0, m_CoordinateArray );

while( criteria.hasMoreElements() )
{
 WakeupCriterion wakeUp =
  (WakeupCriterion) criteria.nextElement();

 //if a key was pressed increase the acceleration at the vertices
 //a little to upset the equilibrium
 if( wakeUp instanceof WakeupOnAWTEvent )
 {
  for( int n = 0; n < m_AccelerationArray.length; n++ )
   m_AccelerationArray[n] += 0.3f;
 }
 else
 {
  //otherwise, print the average acceleration
  System.out.print( "Average acceleration:\t"
  + accel_sum/m_AccelerationArray.length + "\n" );
 }
}

//assign the next WakeUpCondition, so we are notified again
wakeupOn( m_WakeupCondition );
}
```

After pressing a key has disturbed the equilibrium of the model, it can take a considerable length of time to return to equilibrium. In figure 13.4 the model took over 500 frames to stabilize.

13.5 USING BEHAVIORS FOR DEBUGGING

A library of custom `Behavior` classes can be a very useful debugging aid, as they can be quickly added and removed from the scenegraph as needed. It is a simple step to conditionally add the debugging behaviors for development builds and remove them for production builds. For example, I have used the following two behaviors extensively:

1. `BoundsBehavior` is attached to a scenegraph `Node` and creates a wire frame `ColorCube` or `Sphere` to graphically represent the `Bounds` (`BoundingBox` or `BoundingSphere`) for the object at runtime.
2. `FpsBehavior` can be added anywhere in the scenegraph and writes the rendered FPS to the standard output window.

Both behaviors can be found in the `org.selman.java3d.book` package and are illustrated in the `BehaviorTest` example application.

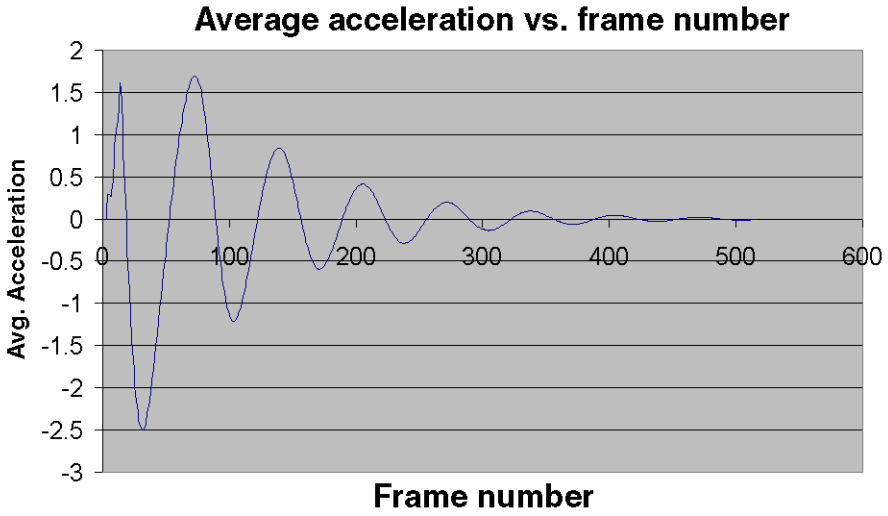

Figure 13.4 The `StretchBehavior` causes the `Sphere` to oscillate in size. By plotting the average vertex acceleration, you can see that the model took in excess of 500 frames to stabilize. The parameters used were Spring Constant 0.8, Acceleration Loss Factor 0.98, and Vertex Mass 50 + 2.5 (average)

13.5.1 Calculating the rendered FPS using a behavior

A useful method of displaying the rendered FPS in a Java 3D application is to add the following `Behavior` class anywhere within the scenegraph. A behavior-based calculation is easier to add and remove to a program than overriding the `Canvas3D` `postSwap` method.

NOTE If accuracy is paramount, `postSwap` may provide more accurate results because `Behavior` processing typically runs on an independent thread to rendering.

From FpsBehavior

```
//this class implements a simple behavior
//that output the rendered Frames Per Second.
public class FpsBehavior extends Behavior
{
 //the wake up condition for the behavior
 protected WakeupCondition  m_WakeupCondition = null;
 protected long        m_StartTime = 0;

 private final int       m_knReportInterval = 100;

 public FpsBehavior()
 {
  //save the WakeupCriterion for the behavior
  m_WakeupCondition =
```

```
    new WakeupOnElapsedFrames( m_knReportInterval );
  }

  public void initialize()
  {
   //apply the initial WakeupCriterion
   wakeupOn( m_WakeupCondition );
  }

  public void processStimulus( java.util.Enumeration criteria )
  {
   while( criteria.hasMoreElements() )
   {
    WakeupCriterion wakeUp =
     (WakeupCriterion) criteria.nextElement();

    //every N frames, report the FPS
    if( wakeUp instanceof WakeupOnElapsedFrames )
    {
     if( m_StartTime > 0 )
     {
      final long interval = System.currentTimeMillis() - m_StartTime;
      System.out.println( "FPS: " + (double) m_knReportInterval / (
       interval / 1000.0));
     }

     m_StartTime = System.currentTimeMillis();
    }
   }

   //assign the next WakeUpCondition, so we are notified again
   wakeupOn( m_WakeupCondition );
  }
}
```

13.6 SUMMARY

The `BehaviorTest` example allows many behaviors to affect a single texture-mapped `Sphere`. `RotationInterpolator` rotates the entire scene, `Object-SizeBehavior` prints the size of the `Sphere` every 20 frames, `ExplodeBehavior` explodes the `Sphere` every 10 seconds, `StretchBehavior` models the vertices of the `Sphere` as weights attached to springs anchored at the origin, and `BoundsBehavior` tracks the `Bounds` of the `Sphere`.

Tying all these behaviors together into a single application allows complex application logic to be built up from relatively simple building blocks. The interactions between the behaviors can be explored by running the example and switching the behaviors on and off using the AWT buttons at the bottom of the `Frame`.

I hope the examples presented in this section have demystified Java 3D's behaviors. You should now start breaking down your application logic into good, reusable, OO

chunks and distributing them across your scenegraph. You should aim to empower your scenegraph objects with the abilities to detect, process, and respond to user interactions.

Keep a careful eye on application performance at all times, because excessive behavior processing can slow your frame rate or make your application appear unresponsive. Do not be afraid of writing more complex behaviors that can affect whole classes of objects within your scenegraph. In this way you may be able to limit the number of behaviors in the scenegraph and use a manager design philosophy, where each behavior manages a given class of objects within the scenegraph, instead of attaching single instances of a behavior to a single object.