Fun with Java: Animated Sea Worms

Baldwin wraps up his series of lessons on animation. He has shown you how to use Java to write programs that produce smooth animation of the sprite animation and frame animation varieties. It's time for you to take that knowledge and do something fun with it. And don't forget, Java programming can be fun.

Published: December 3, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1466

- Preface
- Preview
- Discussion and Sample Programs
- Summary
- Complete Program Listing

Preface

Fun programming

This is one of the lessons in a miniseries that will concentrate on having fun while programming in Java.

This miniseries will include a variety of Java programming topics that fall in the category of *fun programming*. This particular lesson is the ninth in of a group of lessons that will teach you how to write animation programs in Java.

The first lesson in the group was entitled Fun with Java: Sprite Animation, Part 1. That lesson, plus the next seven lessons provided an in-depth explanation of the use of Java for doing both sprite and frame animation. The previous lesson was entitled Fun with Java: Frame Animation.

In this lesson, I will combine sprite animation with frame animation to produce animated sea worms that change their color on a random basis while swimming in a fish tank.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at Baldwin's Java Programming Tutorials.

Preview

This is one of a group of lessons that will teach you how to write animation programs in Java. These lessons will teach you how to write sprite animation, frame animation, and a combination of the two.

Slithering sea worms

The program that I will discuss in this lesson will use a combination of sprite animation and frame animation, along with some other techniques to cause a group of multi-colored sea worms to slither around in the fish tank. In addition to slithering, the sea worms will also change the color of different parts of their body, much like real sea creatures.

A screen shot of the output from this program is shown in Figure 1.

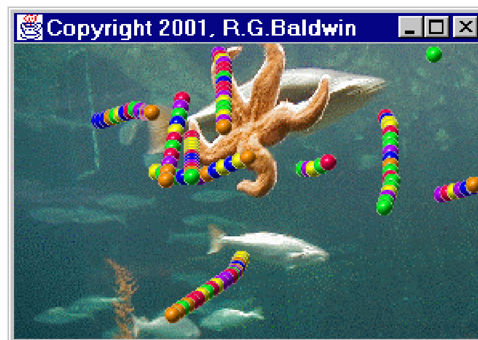

Figure 1. Animated sea worms in a fish tank.

Getting the GIF image files

Figure 2 shows the GIF image files that you will need to run the program.

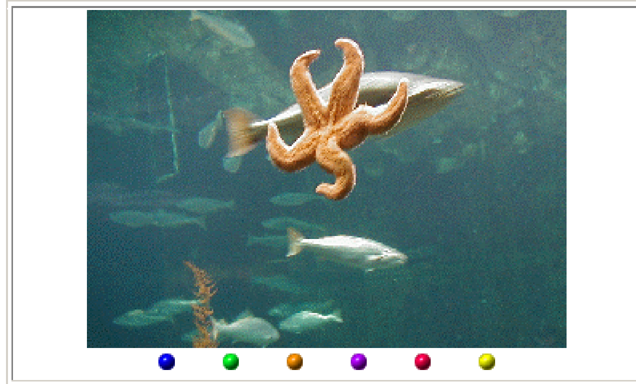

Figure 2. GIF image files that you will need.

You should be able to capture the images by right-clicking on them individually, and then saving them into files on your local disk. Having done that, you will need to rename the files to match the names that are hard-coded into the program.

Discussion and Sample Program

Although this program is quite long, most of the program is identical to the program named **Animate02**, which was discussed in the previous lesson.

Discuss new and different material

In this lesson, I will discuss only those parts of the program that are new or different from the previous program. However, I have provided a copy of the entire program in Listing 8 near the end of the lesson. You can copy it into a source file on your local disk, compile it, run it, and see the results.

Some history may be required

Because I will be discussing only that code that is new or different, it may be necessary for you to go back and study the previous programs named **Animate01** and **Animate02**, in order to understand this program.

Primary differences

The primary difference between this program and the previous program named **Animate02** can be summarized as follows:

The previous program animated 15 spherical sea creatures, which swam around and changed their color according to a specific algorithm having to do with selecting the next color, the duration of a color, etc.

This program animates a similar number of sea worms, which swim around and change their color also. However, this program uses a random number generator to determine the color for each segment of each sea worm.

Good news and bad news

As a result, the code for dealing with color in this program is somewhat simpler than in the previous program. On the other hand, this program contains the code necessary to create sea worms instead of spherical sea creatures. That code is a little more complicated than in the previous program.

All of the changes that were made to this program relative to the previous program named **Animate02** were made to the class named **Sprite**. In addition, a new class named **History** was defined for use in this program.

Discuss in fragments

As usual, I will discuss the program in fragments. I'm going to begin with the new **History** class.

Sea worms constructed from spheres

The mechanism used to draw the sea worms in this program is to draw a series of colored spheres each time the **Sprite** object is asked to draw itself *(each segment of a sea worm is a sphere)*.

The head is a new sphere

The head of the sea worm is a new sphere. The remaining portion of the sea worm is produced by redrawing a given number of the spheres that appeared in the previous animation cycles.

Discard the tail

For a given length worm, a new head is created during each animation cycle, and the sphere that makes up the end of the tail is discarded. This causes the sea worm to appear to move in a forward direction.

Historical data required

To accomplish this, it was necessary to retain historical information about where the head has been, and what color it was when it was there. That is the purpose of the class named **History**.

Each sprite owns a **Vector** collection of objects of the class **History**, with the number of elements in the collection being equal to the length of that particular sea worm.

Once during each animation cycle, a new **History** object is added to the end of the collection and the **History** object at the beginning of the collection is discarded.

The class named History

The **History** class is shown in Listing 1 below.

```
class History{
  Image image;
  int x;
  int y;

  History(Image image, int x, int y){
    this.image = image;
    this.x = x;
    this.y = y;
  }//end constructor
}//end class History
```

Listing 1

As you can see, each object instantiated from this class contains the coordinate values to be used to draw the segment of the worm described by the object along with a reference to the **Image** object used to provide the visual manifestation for that segment of the worm.

Sprites take care of themselves

The **Sprite** class is the workhorse of this program.

Every one of the sprites swimming around in the fish tank is an object of the class named **Sprite**. As is the typical objective in object-oriented programming, a sprite knows how to take care of itself.

What does a sprite know?

For example, an object of the **Sprite** class knows how to tell other objects about the space that it occupies in the fish tank. It knows how to tell other objects about its motion vector, which determines the speed and direction of its motion.

It knows how to use its motion vector in conjunction with a random number generator to incrementally advance its position to the next location in its movement through the water. In so doing, it knows how to protect itself from excessive speed.

It knows how to bounce if it runs into one of the walls of the fish tank. When this happens, it modifies its motion vector accordingly.

When requested to do so, it knows how to draw itself onto a graphics context that it receives along with the request.

When requested to do so, it can determine if its head has collided with the head of another sea worm sprite whose reference it receives along with the request.

Finally, it knows how to use an array of images to change how it looks over time.

Lots of similar code

Most of the code in this revised **Sprite** class is identical to the code in the **Sprite** class used in the previous program named **Animate02**. In keeping with the spirit of this lesson, I will not discuss the code that I discussed in the previous lessons. Rather, for the most part, I will discuss only that code that is new or different. When you see **//...** in the code fragments, that means that code was omitted for brevity.

The Sprite class

Listing 2 shows the beginning of the **Sprite** class along with two new instance variables.

```
class Sprite {
  //...
  private Vector tailData =
                            new
Vector();
  private int wormLength;
```

Listing 2

Length of the sea worm

The **int** variable named **wormLength** is used to store the number of segments belonging to a particular sea worm instantiated from the **Sprite** class. We will see that the value for **wormLength** is obtained from a random number generator when the sprite is instantiated. Thus, the animation has sea worms of different lengths swimming about.

Data about the tail

The **Vector** object referred to by **tailData** is used to store the most recent coordinate and image data for the sprite in the form of references to **History** objects.

The number of elements stored in **tailData** is equal to the value of **wormLength**. When the **Sprite** object is asked to draw itself, it draws one segment for each element in **tailData**, using the location and image information stored in the **History** object referred to by that element in **tailData**.

The constructor

Listing 3 shows an abbreviated listing for the constructor for this version of the **Sprite** class. Most of the code has been removed for brevity because I discussed it in earlier lessons.

```
  public Sprite(Component component,
                Image[] image,
                Point position,
                Point motionVector){
    //...
    wormLength = Math.abs(
                    rand.nextInt() %
20);
    //...
  }//end constructor


Listing 3
```

The value of **wormLength** is set to a positive random number between 0 and 20.

The new head

The only thing that is really new in this program is some of the code in the **drawSpriteImage** method, which begins in Listing 4.

```
  public void drawSpriteImage(
                          Graphics
g){
    frame = Math.abs(
                  rand.nextInt() %
6);

Listing 4
```

The code in Listing 4 uses the absolute value of a random integer, modulo 6 to set the value for **frame**. This value is used later to select the image that will be used to represent the new head segment for the sea worm.

Color of the head

The code in Listing 5 adds a new **History** object to the collection of objects in the **tailData** object. The **History** object is populated with a reference to an **Image** object extracted from the array of references to **Image** objects where the random value of **frame** computed earlier is used as an index. This establishes the color of the new head of the sea worm.

```
    tailData.add(new History(
                          image[frame],
                       spaceOccupied.x,
                     spaceOccupied.y));
```

Listing 5

Getting an Iterator

The code in Listing 6 gets a standard **Iterator** object on the **Vector** object referred to by **tailData**. *(If you are unfamiliar with iterators, you might want to refer to my lessons on the Java Collections Framework on my website.)*

```
    Iterator iterator =
                   tailData.iterator();
```

Listing 6

Using the Iterator to draw the sea worm

The code in Listing 7 uses the iterator to traverse the **Vector** list and access each of the **History** objects *(in order from tail to head).*

As each **History** object is accessed, it is drawn on the screen using the **drawImage** method.

```
    int cnt = 0;
    while(iterator.hasNext()){
      History history =
              (History)iterator.next();
      if(tailData.size()>wormLength &&
           cnt == 0)
        iterator.remove();
      g.drawImage(history.image,
                  history.x,
                  history.y,
                  component);
      cnt++;
    }//end while
  }//end drawSpriteImage()
```

Listing 7

Here's looking at you

Because the element at the end of the list *(the newest element)* is drawn last, it appears to partially cover the drawing for the previous elements. This gives the impression that the sea worms are always facing out of the screen as they swim.

Discarding the tail segment

The *remove* capability of the iterator is used to remove the oldest element from the list during the traversal of the list.

Summary

That's a wrap for now on this series of lessons on animation. I may come back later and add some more lessons if I think of something else that would be interesting to discuss.

In the meantime, you now know how to use Java to write programs that produce smooth animation of the sprite animation and frame animation varieties. It's time for you to take your newfound knowledge and do something fun with it.

And don't forget, Java programming can be fun.

Complete Program Listing

A complete listing of the program is provided in Listing 8.

```
/*File Animate03.java
Copyright 2001, R.G.Baldwin
This program displays several
multicolored sea worms swimming around
in an aquarium.  Each seaworm maintains
generally the same course until it
collides with the head of another
sea worm or with a wall.  However, the
sea worms have the ability to change
course based on the addition or
subtraction of random values from the
components of their motion vector
about once in every ten updates.  The
opportunity to change course is also
random.  The length of each sea worm
may be different based on a random
number gererator.

A sea worm is constructed of segments
where each segment is a colored ball.
Each sea worm uses frame animation to
change the color of each segment.
Each segment switches among red,
green, blue, yellow, purple, and
orange on a random basis.

***********************************/
import java.awt.*;
import java.awt.event.*;
```

```
import java.util.*;

public class Animate03 extends Frame
                  implements Runnable{
  //This class is identical to
  // Animate01 with the exception of
  // the method named makeSprite
  private Image offScreenImage;
  private Image backGroundImage;
  private Image[] gifImages =
                          new Image[6];
  //offscreen graphics context
  private Graphics
                  offScreenGraphicsCtx;
  private Thread animationThread;
  private MediaTracker mediaTracker;
  private SpriteManager spriteManager;
  //Animation display rate, 12fps
  private int animationDelay = 83;
  private Random rand =
                new Random(System.
                  currentTimeMillis());

  public static void main(
                        String[] args){
    new Animate03();
  }//end main
  //---------------------------------//

  Animate03() {//constructor
    // Load and track the images
    mediaTracker =
                new MediaTracker(this);
    //Get and track the background
    // image
    backGroundImage =
        Toolkit.getDefaultToolkit().
          getImage("background02.gif");
    mediaTracker.addImage(
                   backGroundImage, 0);

    //Get and track 6 images to use
    // for sprites
    gifImages[0] =
           Toolkit.getDefaultToolkit().
               getImage("redball.gif");
    mediaTracker.addImage(
                      gifImages[0], 0);
    gifImages[1] =
           Toolkit.getDefaultToolkit().
             getImage("greenball.gif");
    mediaTracker.addImage(
                      gifImages[1], 0);
    gifImages[2] =
           Toolkit.getDefaultToolkit().
              getImage("blueball.gif");
```

```
    mediaTracker.addImage(
                      gifImages[2], 0);
    gifImages[3] =
           Toolkit.getDefaultToolkit().
            getImage("yellowball.gif");
    mediaTracker.addImage(
                      gifImages[3], 0);
    gifImages[4] =
           Toolkit.getDefaultToolkit().
            getImage("purpleball.gif");
    mediaTracker.addImage(
                      gifImages[4], 0);
    gifImages[5] =
           Toolkit.getDefaultToolkit().
            getImage("orangeball.gif");
    mediaTracker.addImage(
                      gifImages[5], 0);

    //Block and wait for all images to
    // be loaded
    try {
      mediaTracker.waitForID(0);
    }catch (InterruptedException e) {
      System.out.println(e);
    }//end catch

    //Base the Frame size on the size
    // of the background image.
    //These getter methods return -1 if
    // the size is not yet known.
    //Insets will be used later to
    // limit the graphics area to the
    // client area of the Frame.
    int width =
        backGroundImage.getWidth(this);
    int height =
       backGroundImage.getHeight(this);

    //While not likely, it may be
    // possible that the size isn't
    // known yet.  Do the following
    // just in case.
    //Wait until size is known
    while(width == -1 || height == -1){
      System.out.println(
                  "Waiting for image");
      width = backGroundImage.
                        getWidth(this);
      height = backGroundImage.
                       getHeight(this);
    }//end while loop

    //Display the frame
    setSize(width,height);
    setVisible(true);
    setTitle(
```

```
        "Copyright 2001, R.G.Baldwin");

    //Create and start animation thread
    animationThread = new Thread(this);
    animationThread.start();

    //Anonymous inner class window
    // listener to terminate the
    // program.
    this.addWindowListener(
                     new WindowAdapter(){
      public void windowClosing(
                          WindowEvent e){
        System.exit(0);}});

  }//end constructor
  //---------------------------------//

  public void run() {
    //Create and add sprites to the
    // sprite manager
    spriteManager = new SpriteManager(
             new BackgroundImage(
               this, backGroundImage));
    //Create 15 sprites from 6 gif
    // files.
    for (int cnt = 0; cnt < 15; cnt++){
      Point position = spriteManager.
        getEmptyPosition(new Dimension(
           gifImages[0].getWidth(this),
           gifImages[0].
                     getHeight(this)));
      spriteManager.addSprite(
        makeSprite(position, cnt % 6));
    }//end for loop

    //Loop, sleep, and update sprite
    // positions once each 83
    // milliseconds
    long time =
            System.currentTimeMillis();
    while (true) {//infinite loop
      spriteManager.update();
      repaint();
      try {
        time += animationDelay;
        Thread.sleep(Math.max(0,time -
          System.currentTimeMillis()));
      }catch (InterruptedException e) {
        System.out.println(e);
      }//end catch
    }//end while loop
  }//end run method
  //---------------------------------//

  private Sprite makeSprite(
```

```
       Point position, int imageIndex){
    return new Sprite(
        this,
        gifImages,
        position,
        new Point(rand.nextInt() % 5,
                  rand.nextInt() % 5));
  }//end makeSprite()
  //---------------------------------//

  //Overridden graphics update method
  // on the Frame
  public void update(Graphics g) {
    //Create the offscreen graphics
    // context
    if (offScreenGraphicsCtx == null) {
      offScreenImage =
         createImage(getSize().width,
                     getSize().height);
      offScreenGraphicsCtx =
          offScreenImage.getGraphics();
    }//end if

    // Draw the sprites offscreen
    spriteManager.drawScene(
                 offScreenGraphicsCtx);

    // Draw the scene onto the screen
    if(offScreenImage != null){
         g.drawImage(
           offScreenImage, 0, 0, this);
    }//end if
  }//end overridden update method
  //---------------------------------//

  //Overridden paint method on the
  // Frame
  public void paint(Graphics g) {
    //Nothing required here.  All
    // drawing is done in the update
    // method above.
  }//end overridden paint method

}//end class Animate01
//===================================//

class BackgroundImage{
  //This class is identical to that
  // used in Animate01
  private Image image;
  private Component component;
  private Dimension size;

  public BackgroundImage(
                  Component component,
                  Image image) {
```

```
    this.component = component;
    size = component.getSize();
    this.image = image;
  }//end construtor

  public Dimension getSize(){
    return size;
  }//end getSize()

  public Image getImage(){
    return image;
  }//end getImage()

  public void setImage(Image image){
    this.image = image;
  }//end setImage()

  public void drawBackgroundImage(
                        Graphics g) {
    g.drawImage(
               image, 0, 0, component);
  }//end drawBackgroundImage()
}//end class BackgroundImage
//===========================

class SpriteManager extends Vector {
  private BackgroundImage
                      backgroundImage;
  //This class is identical to that
  // used in Animate01

  public SpriteManager(
     BackgroundImage backgroundImage) {
    this.backgroundImage =
                      backgroundImage;
  }//end constructor
  //---------------------------------//

  public Point getEmptyPosition(
                 Dimension spriteSize){
    Rectangle trialSpaceOccupied =
      new Rectangle(0, 0,
                   spriteSize.width,
                   spriteSize.height);
    Random rand =
         new Random(
           System.currentTimeMillis());
    boolean empty = false;
    int numTries = 0;

    // Search for an empty position
    while (!empty && numTries++ < 100){
      // Get a trial position
      trialSpaceOccupied.x =
        Math.abs(rand.nextInt() %
                      backgroundImage.
```

```
                    getSize().width);
      trialSpaceOccupied.y =
        Math.abs(rand.nextInt() %
                     backgroundImage.
                     getSize().height);

      // Iterate through existing
      // sprites, checking if position
      // is empty
      boolean collision = false;
      for(int cnt = 0;cnt < size();
                                cnt++){
        Rectangle testSpaceOccupied =
              ((Sprite)elementAt(cnt)).
                    getSpaceOccupied();
        if (trialSpaceOccupied.
                 intersects(
                   testSpaceOccupied)){
          collision = true;
        }//end if
      }//end for loop
      empty = !collision;
    }//end while loop
    return new Point(
                 trialSpaceOccupied.x,
                 trialSpaceOccupied.y);
  }//end getEmptyPosition()
  //---------------------------------//

  public void update() {
    Sprite sprite;

    //Iterate through sprite list
    for (int cnt = 0;cnt < size();
                                cnt++){
      sprite = (Sprite)elementAt(cnt);
      //Update a sprite's position
      sprite.updatePosition();

      //Test for collision. Positive
      // result indicates a collision
      int hitIndex =
              testForCollision(sprite);
      if (hitIndex >= 0){
        //a collision has occurred
        bounceOffSprite(cnt,hitIndex);
      }//end if
    }//end for loop
  }//end update
  //---------------------------------//

  private int testForCollision(
                   Sprite testSprite) {
    //Check for collision with other
    // sprites
    Sprite  sprite;
```

```
  for (int cnt = 0;cnt < size();
                            cnt++){
    sprite = (Sprite)elementAt(cnt);
    if (sprite == testSprite)
      //don't check self
      continue;
    //Invoke testCollision method
    // of Sprite class to perform
    // the actual test.
    if (testSprite.testCollision(
                            sprite))
      //Return index of colliding
      // sprite
      return cnt;
  }//end for loop
  return -1;//No collision detected
}//end testForCollision()
//-------------------------------//

private void bounceOffSprite(
                  int oneHitIndex,
                  int otherHitIndex){
  //Swap motion vectors for
  // bounce algorithm
  Sprite oneSprite =
      (Sprite)elementAt(oneHitIndex);
  Sprite otherSprite =
    (Sprite)elementAt(otherHitIndex);
  Point swap =
         oneSprite.getMotionVector();
  oneSprite.setMotionVector(
      otherSprite.getMotionVector());
  otherSprite.setMotionVector(swap);
}//end bounceOffSprite()
//-------------------------------//

public void drawScene(Graphics g){
  //Draw the background and erase
  // sprites from graphics area
  //Disable the following statement
  // for an interesting effect.
  backgroundImage.
              drawBackgroundImage(g);

  //Iterate through sprites, drawing
  // each sprite
  for (int cnt = 0;cnt < size();
                             cnt++)
    ((Sprite)elementAt(cnt)).
                  drawSpriteImage(g);
}//end drawScene()
//-------------------------------//

public void addSprite(Sprite sprite){
  add(sprite);
}//end addSprite()
```

```
}//end class SpriteManager
//===================================//

class Sprite {
  private Component component;
  private Image[] image;
  private Rectangle spaceOccupied;
  private Point motionVector;
  private Rectangle bounds;
  private Random rand;
  private int frame;
  private Vector tailData =
                          new Vector();
  private int wormLength;

  public Sprite(Component component,
                Image[] image,
                Point position,
                Point motionVector){
    //Seed a random number generator
    // for this sprite with the sprite
    // position.
    rand = new Random(position.x);
    wormLength = Math.abs(
                  rand.nextInt() % 20);
    this.component = component;
    this.image = image;
    setSpaceOccupied(new Rectangle(
       position.x,
       position.y,
       image[0].getWidth(component),
       image[0].getHeight(component)));
    this.motionVector = motionVector;
    //Compute edges of usable graphics
    // area
    int topBanner = (
                 (Container)component).
                       getInsets().top;
    int bottomBorder = (
                 (Container)component).
                    getInsets().bottom;
    int leftBorder = (
                 (Container)component).
                      getInsets().left;
    int rightBorder = (
                 (Container)component).
                     getInsets().right;
    bounds = new Rectangle(
          0 + leftBorder,
          0 + topBanner,
          component.getSize().width -
            (leftBorder + rightBorder),
          component.getSize().height -
           (topBanner + bottomBorder));
  }//end constructor
```

```
//----------------------------------//

public Rectangle getSpaceOccupied(){
  return spaceOccupied;
}//end getSpaceOccupied()
//----------------------------------//

void setSpaceOccupied(
            Rectangle spaceOccupied){
  this.spaceOccupied = spaceOccupied;
}//setSpaceOccupied()
//----------------------------------//

public void setSpaceOccupied(
                     Point position){
  spaceOccupied.setLocation(
             position.x, position.y);
}//setSpaceOccupied()
//----------------------------------//

public Point getMotionVector(){
  return motionVector;
}//end getMotionVector()
//----------------------------------//

public void setMotionVector(
                 Point motionVector){
  this.motionVector = motionVector;
}//end setMotionVector()
//----------------------------------//

public void setBounds(
                   Rectangle bounds){
  this.bounds = bounds;
}//end setBounds()
//----------------------------------//
public void updatePosition() {
  Point position = new Point(
   spaceOccupied.x, spaceOccupied.y);

  //Insert random behavior.  During
  // each update, a sprite has about
  // one chance in 10 of making a
  // small random change to its
  // motionVector.  When a change
  // occurs, the motionVector

  // coordinate values are forced to
  // fall between -7 and 7.
  if(rand.nextInt() % 10 == 0){
    Point randomOffset =
       new Point(rand.nextInt() % 3,
                 rand.nextInt() % 3);
    motionVector.x += randomOffset.x;
    if(motionVector.x >= 7)
      motionVector.x -= 7;
```

```
      if(motionVector.x <= -7)
        motionVector.x += 7;
      motionVector.y += randomOffset.y;
      if(motionVector.y >= 7)
        motionVector.y -= 7;
      if(motionVector.y <= -7)
        motionVector.y += 7;
    }//end if

    //Make the move
    position.translate(
       motionVector.x, motionVector.y);

    //Bounce off the walls
    boolean bounceRequired = false;
    Point tempMotionVector =
             new Point(motionVector.x,
                       motionVector.y);

    //Handle walls in x-dimension
    if (position.x < bounds.x) {
      bounceRequired = true;
      position.x = bounds.x;
      //reverse direction in x
      tempMotionVector.x =
                   -tempMotionVector.x;
    }else if((position.x +
                 spaceOccupied.width) >
            (bounds.x + bounds.width)){
      bounceRequired = true;
      position.x = bounds.x +
                 bounds.width -
                   spaceOccupied.width;
      //reverse direction
      tempMotionVector.x =
                   -tempMotionVector.x;
    }//end else if

    //Handle walls in y-dimension
    if (position.y < bounds.y){
      bounceRequired = true;
      position.y = bounds.y;
      tempMotionVector.y =
                   -tempMotionVector.y;
    }else if ((position.y +
                spaceOccupied.height) >
      (bounds.y + bounds.height)){
      bounceRequired = true;
      position.y =
              bounds.y +
                bounds.height -
                  spaceOccupied.height;
      tempMotionVector.y =
                   -tempMotionVector.y;
    }//end else if
```

```
    //save new motionVector
    if (bounceRequired)
     setMotionVector(tempMotionVector);

    //update spaceOccupied
    setSpaceOccupied(position);
  }//end updatePosition()
  //---------------------------------//

  public void drawSpriteImage(
                          Graphics g){
    frame = Math.abs(
                   rand.nextInt() % 6);
    tailData.add(new History(
                         image[frame],
                      spaceOccupied.x,
                    spaceOccupied.y));
    Iterator iterator =
                   tailData.iterator();
    int cnt = 0;
    while(iterator.hasNext()){
      History history =
              (History)iterator.next();
      if(tailData.size()>wormLength &&
           cnt == 0)
        iterator.remove();
      g.drawImage(history.image,
                  history.x,
                  history.y,
                  component);
      cnt++;
    }//end while
  }//end drawSpriteImage()

  public boolean testCollision(
                    Sprite testSprite){
    // Check for collision with another
    // sprite
    if (testSprite != this){
      return spaceOccupied.intersects(
        testSprite.getSpaceOccupied());
    }//end if
    return false;
  }//end testCollision
}//end Sprite class
//=====================================//

class History{
  Image image;
  int x;
  int y;

  History(Image image, int x, int y){
    this.image = image;
    this.x = x;
    this.y = y;
```

```
  }//end constructor
}//end class History

Listing 8
```

Copyright 2001, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java and XML. In addition to the many platform-independent benefits of Java applications, he believes that a combination of Java and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects involving Java, XML, or a combination of the two. He frequently provides onsite Java and/or XML training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Java Programming Tutorials, which has gained a worldwide following among experienced and aspiring Java programmers. He has also published articles on Java Programming in Java Pro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin.richard@iname.com

-end-