Fun with Java: Sprite Animation, Part 5

Baldwin shows you how to override the update method of the Component class to improve the animation quality of the program over what would normally be achieved using the default version of the update method. In the process, he shows you how to eliminate the flashing that often accompanies efforts to use the default version of the update method for animation purposes. He also shows you how to get and use an offscreen drawing context to accomplish double buffering in the drawing process.

Published: November 4, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1458

Preface

Why the repeated introduction?

If you are one of those orderly people who start reading a book at the beginning and reads through to the end, you are probably wondering why I keep repeating this long introduction. The truth is that this introduction isn't meant for you. Rather, it is meant for those people who start reading in the middle.

Having said that, 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 fifth in of a group of lessons that teach you how to write animation programs in Java.

The first lesson in the group was entitled Fun with Java: Sprite Animation, Part 1. *(Here is your opportunity to go back and start reading at the beginning.)* The previous lesson was entitled Fun with Java: Sprite Animation, Part 4.

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.

Animated spherical sea creatures

The first program, being discussed in this lesson, will show you how to use sprite animation to cause a group of colored spherical sea creatures to swim around in a fish tank. A screen shot of the output produced by this program is shown in Figure 1.

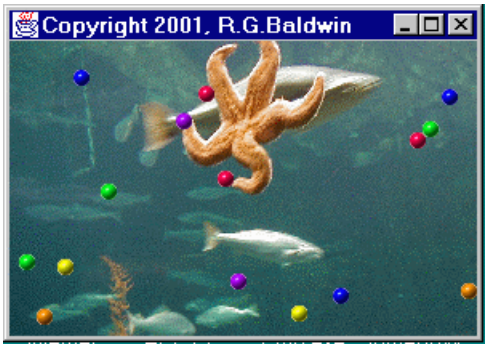

Figure 1. Animated spherical sea creatures in a fish tank.

A creature of many colors

Many sea creatures have the ability to change their color in very impressive ways. The second program that I will discuss in subsequent lessons will simulate that process using a combination of sprite and frame animation.

Slithering sea worms

The third program, also to be discussed in a subsequent lesson, will use a combination of sprite animation, frame animation, and 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 the third program is shown in Figure 2.

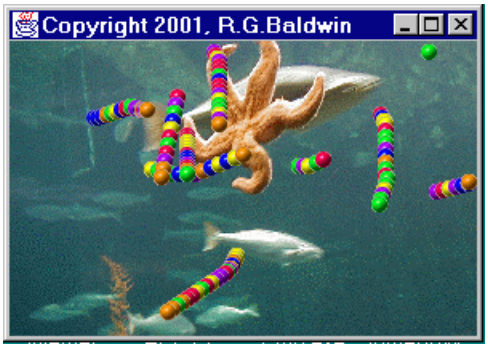

Figure 2. Animated sea worms in a fish tank.

Getting the required GIF images

Figure 3 shows the GIF image files that you will need to run these three programs.

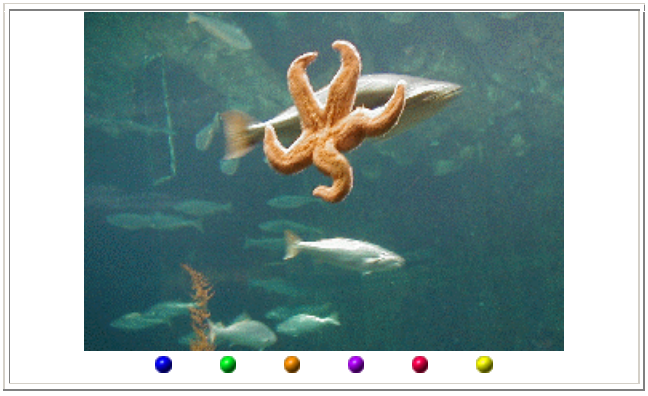

Figure 3. GIF image files that you will need.

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 programs.

Review of previous lesson

In the previous lesson, I explained the behavior of the **run** method of the animation thread as well as the **makeSprite** method of the controlling class.

I provided a preview of the **SpriteManager** class, which will be discussed in detail in a subsequent lesson. I also provided a brief preview of the **Sprite** class, which will be discussed in detail in a subsequent lesson.

I discussed the **repaint**, **update**, and **paint** methods of the **Component** class. I also discussed the timer loop used in this program, and suggested an alternative approach that makes use of a **Timer** object to fire **Action** events.

Also in the previous lesson, I provided a complete recap of everything that we have learned in the first four lessons of this series.

What are the plans for this lesson?

There are only two methods remaining to be discussed in the controlling class: **update** and **paint**. In this lesson, I will explain the behavior of the overridden **update** and **paint** methods. As explained in the previous lesson, the **update** method is invoked by the operating system in response to a **repaint** request on the **Frame**.

Discussion and Sample Program

This program is so long that several lessons will be required to discuss it fully. Rather than to make you wait until I complete all of those lessons to get your hands on the program, I have provided a copy of the entire program in Listing 6 near the end of the lesson. That way, you can copy it into a source file on your local disk, compile it, run it, and start seeing the results.

Discuss in fragments

As usual, I will discuss the program in fragments. As explained in the previous lesson, once during each iteration of the animation loop, the code updates the positions of all of the sprites and then invokes the **repaint** method on the **Frame**. Then it sleeps for a specified period, wakes up, and does it all over again.

What is the behavior of the repaint method?

According to the Sun documentation, invoking the **repaint** method on the **Frame** causes a call to be made to the frame's **update** method as soon as possible.

What is the behavior of the update method?

Sun describes the default behavior of the **update** method as follows:

> *"The update method of Component does the following:*
>
> - *Clears this component by filling it with the background color.*
> - *Sets the color of the graphics context to be the foreground color of this component.*

- *Calls this component's paint method to completely redraw this component."*

Doesn't use default update method

However, this animation program does not use the default behavior of the **update** method. Rather, this program overrides the **update** method to provide behavior that is appropriate for better-quality animation.

Three major changes

This program makes three major changes to the default behavior of the **update** method to improve the animation quality.

Eliminate flashing

First, the new behavior eliminates the clearing of the display area of the **Frame** object at the beginning of each redraw operation. This eliminates a *flashing* effect that is often produced by that default operation.

Use double buffering

Second, the new behavior draws the scene on an offscreen graphics context and then transfers that scene onto the screen context at high speed. *(This is often referred to as double buffering.)* This makes it impossible for the viewer to see the scene as it is being drawn, and usually provides a more pleasing result.

No need for the paint method

Third, since all of the required drawing is accomplished in the **update** method, there is no call to the **paint** method by the **update** method. Although an overridden version of the **paint** method is provided, it doesn't do anything, and it isn't invoked by the animation process.

An offscreen graphics context

The code fragment in Listing 1 gets an offscreen graphics context to be used as described above.

```
  public void update(Graphics g) {
    if (offScreenGraphicsCtx == null) {
      offScreenImage =
         createImage(getSize().width,
                     getSize().height);
      offScreenGraphicsCtx =
          offScreenImage.getGraphics();
    }//end if

Listing 1
```

As you can see, there are two steps involved in getting a useful offscreen graphics context.

1. Create an Image object
2. Create a Graphics object

Create an Image object

The first step is to invoke the **createImage** method of the **Component** class to get an **Image** object of a specified size. This method requires the width and height of the image as parameters and returns a reference to an object of type **Image**.

Create a Graphics object

The second step is to invoke the **getGraphics** method on the **Image** object returned by the first step. This method returns a reference to an object of type **Graphics**. Here is what Sun has to say about the method:

> *"Creates a graphics context for drawing to an off-screen image. This method can only be called for off-screen images."*

Drawing offscreen

Once we have the offscreen graphics context, we can invoke any of the methods of the **Graphics** class to draw pictures on that context.

In this case, we don't draw the pictures within the **update** method directly. Rather, we pass a reference to the offscreen context to the **drawScene** method of the **SpriteManager** object where the drawing is actually accomplished *(you will see how the scene is drawn in a subsequent lesson where I discuss the SpriteManager class in detail).*

The call to the **drawScene** method of the **SpriteManager** class is shown in Listing 2.

```
    spriteManager.drawScene(
                  offScreenGraphicsCtx);
```

Listing 2

Drawing onscreen

When the **drawScene** method returns, the scene has been drawn onto the offscreen graphics context, and it is time to copy it to the screen context at a high rate of speed. This is accomplished using the **drawImage** method of the **Graphics** class, as shown in Listing 3.

```
    if(offScreenImage != null){
```

```
          g.drawImage(
            offScreenImage, 0, 0,
this);
    }//end if
  }//end overridden update method

Listing 3
```

The drawImage method is overloaded

There are several overloaded versions of the **drawImage** method. The version used here requires four parameters. The first parameter is a reference to an offscreen graphics context.

The second and third parameters are the coordinate values of the screen image where the top left-hand corner of the offscreen image will be positioned. In this case, the top left-hand corner of the offscreen image will be placed at the top left-hand corner of the screen image.

The last parameter is a reference to an **ImageObserver** object. I have probably already confused you enough in an earlier lesson regarding the use of **this** as an **ImageObserver** object. I won't add to that confusion by discussing it further here. *(Again, however, I plan to dedicate an entire future lesson to the concept of ImageObserver.)*

Does not invoke the paint method

Note that the code in the overridden **update** method does not invoke the **paint** method, as is the case with the default version of the **update** method. All of the required drawing is handled in the **update** method, and there is nothing further to be drawn by the **paint** method. As a result, the overridden **paint** method shown in Listing 4 below is not used in the animation process.

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

}//end class Animate01

Listing 4
```

That completes my discussion of all the methods of the controlling class.

The BackgroundImage class

Listing 5 contains all of the code for a utility class named **BackgroundImage**. As I mentioned in an earlier lesson, this class was written simply to make it a little easier to deal with the background image.

```
class BackgroundImage{
  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

Listing 5
```

A very simple class

As you can see from Listing 5, there isn't much to this class, so it doesn't deserve much in the way of a discussion.

Basically, an object instantiated from this class has the ability to

- Store a reference to a background **Image** object,
- Return a reference to the object,
- Return the size of the object
- Cause the background image to be drawn on a specified graphics context

Beyond that, there isn't much to be said for this class.

Summary

It has taken five lessons, but I have finally completed my discussion of the controlling class for this animation program.

In this lesson, I showed you how to override the **update** method of the **Component** class to improve the animation quality of the program over what would normally be achieved using the default version of the **update** method.

In the process, I showed you how to eliminate the flashing that often accompanies efforts to use the default version of the **update** method for animation purposes. This flashing is caused by the fact that the default version of **update** draws an empty component *(often white)* at the beginning of each redraw cycle.

I also showed you how to get and use an offscreen drawing context to accomplish *double buffering* in the drawing process. The use of double buffering makes it impossible for the user to see the scene as it is being drawn because the scene is first drawn offscreen and then transferred as a whole to the screen context. Depending on the drawing speed, this can also produce a more pleasing result.

I also provided a very brief discussion of the utility class named **BackgroundImage**.

What's Next?

There are two more classes to cover before my discussion of this animation program is complete: **SpriteManager** and **Sprite**.

I will begin my discussion of the **SpriteManager** class in the next lesson.

Complete Program Listing

A complete listing of the program is provided in Listing 6.

```
/*File Animate01.java
Copyright 2001, R.G.Baldwin

This program displays several animated
colored spherical creatures swimming
around in an aquarium.  Each creature
maintains generally the same course
with until it collides with another
creature or with a wall.  However,
each creature has the ability to
occasionally make random changes in
its course.

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

```
public class Animate01 extends Frame
                  implements Runnable {
  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 Animate01();
  }//end main
  //---------------------------------//

  Animate01() {//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[imageIndex],
```

```
          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{
  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;

  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;

  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);
    this.component = component;
    this.image = image;
    setSpaceOccupied(new Rectangle(
          position.x,
          position.y,
          image.getWidth(component),
          image.getHeight(component)));
    this.motionVector = motionVector;
    //Compute edges of usable graphics
    // area in the Frame.
    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
  // random change to its
  // motionVector.  When a change
  // occurs, the motionVector
  // coordinate values are forced to
  // fall between -7 and 7.  This
  // puts a cap on the maximum speed
  // for a sprite.
  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

  //Move the sprite on the screen
```

```
    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 in x
      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

    if(bounceRequired)
      //save new motionVector
                    setMotionVector(
                      tempMotionVector);
    //update spaceOccupied
    setSpaceOccupied(position);
  }//end updatePosition()
  //--------------------------------//

  public void drawSpriteImage(
```

```
                            Graphics g){
    g.drawImage(image,
                spaceOccupied.x,
                spaceOccupied.y,
                component);
  }//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
//=================================//
```

Listing 6

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-