Fun with Java: Sprite Animation, Part 6

In his continuing series on sprite animation, Baldwin explains the behavior of each of the methods of his SpriteManager class. This includes the following features: finding an empty location for each new sprite, updating sprite positions, drawing the new scene during each animation cycle, and managing collisions between sprites.

Published: November 11, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1460

Preface

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.

That said, 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 sixth 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. *(Here is your opportunity to go back and start reading at the beginning.)* The previous lesson was entitled Fun with Java: Sprite Animation, Part 5.

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

Animation programming

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.

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.

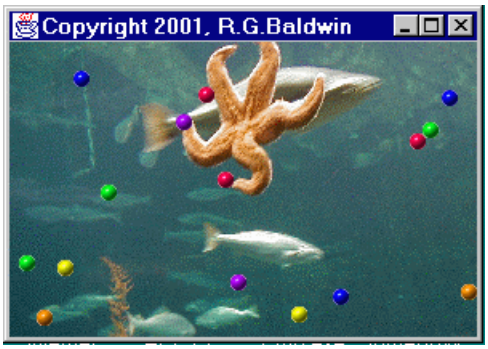

Figure 1. Animated spherical sea creatures in a fish tank.

The ability to change color

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.

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.

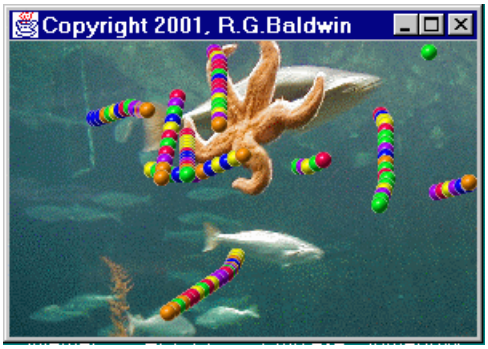

Figure 2. Animated sea worms in a fish tank.

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

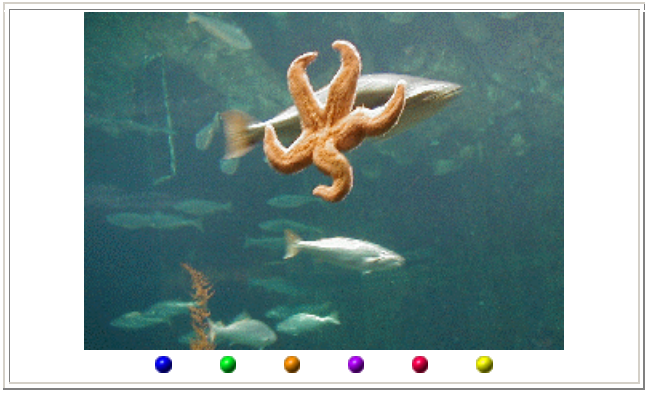

Figure 3. GIF image files that you will need.

Getting the required GIF files

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 completed my discussion of the controlling class for this animation program.

The update method

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.

Offscreen drawing context

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**.

Preview of this lesson

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

In this lesson, I will explain the behavior of each of the methods of the **SpriteManager** class. This will include the following features:

- Finding an empty space for each new sprite
- Updating sprite positions
- Drawing the new scene during each animation cycle
- Managing collisions between sprites

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 11 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.

The SpriteManager class

As the name implies, an object of the **SpriteManager** class can be used to manage a collection of sprites. Before I get into the details, here is a preview of some of the attributes of the **SpriteManager** class.

SpriteManager constructor

The constructor for the **SpriteManager** class requires an incoming parameter of type **BackgroundImage**. The **BackgroundImage** class is a convenience class designed to facilitate certain operations involving the background image displayed on the **Frame**. The code in various

methods of the **SpriteManager** class uses the **BackgroundImage** object to manage the background image against which the animation process is played out. *(You can view the background image by scrolling back up to Figure 3.)*

A collection of sprites in a Vector object

An object of the **SpriteManager** class stores references to a collection of sprites in an object of type **Vector**. A public method named **addSprite** can be invoked to cause a new sprite to be added to the collection.

Finding a parking place for a sprite

One of the public methods of the **SpriteManager** class is a method named **getEmptyPosition**. This method attempts to identify a location within the **Frame** that does not currently contain a sprite. This makes it possible to create a population of sprites without having them initially occupying the same physical space.

Updating the sprite positions

Another public method of the **SpriteManager** class is a method named **upDate** *(not to be confused with the update method of the Component class).* When this method is invoked, the **SpriteManager** object causes all of the sprites in its collection to change their position according to values stored in a *motion vector* owned by each sprite.

When the sprites change their positions, collisions can and do occur. Such collisions are handled by the **SpriteManager** using private methods named **testForCollision** and **bounceOffSprite**.

Drawing the scene

Another public method of the **SpriteManager** class is named **drawScene**. When this method is invoked, a new background image is drawn on the **Frame**. This has the effect of erasing all of the sprites from the scene. The method then causes each of the sprites to be drawn in their respective positions.

Discuss in fragments

As usual, I will discuss the program in fragments.

As shown in Listing 1, the **SpriteManager** class extends the **Vector** class. Hence, an object of the **SpriteManager** class is a **Collection** object according to the Java Collections Framework. *(If you are unfamiliar with that framework, I have published several lessons on the framework on my web site.)*

```
class SpriteManager extends Vector{
  private BackgroundImage
```

```
                        backgroundImage;

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

Listing 1
```

The background

The **SpriteManager** class starts off in a very mundane way. The constructor, shown in Listing 1, receives and saves a reference to an object of the **BackgroundImage** class.

Adding sprites

Continuing to keep things simple, the code in Listing 2 is used to add a new sprite to the collection. This code simply provides a wrapper method with a descriptive name around the standard **add** method of the **Vector** class.

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

Listing 2
```

There are overloaded versions of the **add** method in the **Vector** class. In keeping with the stipulations of the Java Collections Framework, the **add** method used here appends the incoming **Sprite** object's reference to the end of the collection of references stored in the **Vector**.

The getEmptyPosition method

The purpose of the method named **getEmptyPosition** is to try to find a rectangle the size of a sprite in a location that is not already occupied by one of the sprites in the collection.

This method, as shown in Listing 3, is long and ugly, but not conceptually difficult. Therefore, I will provide a brief summary but I won't discuss it in detail.

```
  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()
```

Listing 3

Basically the method uses a **for** loop inside a **while** loop to search for a rectangular area not already occupied by a sprite.

Check up to 100 locations

The **while** loop will iterate up to 100 times looking for an empty area. When it finds an empty area, the **while** loop terminates and the method returns the location of that area to the calling program. If the method fails to find an empty area in 100 tries, it gives up and returns a location that is already occupied by another sprite.

Check random locations

Inside the **while** loop, the code uses a random number generator to specify a trial location. It then uses a **for** loop, in conjunction with the **intersects** method of the **Rectangle** class, to test the trial location against the locations occupied by all of the sprites in the collection.

An empty location

If the rectangular area at the trial location doesn't intersect a rectangular area occupied by any existing sprite, that rectangular area and that location is deemed to be empty.

An occupied location

If the rectangular area at the trial location does intersect a rectangular area occupied by an existing sprite, that location is discarded. In that case, another trial location area is specified using the random number generator and the test is performed again.

Redraw and sleep

As I explained in an earlier lesson, once control enters the animation loop, the **update** method is invoked on the **SpriteManager** object. When the **update** method returns, the code invokes the **repaint** method on the **Frame** and the animation thread goes to sleep for a specified period of time.

When the animation thread wakes up, the process is repeated.

(Be careful not to confuse this method named update with the method having the same name that is defined in the Component class, and overridden in the controlling class for this program.)

The update method

The update method causes all of the sprites being managed by the **SpriteManager** object to assume new positions based on the motion rules defined by the **Sprite** class.

The beginning of the **update** method is shown in Listing 4.

```
  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();

Listing 4
```

Once all of the sprites have assumed new positions, it is possible that some collisions among sprites may have occurred. The **update** method tests for collisions by examining the location information belonging to each of the **Sprite** objects being managed.

Managing collisions

If a collision has occurred, a method named **bounceOffSprite** is invoked to cause the two sprites to behave in a manner similar to that, which occurs between two balls in a game of pool. *(I will discuss the bounceOffSprite method later.)*

Iterate on all sprites

All of the executable code in the **update** method exists inside a **for** loop that iterates through all of the sprites in the collection.

The code inside the **for** loop begins by selecting the next sprite in the collection and causing that sprite to assume a new position by invoking the **updatePosition** method on the sprite. *(We will see exactly how a sprite moves in a subsequent lesson where I discuss the Sprite class in detail.)*

Test for a collision

The code in the **for** loop then invokes a private method of the **SpriteManager** class named **testForCollision** to determine if the sprite that just moved collided with any other sprite. *(I will discuss the testForCollision method shortly.)*

This code is shown in Listing 5.

```
      int hitIndex =
              testForCollision(sprite)

Listing 5
```

Negative return value is OK

If a collision did not occur, the **testForCollision** method returns a negative value. This causes another iteration of the **for** loop to test the next sprite in the collection for a collision.

Who hit me?

If a collision did occur, the **testForCollision** method returns either zero or a positive value. This value is the index in the collection of the *other party to the collision*.

Bounce when collision occurs

In this case, the code shown in Listing 6 passes references to both parties in the collision to the method named **bounceOffSprite** to cause the sprites to bounce off of one another. *(You will see how that is done shortly when I discuss the method named bounceOffSprite.)*

```
      if (hitIndex >= 0){
        //a collision has occurred
        bounceOffSprite(cnt,hitIndex);
      }//end if
    }//end for loop
  }//end update

Listing 6
```

The testCollision method

The actual code that tests to determine if two sprites have collided is performed by a method named **testCollision**, which is a method of the **Sprite** class. Breaking this code out into a separate method makes it easier to customize that code to be appropriate for sprites of new designs.

The testForCollision method

The method named **testForCollision** defined in the **SpriteManager** class invokes the **testCollision** method to determine if two specific sprites have collided.

The **testForCollision** method, shown in Listing 7, receives a reference to a **Sprite** object as a parameter. It uses a **for** loop to test that sprite against all of the sprites in the collection except itself.

```
  private int testForCollision(
                  Sprite testSprite){
    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()

Listing 7
```

True for collision, false otherwise

During each iteration of the loop, it passes a reference to the sprite being tested and a reference to one of the sprites in the collection to the **testCollision** method of the **Sprite** class. If that method determines that a collision has occurred, it returns true. Otherwise, it returns false.

Terminate on collision

If the **testCollision** method returns true, the **testForCollision** method terminates, returning the positive index of the sprite in the collection that collided with the sprite being tested.

If no collisions are detected, the **testForCollision** method returns -1.

Small incremental steps

Although it isn't apparent while watching the animation run, each sprite moves through the water in small incremental steps. Each sprite takes one step in the horizontal direction and one step in the vertical direction during each iteration of the animation loop in the **run** method of the controlling class.

The motion vector

As you will see when I discuss the **Sprite** class in the next lesson, each **Sprite** object has an instance variable of type **Point** named **motionVector** *(the word vector here is used to indicate a mathematical vector, and is not to be confused with the Java class named Vector).*

Each motion vector contains a signed integer describing the incremental horizontal motion of the sprite and a signed integer describing the incremental vertical motion of the sprite. Thus, the contents of the **Point** object referred to by **motionVector** determine the speed and direction of motion for the sprite. *(We will also see that these two values can occasionally change on a random basis causing a sprite to change its speed and direction of motion.)*

The bounceOffSprite method

The method named **bounceOffSprite** shown in Listing 8 receives references to two **Sprite** objects. This method is called by the **SpriteManager** object when two sprites are determined to have collided.

Swap motion vectors

The concept is simple. If two sprites, *(call them A and B)*, collide, they swap motion vectors. This causes sprite B to assume the speed and direction previously exhibited by sprite A, and causes sprite A to assume the speed and direction previously exhibited by sprite B.

Thus, the code shown in Listing 8 simply swaps the motion vectors between the two sprites received as incoming parameters.

```
  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()
```

Listing 8

An unplanned behavior

However, if you watch the animation very carefully, you may witness an interesting and totally unplanned behavior. Sometimes, two sprites will travel side-by-side for a while and then separate.

Why does this happen?

I haven't performed a detailed analysis to determine what causes this behavior, but I suspect that it results from two sprites colliding at a very shallow angle. I suspect that simply swapping motion vectors doesn't actually separate them, and they are deemed to have collided again during the next animation update. If so, this causes them to swap motion vectors again, and the process continues. Eventually, one or both of the sprites makes a random change in speed and direction, at which point they separate.

Since these are animated sea creatures, I have chalked this behavior up to the natural mating tendencies of sea creatures.

The drawScene method

In this program, the **drawScene** method is invoked by code in the overridden **update** method of the controlling class. When the **update** method is invoked, it is passed a reference to an offscreen graphics context.

An offscreen graphics context

The **drawScene** method receives a reference to an object of type **Graphics** as an incoming parameter. This **Graphics** object represents a graphics context. While it could represent any graphics context, in this program it represents the offscreen graphics context mentioned above.

Drawing pictures

The code in the **drawScene** method can draw pictures on the graphics context represented by the incoming **Graphics** object by invoking the methods of the **Graphics** class on that object.

Listing 9 shows the beginning of the method named **drawScene**.

```
  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);

Listing 9
```

The drawBackgroundImage method

The code in Listing 9 invokes the **drawBackgroundImage** method on the **BackgroundImage** object, passing the graphics context as a parameter.

If you refer back to the discussion of the **BackgroundImage** class in the previous lesson, you will see that the code in the **drawBackgroundImage** method invokes the **drawImage** method to draw the background image on the graphics context that it receives an incoming parameter.

Covers all existing sprites

Because the size of the background image completely fills the graphics context, this has the effect of drawing the background over all of the sprites that were previously drawn there.

An interesting effect

(You can disable the statement in Listing 9 to see an interesting effect that may help you to understand what is going on in the drawing process.

Disabling this statement prevents the previously-drawn sprites from being covered up by the background image, and the sprites simply accumulate in the drawing area. They end up looking something like colored tube worms.)

Now draw the sprites

Once the new background image has been drawn on the graphics context, it is ready for the drawing of the individual sprites in their new locations. This is accomplished by the code in Listing 10.

```
    for (int cnt = 0;cnt < size();
                                  cnt++)
      ((Sprite)elementAt(cnt)).
                      drawSpriteImage(g);
  }//end drawScene()

Listing 10
```

The drawSpriteImage method

The code in Listing 10 iterates through all of the sprites in the collection, invoking the **drawSpriteImage** method on each sprite and passing a reference to the graphics context as a parameter. This causes each of the individual sprites to be drawn in their new locations. *(I will discuss the drawSpriteImage method of the Sprite class in the next lesson.)*

Collisions with the walls

The **drawSpriteImage** method of the **Sprite** class deals with the problem of sprites running into the walls. When a sprite runs into one of the walls, it bounces off the wall.

Summary

In this lesson, I have explained the behavior of each of the methods of the **SpriteManager** class. This included the following features:

- Finding an empty space for each new sprite
- Updating sprite positions
- Drawing the new scene during each animation cycle
- Managing collisions between sprites

What's Next?

There is one more class to cover before my discussion of this animation program is complete: **Sprite**.

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

Complete Program Listing

A complete listing of the program is provided in Listing 11.

```
/*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 11
```

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-