Fun with Java: Sprite Animation, Part 4

Baldwin explains the behavior of the run method of the animation thread as well as the makeSprite method of the controlling class. He provides a preview of the SpriteManager class and the Sprite class. He discusses the repaint, update, and paint methods of the Component class, and discusses the timer loop used in this program. He also suggests an alternative approach that makes use of a Timer object to fire Action events. Finally, he recaps the previous three lessons in the series.

Published: October 22, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1456

Preface

This is one of the lessons in a miniseries that concentrates 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 fourth 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. The previous lesson was entitled Fun with Java: Sprite Animation, Part 3.

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.

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.

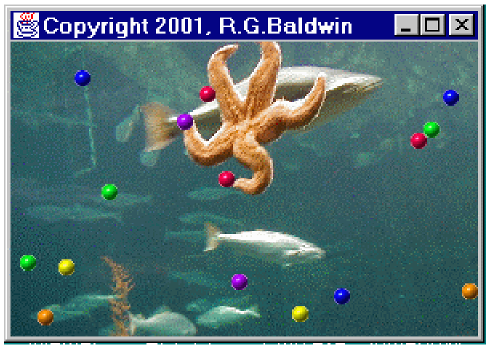

Figure 1. Animated spherical sea creatures in a fish tank.

Changing color with frame animation

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.

Animated 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.

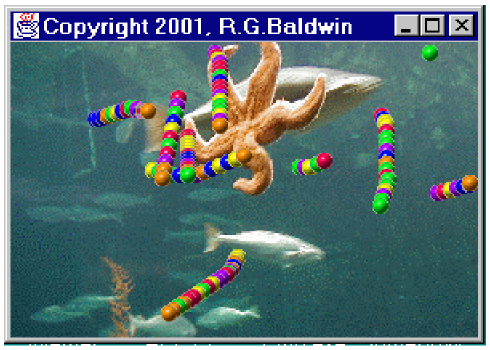

Figure 2. Animated sea worms in a fish tank.

Getting the GIF files

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

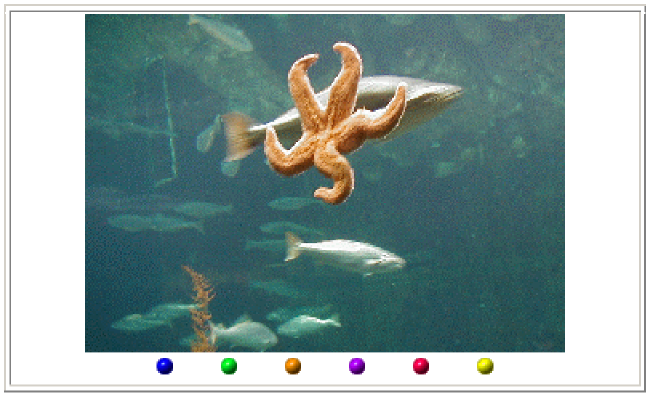

Figure 3. GIF image files that you will need.

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 discussed the unusual nature of the *getter* methods for the **width** and **height** properties of an **Image** object.

I introduced and briefly discussed the concept of an **ImageObserver** object in conjunction with the **getWidth** and **getHeight** methods of an **Image** object.

I showed you how to set the size of the **Frame** to be the same as the size of the background image.

I discussed the use of an object of the controlling class as an animation thread.

Also, in the previous lesson, I completed my discussion of the constructor for the controlling class.

What's in this lesson?

In this lesson, I will explain the behavior of the **run** method of the animation thread as well as the **makeSprite** method of the controlling class.

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

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

Finally, I will summarize everything that we have learned so far in this and the previous three lessons.

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. In the previous lesson, I completed my discussion of the constructor for the controlling class and promised to explain the **run** method of the controlling class in this lesson.

The run method

The **run** method sets up the animation scenario and then goes into an infinite loop, updating the animation process approximately twelve times per second.

The code in Listing 1 shows the beginning of the **run** method and the instantiation of a new object of the class **SpriteManager**.

```
  public void run() {
    spriteManager = new SpriteManager(
                 new BackgroundImage(
                   this,
backGroundImage));
```

```
Listing 1
```

The SpriteManager class

As the name implies, an object of the **SpriteManager** class can be used to manage a collection of sprites. This class will be discussed in detail later. For the time being, here are 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**.

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.

Creating the collection of sprites

The code in Listing 2 shows the beginning of a **for** loop that creates fifteen individual sprites and stores references to those sprites in the collection managed by the **SpriteManager** object.

Six **Image** objects were created earlier and stored in an array of type **Image[]** by the constructor. These **Image** objects are used to provide the visual manifestations of the sprites. *(Unfortunately, this code may be a little difficult to follow due to the squeezing required by this narrow publication format.)*

```
    for (int cnt = 0; cnt < 15; cnt++){
      Point position = spriteManager.
        getEmptyPosition(new Dimension(
           gifImages[0].getWidth(this),
           gifImages[0].
                      getHeight(this)));

Listing 2
```

Getting the size of a sprite

The code in Listing 2 assumes that all of the images used to create sprites are the same size, *(which they are in this program).* In order to get a representative size for a sprite, this code applies the **getWidth** and **getHeight** methods to the **Image** object referred to by the reference stored in element 0 of the array of **Image** objects.

Finding an empty parking place for a sprite

The resulting width and height values are used to populate a **Dimension** object, which is passed to the **getEmptyPosition** method of the **SpriteManager** object. As explained earlier, this method locates a position not currently occupied by a sprite and returns the coordinates of that position as a reference to an object of type **Point**.

The makeSprite method

The controlling class also contains a method named **makeSprite**, which I will discuss in more detail later. For the time being, suffice it to say that this method is used to create and return an object of the **Sprite** class. *(I also haven't discussed the Sprite class yet, but will discuss it in a subsequent lesson.)*

Among other things, the constructor for the **Sprite** class requires a reference to an **Image** object and a reference to a **Point** object. The new **Sprite** object represents itself visually using the **Image**. The initial position of the new **Sprite** object is determined by the contents of a **Point** object.

Creating a new Sprite object

The code in Listing 3 *(still inside the for loop)* passes the **Point** object obtained from the **getEmptyPosition** method above, along with an integer value between 0 and 6 to the **makeSprite** method. The **makeSprite** method uses that integer to identify an element in the array of **Images**, and passes the **Point** and the **Image** to the constructor for the **Sprite** class *(along with some other required information).*

The **makeSprite** method returns a reference to a new **Sprite** object, which is added to the collection of **Sprite** objects being managed by the **SpriteManager** object.

```
      spriteManager.addSprite(
        makeSprite(position, cnt % 
6));
    }//end for loop

Listing 3
```

The SpriteManager is populated

The result of the **for** loop that ends in Listing 3 is a collection of 15 sprites being managed by the **SpriteManager** object. Because some of the sprites share the same **Image** objects for their visual manifestation, some of the spherical sea creatures in Figure 1 look the same.

Which way should I go?

In addition to an initial position and **Image**, each of the **Sprite** objects contains a two-dimensional *motion vector*, which indicates the direction and speed used by the sprite when it changes its location.

The components of the initial motion vector for each sprite are created using a random number generator by the **makeSprite** method. As we will see when we examine the **Sprite** class in detail, the motion vector for each sprite can be modified later, also based on a random number generator.

It's time to party

At this point, the stage is set. The background is in place. Each of the fifteen sprites has been positioned and has been given a motion vector. The time has come to start the animation process running.

The animation loop

The code in Listing 4 was taken from the book entitled Teach Yourself Internet Game Programming with Java in 21 Days, by Michael Morrison.

(For those systems where animation timing is really critical, a newer, and possibly better approach uses a Timer object that can be set to fire an Action event at predetermined

intervals. This approach is described in The JFC Swing Tutorial, A Guide to Constructing GUIs, by Walrath and Campione.)

Update the display

The code in Listing 4 attempts to cause the display to update itself once each 83 milliseconds, or about twelve times per second.

```
    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

Listing 4
```

Update, repaint, and sleep

The code in Listing 4 enters an infinite loop where it invokes the **update** method on the **SpriteManager** object to cause all the sprites to change their position. This causes sprites to move, causes collisions between sprites to occur, causes collisions to be handled, causes sprites to bounce off the walls, etc.

Then the code in Listing 4 invokes the **repaint** method on the **Frame** object. This sends a message to the operating system asking that the **Frame** object and all its contents be redrawn as soon as possible.

Tell me more about the repaint method

The repaint method of the **Frame** class is inherited from the **Component** class. Here is what Sun has to say about the **repaint** method of the **Component** class:

> *"This method causes a call to this component's update method as soon as possible."*

Now, tell me more about the update method

At this point, we need to take a look at the **update** method of the **Component** class. *(Don't confuse this method named update with the update method of the SpriteManager class. I now realize that it would have been less confusing if I had named the method in the SpriteManager class something other than update.)*

Here is part of what Sun has to say about the **update** method of the **Component** class:

> *"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."*

Is this the behavior that we want?

The above quotation from Sun describes the default behavior of the **update** method. Normally for non-animated programs, we would be happy with that default behavior and wouldn't override the **update** method. We would simply leave it alone and override the **paint** method to cause the overridden **paint** method to produce the output that we want to see on the screen.

Overriding the update method of the Component class

However, filling the component with the background color during every repaint can sometimes cause an undesirable flashing effect. As a result, animation programmers often override the **update** method to give it different behavior, and that is what I will do. I will discuss the behavior of my overridden **update** and **paint** methods in the next lesson.

Time for a little nap

Following the call to **repaint**, the thread goes to sleep for a period of time *(other activities could be taking place on other threads during this sleep period)*. The length of the sleep period is calculated such that the sleep period plus the processing time is approximately equal to 83 milliseconds *(twelve repaints per second).*

How accurate is the repaint rate?

Just how well this approach succeeds in achieving a uniform repaint rate of twelve repaints per second will depend on the accuracy of the time returned by the method named **currentTimeMillis**. *(This is the area where the use of a Timer object may be more reliable than the homebrew timer approach used in this program.)*

Time to wake up

The thread wakes up at the end of the specified sleep period. Each time the thread wakes up, it invokes another **update** on the **SpriteManager** object to cause the sprites to change their positions, requests another **repaint**, and goes back to sleep.

This process continues until the user terminates the program by clicking the close button on the **Frame**.

The end of the run method

That completes the discussion of the **run** method of the controlling class.

Before closing out this lesson, I'm going to explain the behavior of the **makeSprite** method that I used earlier to populate the **SpriteManager** object.

The makeSprite method

The **makeSprite** method is a short and very simple method. The entire method is shown in Listing 5.

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

Listing 5

A new Sprite object, please

This method instantiates and returns a new object of the **Sprite** class *(I will provide a detailed discussion of the Sprite class in a subsequent lesson).*

The constructor for the **Sprite** class requires four parameters:

- A reference to an **ImageObserver** object *(this)* that can be used later in calls to the **drawImage** method of the **Graphics** class.
- A reference to an **Image** object that can be used as the visual manifestation of the sprite.
- The initial position for the sprite.
- A reference to a **Point** object containing the horizontal and vertical components for the initial *motion vector* for the sprite.

The motion vector

Of these four parameters, only the motion vector is relatively new to us at this point *(the initial motion vector determines the initial direction and speed of motion for the sprite.)*.

The **makeSprite** method uses a random number generator to get the values for the components of the motion vector. The modulus operator (%) is used to guarantee that each of the component values is an integer value between -5 and +5.

An aside to this discussion

In a subsequent lesson, you will see that I am able to make major changes to the animation behavior of the program by making a very simple modification to the **makeSprite** method and by making changes to the definition of the **Sprite** class. Otherwise, all of the code that I have discussed so far will remain unchanged when I make those behavioral changes to the program.

Summary

In this 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.

Let's recap

This would probably be a good place to recap what we have learned so far. The controlling class extends the **Frame** class and implements the **Runnable** interface. Thus, an object of the controlling class is used to provide the visual manifestation of the program as a visual **Frame** object. An object of the controlling class is also suitable for using as an animation thread, which controls the overall behavior of the animation process. In other words, an object of the controlling class acts both as the director of the play, and the stage upon which the play is performed.

The constructor for the controlling class

The **main** method of the controlling class instantiates an object of the controlling class, thus causing the constructor for the controlling class to be executed.

The constructor for the controlling class causes seven **Image** objects to be created. Each **Image** object is based on the pixel contents of a GIF file.

The Image objects

One of the **Image** objects is used to produce the background scenery against which the animation is played out. The other six **Image** objects are used to provide the visual manifestation of the sprites. Each **Image** object provides the visual manifestation for more than one sprite. Therefore, some of the sprites look alike *(twins in some cases and triplets in others).*

After the **Image** objects have been created, the size of the **Image** object used for the background scenery is used by the constructor to set the size of the **Frame**. Then the **Frame** is made visible.

The animation thread

Finally, the constructor creates the animation thread and starts it running. From this point forward, the **run** method of the controlling class controls the animation behavior of the program.

The run method

The **run** method begins by creating and populating a **SpriteManager** object. An object of the **SpriteManager** class is capable of managing a collection of sprites, causing them to update their positions on demand, and dealing with collisions between the sprites.

The SpriteManager object

The **SpriteManager** object is populated with fifteen separate **Sprite** objects. Each sprite object has a visual manifestation based on one of the six **Image** objects. Each sprite object also has an initial position based on a random number and a *motion vector* whose components are also based on random numbers. The motion vector is used to determine the next position of the sprite when the sprite is told by the **SpriteManager** to change its position.

The animation loop

Then the **run** method enters an infinite loop, iterating approximately twelve times per second. At the beginning of each iteration, the **SpriteManager** is told to update the positions of all of the sprites in its collection. It does so, dealing with collisions in the process.

A repaint request

Once during each iteration, the **run** method sends a message to the operating system asking it to repaint the **Frame** object on the screen. That brings us to the point where we are right now.

Honoring the repaint request

When the operating system honors the request to repaint, it invokes the **upDate** method on the **Frame** object, *(which normally does some initialization and then invokes the paint method).* The **update** method is overridden in this program to cause the new scene to be drawn in its entirety, showing each of the sprites in its new position superimposed upon the background image. *(Note* *that in this case, the update method does not invoke the paint method, because there is nothing for the paint method to do.)*

An offscreen graphics context

When drawing the scene, the **update** method first draws the scene on an offscreen graphics context, and then causes the scene to be transferred from that context to the screen context. This is done to improve the animation quality of the program.

What's Next?

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

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 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-