Fun with Java: Frame Animation

Baldwin shows you how to achieve frame animation in Java. He accomplishes this by showing you how to upgrade the sprite animation program from the earlier lessons into a new program that provides both sprite animation and frame animation.

Published: November 25, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1464

Preface

Having fun with Java

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 eighth in of a group of lessons that will teach you how to write animation programs in Java.

Animation programs

The first lesson in the group was entitled Fun with Java: Sprite Animation, Part 1. That lesson, plus the six lessons following that one, provided an in-depth explanation of the use of Java for doing sprite animation.

The previous lesson, *(and the last in the series on sprite animation),* was entitled Fun with Java: Sprite Animation, Part 7.

Frame animation

In this lesson, I will move forward and teach you how to also do *frame animation* using Java. In fact, I will combine sprite animation with frame animation in this lesson.

What is the difference?

What is the difference between sprite animation and frame animation? This is how I view the difference between the two.

With sprite animation, an image moves around on the computer screen, but the look of that image doesn't change over time.

With frame animation, the look of an image can change over time, whether it moves or not.

Spherical sea creatures

In this program, I will combine the two such that spherical sea creatures will swim around in a fish tank. Each creature will change the way it looks over time. In particular, each creature will change its color as it swims.

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 in Java

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, which was discussed in the previous seven lessons, showed 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 that program is shown in Figure 1.

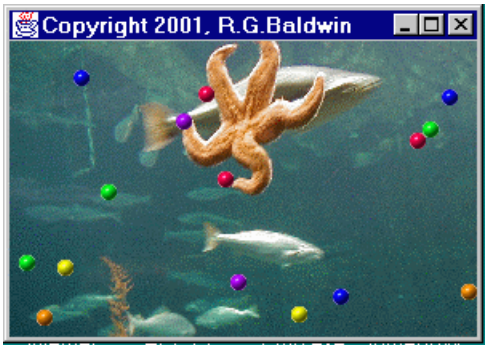

Figure 1. Animated spherical sea creatures in a fish tank.

Changing color

Many sea creatures have the ability to change their color in very impressive ways. The second program, which I will cover in its entirety in this lesson, will simulate that process using a combination of sprite and frame animation. *(This will really be an upgrade to the previous program, and much of the code from the previous program will be used.)*

Because a screen shot cannot represent changes over time, the screen shot shown in Figure 1 also represents the output from the program being discussed in this lesson.

Sea worms

The third program, to be discussed in the next 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.

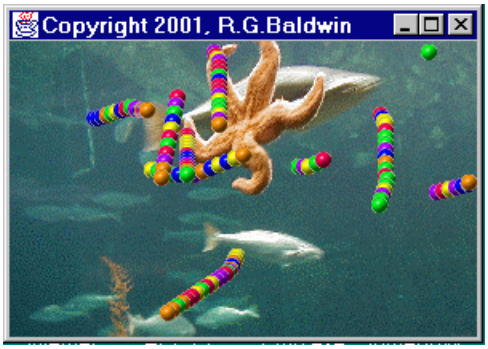

Figure 2. Animated sea worms in a fish tank.

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

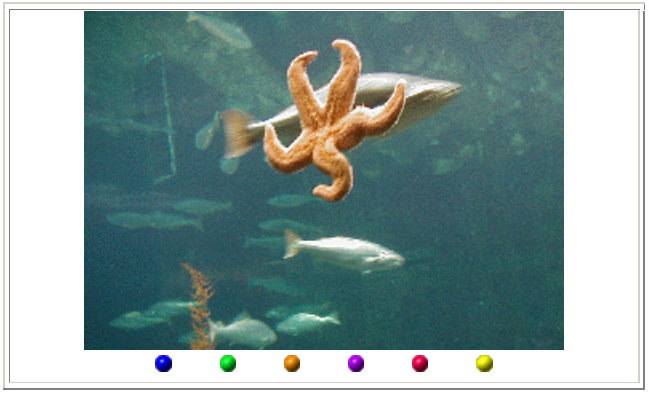

Figure 3. GIF image files that you will need.

Getting the images

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.

Discussion and Sample Program

Very similar to the previous program

Although this program is quite long, most of the program is identical to the program named **Animate01**, which was discussed in the previous series of lessons.

Only discuss new material

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

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

The makeSprite method

This program differs from the previous program in only two areas:

- The **Sprite** class from which new sprites are instantiated.
- The **makeSprite** method of the controlling class, which is used to create new sprites.

Listing 1 shows an abbreviated version of the controlling class with all of the code other than the new method named **makeSprite** removed for brevity.

```
public class Animate02 extends Frame
                    implements
Runnable{
  //All the methods in this class are
  // identical to those in Animate01
  // except for the method named
  // makeSprite.

  //Code deleted for brevity
  //--------------------------------
//

  private Sprite makeSprite(
       Point position, int
imageIndex){
    return new Sprite(
        this,
        gifImages,
        imageIndex,
        1,
        rand.nextInt() % 20,
        position,
        new Point(rand.nextInt() % 5,
                  rand.nextInt() %
5));
  }//end makeSprite()
  //--------------------------------
//

  //Code deleted for brevity

}//end class Animate02
```

Listing 1

The primary differences

The primary difference between this program and the previous program named **Animate01** can be summarized as follows:

Each Sprite had only one image

Each **Sprite** object in the previous program had an instance variable of type **Image**. The image whose reference was stored in the instance variable was used to provide the visual manifestation of the sprite on the screen. Because a given sprite owned a reference to only a single image, that sprite always looked the same on the screen.

Each Sprite has an array of images

In this program, each **Sprite** object has an instance variable, which is a reference to an array of references to **Image** objects.

The images stored in the **Image** objects are used to provide the visual manifestation of the sprites when they appear on the screen. However, because each sprite now owns references to more than one image, the sprite is able to change how it looks over time by using different images as its visual manifestation.

What is an animation frame?

Each of the images in the array is typically referred to as a *frame*. The process of causing a sprite to cycle through the frames, changing the way that it looks in the process, is often referred to as **frame animation**.

(Note that the use of the word frame in this context has nothing to do with the Java class named Frame.)

The substantive changes

All of the substantive changes in this program were made in the class named **Sprite**. However, because the argument list for the constructor of the **Sprite** class changed *(to accommodate more incoming information),* it was also necessary to modify the method named **makeSprite** shown in Listing 1.

The changes that were made had to do with changes to the parameter list for the **Sprite** constructor. The new parameter list includes a reference to an array of references to **Image** objects *(gifImages)* along with some other new parameters that deal with the manner in which the sprite cycles through the frames.

The Sprite class

As before, the **Sprite** class is the workhorse of this animation program.

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

Where are you?

For example, an object of the **Sprite** class knows how to tell other objects about the space that it occupies in the fish tank.

What is your speed and direction?

It knows how to tell other objects about its motion vector, which determines the speed and direction of its motion.

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

Oops, just hit a wall!

It knows what to do if it runs into one of the walls of the fish tank. Basically, it bounces off the wall much like a pool ball bounces off the cushions on a pool table. When this happens, it modifies its motion vector accordingly.

Please draw a picture of yourself

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

Did we just collide?

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

Change your appearance

Finally, this new and improved sprite knows how to use an array of images to change how it looks over time.

This is a very key class insofar as this program is concerned. The behavior of the methods in this class determines the overall behavior of the animation process.

Discuss in fragments

As usual, I will discuss the program in fragments.

Most of the code in this revised **Sprite** class is identical to the code in the **Sprite** class used in the previous program named **Animate01**. In keeping with the spirit of this lesson, I will not discuss the code that I discussed in the previous lessons. Rather, for the most part, I will discuss only that code that is new or different. When you see **//...** in the code fragments, that means that code was omitted for brevity.

The beginning of the Sprite class

Listing 2 shows the beginning of the **Sprite** class along with some new or different declarations for instance variables.

```
class Sprite {
  private Image[] image;
```

```
  private int frame;
  private int frameDisplayIncrement;
  private int frameDisplayDuration;
  private int frameDurationCounter;
  //...

Listing 2
```

An array of Image references

In the earlier version, the reference variable named **image** was of type **Image**. However, in this version, it is of type **Image[]**. In other words, it previously referred to a single object of type **Image**. Now it refers to an array of references to **Image** objects.

The constructor

Listing 3 shows the signature for the constructor. The significant thing about Listing 3 is the makeup of the formal argument list, which I will discuss below.

```
  public Sprite(//constructor
              Component component,
              Image[] image,
              int startingFrame,
              int
frameDisplayIncrement,
              int
frameDisplayDuration,
              Point position,
              Point motionVector){

Listing 3
```

More parameters required

The constructor for the new **Sprite** class takes seven parameters, as opposed to only four parameters for the previous version.

As before, the first parameter is a reference to a **Component** object. In fact, this parameter is assumed to be a reference to the **Frame** object in which this animation is run.

This parameter is used to determine the size of the **Frame**. It is also used as a required **ImageObserver** in some of the methods in the class. *(I discussed image observers in an earlier lesson in this series.)*

An array of Image references

Whereas previously, the second parameter was a reference to an object of type **Image**, the second parameter in the new version is a reference to an array containing references of objects of type **Image**.

The images in the array are used to provide a visual manifestation for the sprite, and that visual manifestation changes over time.

Cycling through the animation frames

The third, fourth, and fifth parameters are new to this version of the **Sprite** class. These parameters are used to determine how the sprite cycles through the set of images in the array.

The startingFrame parameter

The parameter named **startingFrame** specifies the first frame to use at the beginning of the frame animation process. In effect, this determines the color of a sprite when it first appears on the screen. If you examine the code in Listing 8 near the end of this lesson, you will see that different values are used for the fifteen sprites that populate the fish tank.

The frameDisplayIncrement parameter

The parameter named **frameDisplayIncrement** allows for skipping some of the images in the array of images.

In this program, this value was set to 1 for all sprites. Thus, when cycling through the images, each sprite uses every image as its visual manifestation.

The frameDisplayDuration parameter

The parameter named **frameDisplayDuration** is used to determine how long a given image will be used as the visual manifestation for a sprite.

In this program, this value was specified as a random number with a maximum value of 20 for each new sprite. Thus, each sprite will cycle through the six images in the array at a different rate.

Same as previous parameters

The sixth and seventh parameters for the new constructor are the same as the third and fourth parameters of the constructor in the previous program. The sixth parameter named **position** specifies the initial position of the sprite.

The seventh parameter named **motionVector** is a motion vector, which determines the initial speed and direction of motion for a new **Sprite**, object.

Some new code

Listing 4 shows some of the code in the constructor that is new to this version of the program.

```
    //...
    frame = startingFrame;
    this.frameDisplayIncrement =
                 frameDisplayIncrement;

    frameDisplayDuration =
        Math.abs(frameDisplayDuration);

    if(frameDisplayDuration < 5)
      frameDisplayDuration = 5;
    this.frameDisplayDuration =
                  frameDisplayDuration;

    frameDurationCounter =
             this.frameDisplayDuration;
    //...
```

Listing 4

The first two statements in Listing 4 simply save the incoming values for **startingFrame** and **frameDisplayIncrement** in the appropriate instance variables.

Duration must be positive

The code having to do with the **frameDisplayDuration** does two things. First it uses the **abs** method of the **Math** class to guarantee that the duration is a positive value.

Minimum duration is five animation cycles

Second, it guarantees that the value is at least 5. This means that each sprite object will use an **Image** object as its visual manifestation for at least five animation cycles *(or about one-half second at twelve frames per second).*

In this program, this means that each sprite will remain the same color for at least five animation cycles.

The frameDurationCounter

As you may have surmised earlier, the instance variable named **frameDurationCounter** is used to determine when a sprite chooses a new **Image** object as its visual manifestation *(when it changes color in this program).* The value of the **frameDurationCounter** is initialized in the code of Listing 4.

All of the remain code in the constructor is essentially the same as in the previous version of the program, so I won't discuss it here.

The incFrame method

That brings us to a method named **incFrame**, which is completely new to this version of the program. The entire method is shown in Listing 5.

```
  private void incFrame() {
    if ((frameDisplayDuration > 0) &&
        (--frameDurationCounter <= 0)){
      // Reset the frame trigger
      frameDurationCounter =
                  frameDisplayDuration;

      // Increment the frame
      frame += frameDisplayIncrement;
      if (frame >= image.length)
        frame = 0;
      else if (frame < 0)
        frame = image.length - 1;
    }//end if
  }//end incFrame

Listing 5
```

Later we will see that this method is invoked each time the sprite is requested to update its location on the screen *(once during each animation cycle).*

Decrement the frameDurationCounter

The code in this method is fairly ugly, particularly in this narrow publication format.

When you wade through it, you discover that it decrements the **frameDurationCounter**, *(which was initialized to the value of the frameDisplayDuration)* each time the **incFrame** method is invoked.

Reset the counter and increase the frame variable

When the counter hits zero,

- The counter is reset back to the value of the **frameDisplayDuration**
- The value of the variable named **frame** is increased by the value of the **frameDisplayIncrement**

Choose an image

Later we will see that the value of **frame** is used to choose an image from the array of images as the visual manifestation of the sprite.

Bracket the value of the frame variable

Some testing is done to ensure that the value for **frame** is not greater than the number of images in the array of images and is not less than zero. If so, the value of **frame** is corrected appropriately.

Frame control code

This is essentially the control code that determines how the sprite cycles through the array of images, selecting an image for its visual manifestation during each animation cycle. There are only about two more statements in the entire **Sprite** class that have anything to do with frame animation. The first of those appears in Listing 6.

Revised updatePosition method

The code in Listing 6 shows the beginning of the revised **updatePosition** method of the **Sprite** class.

You may recall from the previous lessons that this is the method that the **SpriteManager** invokes, once during each animation cycle, to cause each individual sprite to draw itself on the screen in its new position.

```
  public void updatePosition() {
    incFrame();//increment the frame
    //...

Listing 6
```

Incrementing the frame variable

As you can see, the first statement in this new version of the method invokes the **incFrame** method discussed above to cause the value of the variable named **frame** to change when appropriate.

As you can also see, a great deal of code that is essentially the same as in the previous program was deleted for brevity, taking us all the way down to the revised method named **drawSpriteImage** shown in Listing 7.

The revised drawSpriteImage method

As you may recall from the previous lessons, the **drawSpriteImage** method is the method that the **SpriteManager** invokes on each **Sprite** object to cause the sprite to draw itself onto the offscreen graphics context once during each animation cycle

```
  public void drawSpriteImage(
                           Graphics
g){
```

```
    g.drawImage(image[frame],
                spaceOccupied.x,
                spaceOccupied.y,
                component);
  }//end drawSpriteImage()

Listing 7
```

What is the difference?

The difference between the new version of the **drawSpriteImage** method and the previous version is so important that I have provided the previous version in a different color in Listing 8 below for comparison.

```
  public void drawSpriteImage(
                            Graphics
g){
    g.drawImage(image,
                spaceOccupied.x,
                spaceOccupied.y,
                component);
  }//end drawSpriteImage()

//--NOTE: THIS IS CODE FROM AN EARLIER
PROGRAM--

Listing 8
```

A reference to an Image object

The thing to note in these two listings is the invocation of the **drawImage** method. In both versions, the first parameter to the **drawImage** method is a reference to the **Image** object that will be rendered onto the screen as the visual manifestation of the sprite.

One Image object available

However, in the previous version of the program, this is a reference to a single **Image** object passed into the **Sprite** constructor when the **Sprite** object was instantiated.

A choice of Image objects

In the new version, this is a reference extracted from an array of references using the **frame** variable as an index into the array.

The reference to the array of **Image** objects was received as an incoming parameter to the constructor when the **Sprite** object was instantiated. In the new version, the **incFrame** method discussed earlier is used to establish the value stored in **frame**, which identifies the **Image** object that is used to provide the visual manifestation for the sprite during each animation cycle.

Six colored spheres

In this program, the array contains references to images of six small spheres of different colors. As the animation progresses and the **incFrame** method controls the value of the variable named **frame**, different colored spheres are selected as the visual manifestation of a sprite, and the sprites appear to change color over time.

The remaining code in the **Sprite** class was discussed in earlier lessons and therefore won't be discussed further here.

Summary

In this lesson, I have taught you how to achieve frame animation in Java. I accomplished this by showing you how to upgrade the sprite animation program from the earlier lessons into a new program that provides both sprite animation and frame animation.

What's Next?

In the next lesson in this series, I will provide an additional upgrade to convert the spherical sea creatures into sea worms that have the ability to slither around in the fish tank and to change the colors of different parts of their bodies in a random fashion as they slither.

Complete Program Listing

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

```
/*File Animate02.java
Copyright 2001, R.G.Baldwin
This program displays several animated
colored spherical sea creatures
swimming around in an aquarium. Each
creature maintains generally the same
course until it collides with another
creature or with a wall.  However,
the creatures have the ability to
change course based on the addition or
subtraction of random values from the
components of their motion vector
about once in every ten updates.

In addition, each creature uses frame
animation to change colors in the order
red, green blue, yellow, purple,
orange.  Each creature switches among
the colors on a periodic rate, but the
period may be different for each
creature.  Thus, each creature cycles
through the same colors, but remains a
```

```
particular color for a different
amount of time than the other
creatures.  The amount of time that a
creature remains a particular color is
based on a random number between 5
and 20 updates.

The primary changes in this program
relative to the program named Animate01
were made in the Sprite class, and in
calls to the constructor for the Sprite
class in the makeSprite() method.

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

public class Animate02 extends Frame
                  implements Runnable{
  //All the methods in this class are
  // identical to those in Animate01
  // except for the methodnamed
  // makeSprite.
  private Image offScreenImage;
  private Image backGroundImage;
  private Image[] gifImages =
                          new Image[6];
  //offscreen graphics context
  private Graphics
                  offScreenGraphicsCtx;
  private Thread animationThread;
  private MediaTracker mediaTracker;
  private SpriteManager spriteManager;
  //Animation display rate, 12fps
  private int animationDelay = 83;
  private Random rand =
                new Random(System.
                  currentTimeMillis());

  public static void main(
                        String[] args){
    new Animate02();
  }//end main
  //---------------------------------//

  Animate02() {//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,
        1,
        rand.nextInt() % 20,
        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 Animate02
//===================================//

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

  public BackgroundImage(
                  Component component,
                  Image image) {
    this.component = component;
    size = component.getSize();
    this.image = image;
  }//end construtor

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

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

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

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

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

  private int frame;
  private int frameDisplayIncrement;
  private int frameDisplayDuration;
  private int frameDurationCounter;

  public Sprite(//constructor
              Component component,
              Image[] image,
              int startingFrame,
              int frameDisplayIncrement,
              int frameDisplayDuration,
              Point position,
              Point motionVector){
    //Seed a random number generator
    // for this sprite with the sprite
    // position.
    rand = new Random(position.x);

    frame = startingFrame;
    this.frameDisplayIncrement =
                  frameDisplayIncrement;
    frameDisplayDuration =
        Math.abs(frameDisplayDuration);
```

```
    if(frameDisplayDuration < 5)
      frameDisplayDuration = 5;
    this.frameDisplayDuration =
                  frameDisplayDuration;
    frameDurationCounter =
             this.frameDisplayDuration;
    this.component = component;
    this.image = image;
    setSpaceOccupied(new Rectangle(
       position.x,
       position.y,
       image[0].getWidth(component),
       image[0].getHeight(component)));
    this.motionVector = motionVector;
    //Compute edges of usable graphics
    // area
    int topBanner = (
                 (Container)component).
                       getInsets().top;
    int bottomBorder = (
                 (Container)component).
                    getInsets().bottom;
    int leftBorder = (
                 (Container)component).
                      getInsets().left;
    int rightBorder = (
                 (Container)component).
                     getInsets().right;
    bounds = new Rectangle(
          0 + leftBorder,
          0 + topBanner,
          component.getSize().width -
            (leftBorder + rightBorder),
          component.getSize().height -
           (topBanner + bottomBorder));
  }//end constructor
  //---------------------------------//

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

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

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

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

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

  public void setBounds(
                    Rectangle bounds){
    this.bounds = bounds;
  }//end setBounds()
  //---------------------------------//

  private void incFrame() {
    if ((frameDisplayDuration > 0) &&
        (--frameDurationCounter <= 0)){
      // Reset the frame trigger
      frameDurationCounter =
                  frameDisplayDuration;

      // Increment the frame
      frame += frameDisplayIncrement;
      if (frame >= image.length)
        frame = 0;
      else if (frame < 0)
        frame = image.length - 1;
    }//end if
  }//end incFrame
  //---------------------------------//

  public void updatePosition() {
    incFrame();//increment the frame

    Point position = new Point(
     spaceOccupied.x, spaceOccupied.y);

    //Insert random behavior.  During
    // each update, a sprite has about
    // one chance in 10 of making a
    // small random change to its
    // motionVector.  When a change
    // occurs, the motionVector
    // coordinate values are forced to
    // fall between -7 and 7.
    if(rand.nextInt() % 10 == 0){
      Point randomOffset =
         new Point(rand.nextInt() % 3,
                   rand.nextInt() % 3);
      motionVector.x += randomOffset.x;
      if(motionVector.x >= 7)
        motionVector.x -= 7;
      if(motionVector.x <= -7)
```

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

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

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

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

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

    //save new motionVector
```

```
    if (bounceRequired)

     setMotionVector(tempMotionVector);

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

  public void drawSpriteImage(
                            Graphics g){
    g.drawImage(image[frame],
                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 8

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-