Fun with Java: Sprite Animation, Part 7

Baldwin completes his sprite-animation program. Along the way, he explains the methods of the Sprite class, including the following features: establishing the initial position of the sprite, determining the location of the sprite, determining the speed and direction of the sprite, updating the sprite's position, implementing some randomness in the sprite's motion, bouncing off the walls, drawing the sprite, and detecting collisions with other sprites.

Published: November 18, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1462

- Preface
- Preview
- Discussion and Sample Programs
- Summary
- What's Next
- Complete Program Listing

Preface

Why the intro?

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.

Fun programming

This miniseries will include a variety of Java programming topics that fall in the category of *fun programming*. This particular lesson is the seventh 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 6.

This is the final lesson in the group dedicated to sprite animation. The next lesson in this group will be dedicated to a combination of sprite and frame animation.

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

Writing animation programs

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.

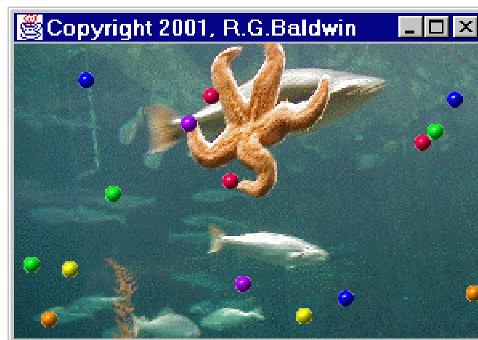

Figure 1. Animated spherical sea creatures in a fish tank.

Changing color

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.

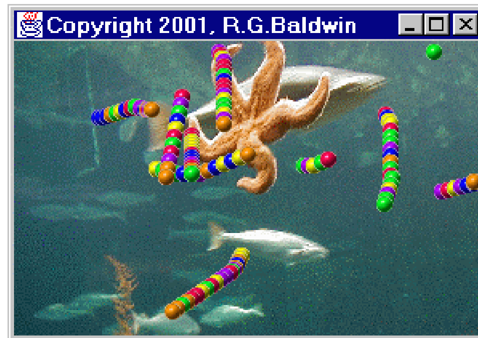

Figure 2. Animated sea worms in a fish tank.

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

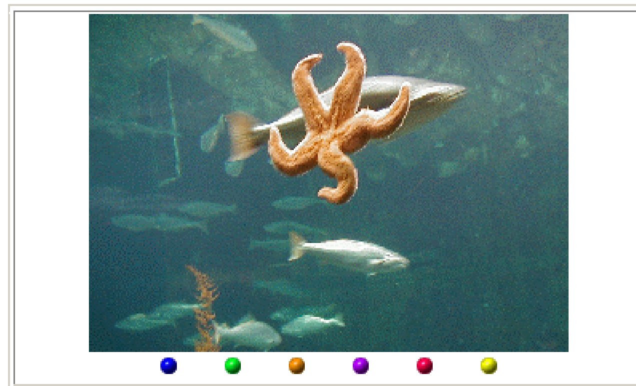

Figure 3. GIF image files that you will need.

Getting the 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.

In the previous lesson, I explained the behavior of all of the methods in the **SpriteManager** class.

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

Preview

I will explain the methods of the **Sprite** class in this lesson. This will include an explanation of the following features:

- Establishing the initial position for the sprite
- Determining the location of the sprite
- Determining the speed and direction of the sprite
- Updating the sprite's location
- Implementing some randomness in the sprite's motion
- Bouncing off the wall
- Drawing the sprite
- Detecting collisions with other sprites

Discussion and Sample Program

Several lessons required

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 12 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 Sprite class

The **Sprite** class is the workhorse of this program *(perhaps I should have used seahorse images for the sprites instead of simple balls)*.

Objects of the Sprite class

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.

Get out of my space

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

Get out of my way

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

I'm outta here

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, I hit the 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.

This is what I look like

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

Pardon me!

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

A key class

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.

Two more animation programs

As indicated in the preface section above, I plan to discuss at least two more animation programs in this series on animation programs.

Sprites with different behavior

The behavior of the sprites will be significantly different in each of the three programs. For example, the second program combines sprite and frame animation to produce animated sea creatures that change their color as they swim through the water.

The third program combines sprite animation, frame animation, and some other techniques to convert the spherical sea creatures into sea worms that slither through the water and change their colors in the process.

Heavy code reuse

When developing the two additional programs, I will make very heavy use of the code that I have developed for this program. Most of the changes that I will make will be made to the

Sprite class. Very few changes will be required outside the **Sprite** class in the development of the other two programs.

Discuss in fragments

As usual, I will discuss the program in fragments.

The definition for the **Sprite** class begins in Listing 1. Listing 1 shows the declaration of several instance variables and the signature for the constructor.

```
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){

Listing 1
```

Descriptive variable names

The names of the instance variables are generally descriptive of their use, so you can probably surmise in general what they are used for. We will learn in detail what each is used for as we examine the code.

Constructor parameters

The constructor for the **Sprite** class takes four parameters.

First parameter is a Component

The first parameter is a reference to a **Component** object. In fact, it is assumed to be a reference to the **Frame** object in which this animation is execution.

The first 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.)*

Second parameter is an Image

The second parameter is a reference to an object of type **Image**. The **Image** is used to provide a visual manifestation for the sprite. *(In a later program, this parameter will be replaced by an array of Image objects where each element in the array represents one frame in a frame-animation sequence.)*

Initial position

The third parameter is the initial position of the sprite.

A motion vector

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

A pseudorandom number generator

The code in Listing 2 instantiates an object of the class **Random** from which pseudorandom numbers can be extracted later in the program. The reference to this object is stored in one of the instance variables listed above.

```
    //Seed a random number generator
    // for this sprite with the sprite
    // position.
    rand = new Random(position.x);
```

Listing 2

You have seen the use of the **Random** class in other areas of this program. The thing that is interesting about this **Random** object is the manner in which it is seeded.

A different seed is required

Previous uses of the **Random** class used a seed based on the time in milliseconds that the object is instantiated. However, that is not a suitable approach here, because it is possible to instantiate a large number of **Sprite** objects within a single millisecond. If the time in milliseconds were used as the seed in this case, many **Sprite** objects would contain **Random** objects that produce the same sequence of pseudorandom numbers. Then, the numbers wouldn't really be random, at least not between **Sprite** objects.

Seed with initial position

Therefore, in this case, the **Random** object was seeded with the initial position of the sprite. *(Recall that code discussed in an earlier lesson went to great lengths to make certain that the initial position of each new sprite is different from the current position of any existing sprite.)*

The constructor body

The code shown in Listing 3 is the beginning of the body of the constructor. The purpose of this code is simply to set the initial values for some of the instance variables in the new object. This code is reasonably straightforward, and doesn't deserve much in the way of discussion.

```
    this.component = component;
    this.image = image;
    setSpaceOccupied(new Rectangle(
          position.x,
          position.y,
          image.getWidth(component),
          image.getHeight(component)));
    this.motionVector = motionVector;

Listing 3
```

The size of the Frame

Recall that this animation runs inside a **Frame** object. The **size** property of a **Frame** object is represented by the *outer* dimensions of the **Frame**. This includes the width of the banner at the top and the borders on three sides.

Save the usable graphics area

The purpose of the code in Listing 4 is to determine and save the usable graphics area inside the banner and the borders.

Although rather ugly, this code is straightforward. Perhaps the only thing worth mentioning in this code is the use of the **getInsets** method of the **Container** class to determine the size of the banner at the top and the borders on the three sides.

```
    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
```

Listing 4

Once the inset values are obtained, simple arithmetic is used to combine them with the size information for the **Frame** to produce a **Rectangle** object that describes the usable graphics area inside the borders and the banner. This is the rectangle that is used later to cause the sprites to bounce off the inside edges of the borders and the banner.

Typical setter and getter methods

The methods in Listing 5 are typical property setter and getter methods. There is nothing about these methods that deserves any discussion.

```
  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()
  //--------------------------------
//

Listing 5
```

The updatePosition method

That brings us to one of the most important methods in the **Sprite** class: **updatePosition**.

This method is invoked by the **SpriteManager** object on each **Sprite** object each time the animation process needs to be updated.

The motion vector

Each **Sprite** object has an instance variable named **motionVector** that contains an x-component and a y-component. The values of these two components determine the direction and distance that a sprite will move during each incremental change in position.

The motion vector can change

If the values of these two components don't change, a sprite will continue in the same direction at the same speed forever. However, there are three ways that the value of one or both of the components can change:

1. By running into the inside edge of the border or banner on the **Frame**
2. By colliding with another sprite
3. By adding a random value

Colliding with the edge

For the first case, the change in component values is handled by code inside the **updatePosition** method that detects collision with the edge and takes appropriate action.

Colliding with another sprite

For the second case, the change in component values is handled by a method in the **SpriteManager** class that deals with collisions between sprites.

A random change in speed and direction

For the third case, the change in component values is handled by code inside the **updatePosition** method. This code is designed to insert a small amount of random behavior into the motion of the sprites.

Finally, the updatePosition method code

Listing 6 shows the beginning of the **updatePosition** method. Listing 6 also shows the declaration and initialization of a local variable describing the current position of the sprite.

```
  public void updatePosition(){
    Point position = new Point(
     spaceOccupied.x,
spaceOccupied.y);
```

Listing 6

Insert random behavior

The code in Listing 7 purposely inserts some random behavior into the motion of the sprite by occasionally making a small random change to the component values of the motion vector.

One change in ten

The code is structured to make such a random change about once in every ten incremental moves of the sprite. This is accomplished by generating a random number modulo 10 each time the sprite is instructed to update its position. On those occasions that the random number modulo 10 equals 0, a random change is made to both components of the motion vector

```
    //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
```

Listing 7

Random change, -3 to 3 units

On those occasions that the random number modulo 10 equals 0, the code inside the **if** statement in Listing 7 is executed. This code generates a **randomOffset** of type **Point**, whose component values fall between -3 and +3.

The values of the components in the **randomOffset** are then added to the components of the motion vector to cause a random change in speed and direction.

Limit the speed to seven units per update

However, there is a problem with this. Up to this point in the code, it is possible that a series of cumulative adjustments can be made in the same direction, resulting in large incremental steps during each position update. This manifests itself as a sea creature that swims very fast.

In order to prevent this, after the adjustment to the motion vector is made, tests are performed to determine if the absolute value of either component exceeds a value of 7. If so, an additional adjustment is made by either adding or subtracting the value 7. This has the effect of limiting the speed of any sprite to no more than 7 units per update on either axis.

Experimental values

You might wonder how I arrived at the values of 3 and 7 in the above code. I arrived at those values simply by experimenting with different values and choosing values that produced a pleasing animation. You, of course, can change the values to either speed the sprites up, or slow them down.

Move the sprite

Just in case you may have lost your place, we are still inside the **updatePositon** method.

The single statement in Listing 8 uses the **translate** method of the **Point** class to effect the actual movement of the sprite.

```
    position.translate(
       motionVector.x,
motionVector.y);

Listing 8
```

This code adds the components of the motion vector to the corresponding component of the sprite's current position.

Bounce off the walls

When a sprite moves, it may collide with the inside edge of the banner or one of the borders.

The really long and ugly code in listing 9 causes the sprite to bounce whenever it collides with the inside edge of the banner or one of the borders *(identified by the bounds computed earlier).*

```
    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()
```

Listing 9

Reverse direction along one axis

This code is long and ugly, but basically straightforward. Tests are performed to determine if the new sprite position collides with the edges defined by **bounds**. If so, depending on which edge is involved in the collision, the appropriate motion vector component is modified to send the sprite off in the opposite direction.

For example, if the sprite is moving toward the top of the screen *(negative Y direction)* and collides with the inside edge of the banner, the sign on the y-component of the motion vector will be changed to positive so that the sprite will move down the screen on the next call to **positionUpdate**.

The drawSpriteImage method

The **drawSpriteImage** method shown in Listing 10 gives a **Sprite** object the ability to draw itself on a graphics context received as an incoming parameter.

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

Listing 10

This method uses the **drawImage** method of the **Graphics** class to accomplish the drawing *(I have discussed this method in previous lessons).*

The first parameter to the **drawImage** method specifies the **Image** object that will be drawn. The second and third parameters specify the location within the graphics context where it will be drawn. Finally, the fourth parameter is a reference to the **Frame** on which the image is being drawn, which serves an **ImageObserver**.

In this program, the **drawSpriteImage** method is invoked on each **Sprite** object from within the **drawScene** method of the **SpriteManager** object.

The testCollision method

The **testCollision** method shown in Listing 11 determines if the space occupied by this **Sprite** object intersects the space occupied by another sprite received as an incoming parameter.

```
  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

Return true for collision

If there is an intersection between the spaces occupied by the two sprites, this method returns true. Otherwise, it returns false.

In this program, the **testCollision** method on the **Sprite** object is called by the **testForCollision** method of the **SpriteManager** object when it is in the process of determining if the latest move by all of the sprites resulted in any collisions.

Summary

In this lesson, I have explained all of the methods in the **Sprite** class.

In this and the previous six lessons, I have explained the following aspects of this animation program.

The director and the stage

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.

Instantiate Image objects

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 background scenery

One of the **Image** objects is used to produce the background scenery against which the animation is played out.

Six colored spheres

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).*

Set the Frame size

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.

Run 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.

Needed, one sprite manager

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.

Fifteen Sprite objects

The **SpriteManager** object is populated with fifteen separate **Sprite** objects. Each sprite 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
- A *motion vector* whose components are also based on random numbers

The purpose of the initial position should be intuitive. 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.

Twelve updates per second

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.

The **run** method sends a message to the operating system asking it to repaint the **Frame** object on the screen.

An overridden update method

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.

The paint method is not invoked

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 drawing 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.

The end result

The end result is a set of fifteen spherical sea creatures *(sprites)* swimming around in a fish tank against a background captured from a photograph that I took while on a recent trip to the aquarium in Monterey, CA.

What about some jellyfish?

However, any GIF file of appropriate size could be used for the background. Also, any GIF files of appropriate size could be used for the sprites.

(I just remembered that I also have some photos of jellyfish that would make good sprites in this context, but I'm not going to go back and rewrite all of this just to substitute the jellyfish for the spherical sea creatures.)

What's Next?

In the next lesson in this series, I will expand the program to incorporate both sprite animation and frame animation.

Complete Program Listing

A complete listing of the program is provided in Listing 12.

```
/*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 12

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

About the author

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

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

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

baldwin.richard@iname.com

-end-