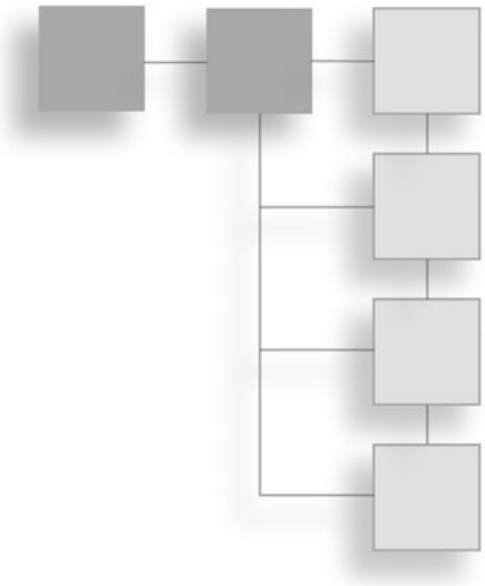
Chapter 6



Simple Sprites

Up to this point you have learned about a lot of Java classes that are useful for making a game, particularly the `Graphics2D` class. The previous two chapters provided the groundwork for this chapter by showing you how to tap into the `Graphics2D` class to draw vectors and bitmaps. At this point, the source code for even a simple bitmap-based game will tend to be too complicated and too difficult to manage without a better way to handle the objects in a game. What you need at this point is a new class that knows how to work with game objects—something known as an *actor* or a *sprite*. The goal of this chapter is to develop a way to handle the game objects moving around on the screen.

Here are the specific topics covered in this chapter:

- Programming simple sprites
- Creating a `Sprite` class
- Learning about collision testing

Programming Simple Sprites

A sprite usually represents an animated graphic image that moves around in a game and is involved with other images to form the gameplay of a game. The difference between a regular image and a sprite is often that a sprite will encapsulate the image data as well as the methods needed to manipulate it. We will create a new class later in this chapter called `ImageEntity`, which will be able

to load and draw a bitmap, and we will then create a new `Sprite` class that will use `ImageEntity`. Animation will be held off until the next chapter.

I would like to build a pair of classes to simplify sprite programming. We will create the `Sprite` class in this chapter and then add the `AnimatedSprite` class in the next chapter to handle animation. The new `Sprite` class that I'm going to show you here might be described as a *heavy class*. What do I mean by "heavy"? This is not a simple, abstract class. Instead, it is tied closely to the `JFrame` and `Graphics2D` objects in our main program. You would not be able to use this `Sprite` class on its own in a Java applet (running in a web browser) without modifications, because it relies on the presence of the JFrame to work. Although it is possible to write a Java game that runs in a JFrame or a JApplet (which are both somewhat related), the code to support both applications and applets at the same time is messy. Our new `Sprite` class will work fine as a support class for an application-based Java game. If we want to use it in an applet, minor modifications can be made (they are trivial!).

A sprite cannot draw itself without the `JFrame` and `Graphics2D` objects in a main program. Although the `Sprite` class could use methods such as `getGraphics()` to pull information from the main applet, our examples use a double buffer (a back buffer image used to update graphics smoothly, without flickering the screen).

The `BaseGameEntity` class will handle all of the position, velocity, rotation, and other logistical properties, while `ImageEntity` will make use of them by providing methods such as `transform()` and `draw()`. I want to simplify the `Sprite` class so it doesn't expose all of these properties and methods, but provides a simpler means to load and draw images. This simplification will be especially helpful in the next chapter because animation tends to complicate things. Although we have three classes just to draw a single sprite, there's reason behind this apparent madness—I don't want to duplicate code in all of the game entities if it can be helped. In the next few chapters we'll be drawing vectors *and* sprites, and it is helpful if we can reuse some of the code.

A useful sprite class should handle its own *position* and *velocity* data, rather than individual X and Y values for these properties. The sprite's position and velocity will be handled by the `BaseGameEntity` class. The `Sprite` class will not inherit from `ImageEntity`; instead, it will *use* this class internally, like a regular variable.

I also want the get methods that return values to resemble simple properties, while the change methods will be in the usual "set" format. For instance, I want the `Sprite` class to have a `position()` method that returns the position of the `Sprite` object, but it will use a `setPosition()` method to change the X and Y values. We should be able to access position and velocity by writing code like this:

```
sprite.position().x
sprite.position().y
```

Whenever possible, we will forego *good* object-oriented design in favor of *simpler* source code. This is especially helpful for beginners who may have never seen a truly huge source code listing, and thus would not understand why such things are important.

On top of these requirements, we should not be concerned with numeric data types! I don't want to typecast integers, floats, and doubles! So, this `Sprite` class will need to deal with the differences in the data types automatically and not complain about it! These are minor semantic issues, but they tend to seriously clean up the code. The result will be a solidly built sprite handler. First, let's take a look at a support class that will make it possible.

Tip

An *accessor method* is a method that returns a private variable in a class. A *mutator method* is a method that changes a private variable in a class. These are also commonly called "get" and "set" methods.

Basic Game Entities

The `BaseVectorShape` class was introduced back in Chapter 3 for the *Asteroids*-style game. We will use a very similar class for sprite programming in a future version of Galactic War (beginning in Chapter 11). Here is the code for this class.

```
public class BaseGameEntity extends Object {
    //variables
    protected boolean alive;
    protected double x,y;
    protected double velX, velY;
    protected double moveAngle, faceAngle;
```

```
    //accessor methods
    public boolean isAlive() { return alive; }
    public double getX() { return x; }
    public double getY() { return y; }
    public double getVelX() { return velX; }
    public double getVelY() { return velY; }
    public double getMoveAngle() { return moveAngle; }
    public double getFaceAngle() { return faceAngle; }

    //mutator methods
    public void setAlive(boolean alive) { this.alive = alive; }
    public void setX(double x) { this.x = x; }
    public void incX(double i) { this.x += i; }
    public void setY(double y) { this.y = y; }
    public void incY(double i) { this.y += i; }
    public void setVelX(double velX) { this.velX = velX; }
    public void incVelX(double i) { this.velX += i; }
    public void setVelY(double velY) { this.velY = velY; }
    public void incVelY(double i) { this.velY += i; }
    public void setFaceAngle(double angle) { this.faceAngle = angle; }
    public void incFaceAngle(double i) { this.faceAngle += i; }
    public void setMoveAngle(double angle) { this.moveAngle = angle; }
    public void incMoveAngle(double i) { this.moveAngle += i; }

    //default constructor
    BaseGameEntity() {
        setAlive(false);
        setX(0.0);
        setY(0.0);
        setVelX(0.0);
        setVelY(0.0);
        setMoveAngle(0.0);
        setFaceAngle(0.0);
    }
}
```

The ImageEntity Class

The `ImageEntity` class gives us the ability to use a bitmap image for the objects in the game instead of just vector-based shapes (such as the asteroid polygon). It's never a good idea to completely upgrade a game with some new technique,

which is why some of the objects in the first version of Galactic War will still be vectors, while the player's ship will be a bitmap. When you reach Chapter 11, you will have an opportunity to examine the progression of the game from its meager beginning to a complete and complex game with sprite entity management. I always recommend making small, incremental changes, play-testing the game after each major change to ensure that it still runs. There's nothing more frustrating than spending two hours making dramatic changes to a source code file, only to find the changes have completely broken the program so that either it will not compile or it is full of bugs.

The `ImageEntity` class also inherits from the `BaseGameEntity` class, so it is related to `VectorEntity`. This class is awesome! It encapsulates all of the functionality we need to load and draw bitmap images, while still retaining the ability to rotate and move them on the screen!

```
// Base game image class for bitmapped game entities
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
import java.net.*;

public class ImageEntity extends BaseGameEntity {
    //variables
    protected Image image;
    protected JFrame frame;
    protected AffineTransform at;
    protected Graphics2D g2d;

    //default constructor
    ImageEntity(JFrame a) {
        frame = a;
        setImage(null);
        setAlive(true);
    }

    public Image getImage() { return image; }

    public void setImage(Image image) {
        this.image = image;
        double x = frame.getSize().width/2- width()/2;
```

```
        double y = frame.getSize().height/2 - height()/2;
        at = AffineTransform.getTranslateInstance(x, y);
    }

    public int width() {
        if (image != null)
            return image.getWidth(frame);
        else
            return 0;
    }
    public int height() {
        if (image != null)
            return image.getHeight(frame);
        else
            return 0;
    }

    public double getCenterX() {
        return getX() + width() / 2;
    }
    public double getCenterY() {
        return getY() + height() / 2;
    }

    public void setGraphics(Graphics2D g) {
        g2d = g;
    }

    private URL getURL(String filename) {
        URL url = null;
        try {
            url = this.getClass().getResource(filename);
        }
        catch (Exception e) { }
        return url;
    }

    public void load(String filename) {
        Toolkit tk = Toolkit.getDefaultToolkit();
        image = tk.getImage(getURL(filename));
```

```
        while(getImage().getWidth(frame) <= 0);
        double x = frame.getSize().width/2- width()/2;
        double y = frame.getSize().height/2 - height()/2;
        at = AffineTransform.getTranslateInstance(x, y);
    }

    public void transform() {
        at.setToIdentity();
        at.translate((int)getX() + width()/2, (int)getY() + height()/2);
        at.rotate(Math.toRadians(getFaceAngle()));
        at.translate(-width()/2, -height()/2);
    }

    public void draw() {
        g2d.drawImage(getImage(), at, frame);
    }

    //bounding rectangle
    public Rectangle getBounds() {
        Rectangle r;
        r = new Rectangle((int)getX(), (int)getY(), width(), height());
        return r;
    }
}
```

Creating a Reusable Sprite Class

Following is the source code for the new `Sprite` class. This class includes a *ton* of features! In fact, it's so loaded with great stuff that you probably won't even know what to do with it all at this point. I am not a big fan of inheritance, preferring to build core functionality into each class I use. We will peruse the properties and methods of this class as we need them. This highly reusable `Sprite` class will be a useful helper class for any future game project you work on! It is a bit daunting only because I wanted to provide a complete class now rather than give it to you in parts over time. This class resulted from work done on the Galactic War game.

Collision Testing

The `Sprite` class includes several methods for detecting collisions with other sprites, and it also provides tests for collision with `Rectangle` and `Point2D` objects as a convenience. Remember that I wanted this `Sprite` class to be intuitive and not cause the compiler to complain about silly things, such as data type conversions? Well, the same is true of the collision testing code. There are three versions of the `collidesWith()` method in the `Sprite` class, providing support for three different parameters:

- `Rectangle`
- `Sprite`
- `Point2D`

This should cover almost any game object that you would like to test for a collision. Since these methods are built into the `Sprite` class, you can call them with a single parameter, and the internal data in the sprite itself is used for the second parameter that would normally be passed to a collision routine.

Sprite Class Source Code

This new `Sprite` class does not inherit from anything other than the base `Object`, although it uses `ImageEntity` internally for access to that class' excellent support for image loading and drawing. Why doesn't this class inherit from `BaseGameEntity` or `ImageEntity`? Those classes followed a logical inheritance chain but also included a lot of features that do not need to be in the core of the `Sprite` class. We still have access to those properties and methods if we want to use them, by using an `ImageEntity` as a private variable, but we get around the problem of having to deal with private/public access and inheritance. Inheritance is a beautiful concept, but in practice too much of it can make a program too complicated.

```
// Sprite class
import java.awt.*;
import javax.swing.*;

public class Sprite extends Object {
    private ImageEntity entity;
    protected Point pos;
```

```
protected Point vel;
protected double rotRate;
protected int currentState;

//constructor
Sprite(JFrame a, Graphics2D g2d) {
    entity = new ImageEntity(a);
    entity.setGraphics(g2d);
    entity.setAlive(false);
    pos = new Point(0, 0);
    vel = new Point(0, 0);
    rotRate = 0.0;
    currentState = 0;
}

//load bitmap file
public void load(String filename) {
    entity.load(filename);
}

//perform affine transformations
public void transform() {
    entity.setX(pos.x);
    entity.setY(pos.y);
    entity.transform();
}

//draw the image
public void draw() {
    entity.g2d.drawImage(entity.getImage(),entity.at,entity.frame);
}

//draw bounding rectangle around sprite
public void drawBounds(Color c) {
    entity.g2d.setColor(c);
    entity.g2d.draw(getBounds());
}

//update the position based on velocity
public void updatePosition() {
```

```
        pos.x += vel.x;
        pos.y += vel.y;
    }

    //methods related to automatic rotation factor
    public double rotationRate() { return rotRate; }
    public void setRotationRate(double rate) { rotRate = rate; }
    public void updateRotation() {
        setFaceAngle(faceAngle() + rotRate);
        if (faceAngle() < 0)
            setFaceAngle(360 - rotRate);
        else if (faceAngle() > 360)
            setFaceAngle(rotRate);
    }

    //generic sprite state variable (alive, dead, collided, etc)
    public int state() { return currentState; }
    public void setState(int state) { currentState = state; }

    //returns a bounding rectangle
    public Rectangle getBounds() { return entity.getBounds(); }

    //sprite position
    public Point position() { return pos; }
    public void setPosition(Point pos) { this.pos = pos; }

    //sprite movement velocity
    public Point velocity() { return vel; }
    public void setVelocity(Point vel) { this.vel = vel; }

    //returns the center of the sprite as a Point
    public Point center() {
        int x = (int)entity.getCenterX();
        int y = (int)entity.getCenterY();
        return(new Point(x,y));
    }

    //generic variable for selectively using sprites
    public boolean alive() { return entity.isAlive(); }
    public void setAlive(boolean alive) { entity.setAlive(alive); }
```

```
//face angle indicates which direction sprite is facing
public double faceAngle() { return entity.getFaceAngle(); }
public void setFaceAngle(double angle) {
    entity.setFaceAngle(angle);
}
public void setFaceAngle(float angle) {
    entity.setFaceAngle((double) angle);
}
public void setFaceAngle(int angle) {
    entity.setFaceAngle((double) angle);
}

//move angle indicates direction sprite is moving
public double moveAngle() { return entity.getMoveAngle(); }
public void setMoveAngle(double angle) {
    entity.setMoveAngle(angle);
}
public void setMoveAngle(float angle) {
    entity.setMoveAngle((double) angle);
}
public void setMoveAngle(int angle) {
    entity.setMoveAngle((double) angle);
}

//returns the source image width/height
public int imageWidth() { return entity.width(); }
public int imageHeight() { return entity.height(); }

//check for collision with a rectangular shape
public boolean collidesWith(Rectangle rect) {
    return (rect.intersects(getBounds()));
}
//check for collision with another sprite
public boolean collidesWith(Sprite sprite) {
    return (getBounds().intersects(sprite.getBounds()));
}
//check for collision with a point
public boolean collidesWith(Point point) {
    return (getBounds().contains(point.x, point.y));
}
```

```
    public JFrame frame() { return entity.frame; }
    public Graphics2D graphics() { return entity.g2d; }
    public Image image() { return entity.image; }
    public void setImage(Image image) { entity.setImage(image); }
}
```

Tip

Animation is a feature missing from the `Sprite` class at this point; we will go over that subject in the next chapter.

Testing the Sprite Class

Let's give the new classes we've developed in this chapter a test run. The following program (shown in Figure 6.1) draws a background image and then draws a sprite randomly on the screen. This test program uses a thread and the `Runnable` interface in order to draw a sprite repeatedly on the screen without user input. We'll study this feature more thoroughly in Chapter 10, when we learn more about threads and the game loop. Study this short demo program

Figure 6.1
The SpriteTest program demonstrates how to use the `Sprite` class.

well, because it demonstrates perhaps the first high-speed example you've seen thus far.

```
// SpriteTest program
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;
import java.util.*;
import java.net.*;

public class SpriteTest extends JFrame implements Runnable {
    int screenWidth = 640;
    int screenHeight = 480;

    //double buffer objects
    BufferedImage backbuffer;
    Graphics2D g2d;

    Sprite asteroid;
    ImageEntity background;
    Thread gameloop;
    Random rand = new Random();

    public static void main(String[] args) {
        new SpriteTest();
    }

    public SpriteTest() {
        super("Sprite Test");
        setSize(640,480);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //create the back buffer for smooth graphics
        backbuffer = new BufferedImage(screenWidth, screenHeight,
            BufferedImage.TYPE_INT_RGB);
        g2d = backbuffer.createGraphics();

        //load the background
        background = new ImageEntity(this);
        background.load("bluespace.png");
```

```
        //load the asteroid sprite
        asteroid = new Sprite(this, g2d);
        asteroid.load("asteroid2.png");

        gameloop = new Thread(this);
        gameloop.start();
    }

    public void run() {
        Thread t = Thread.currentThread();
        while (t == gameloop) {
            try {
                Thread.sleep(30);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }

            //draw the background
            g2d.drawImage(background.getImage(), 0, 0, screenWidth-1,
                screenHeight-1, this);

            int width = screenWidth - asteroid.imageWidth() - 1;
            int height = screenHeight - asteroid.imageHeight() - 1;

            Point point = new Point(rand.nextInt(width),
                rand.nextInt(height));
            asteroid.setPosition(point);
            asteroid.transform();
            asteroid.draw();

            repaint();
        }
    }

    public void paint(Graphics g) {
        //draw the back buffer to the screen
        g.drawImage(backbuffer, 0, 0, this);
    }
}
```

What You Have Learned

This significant chapter produced a monumental new version of Galactic War that is a foundation for the chapters to come. The final vestiges of the game's vector-based roots have been discarded, and the game is now fully implemented with bitmaps. In this chapter, you learned:

- How to create a new, powerful `Sprite` class
- How to detect sprite collision
- How to write reusable methods and classes

Review Questions

The following questions will help you to determine how well you have learned the subjects discussed in this chapter. The answers are provided in Appendix A, "Chapter Quiz Answers."

1. What is the name of the support class created in this chapter to help the `Sprite` class manage position and velocity?
2. During which keyboard event should you disable a keypress variable, when detecting multiple key presses with global variables?
3. What are the three types of parameters you can pass to the `collidesWith()` method?
4. What Java class provides an alternate method for loading images that is not tied to the applet?
5. Which Java package do you need to import to use the `Graphics2D` class?
6. What numeric data type does the `Point` class use for internal storage of the X and Y values?
7. What data types can the `Point` class work with at the constructor level?
8. Which sprite property determines the angle at which the sprite will move?
9. Which sprite property determines at which angle an image is pointed, regardless of movement direction?
10. Which `AffineTransform` method allows you to translate, rotate, and scale a sprite?

On Your Own

Use the following exercises to test your understanding of the material covered in this chapter.

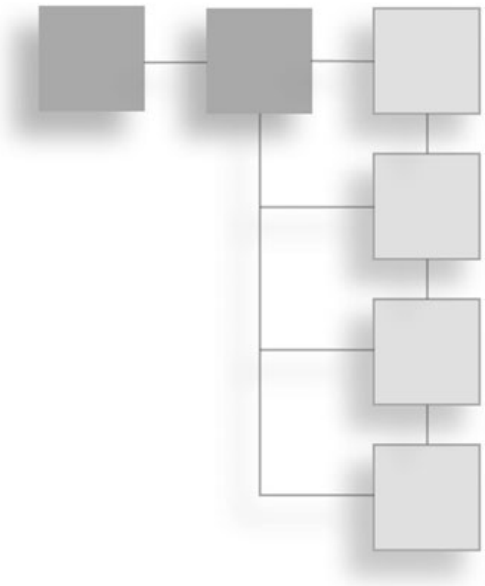
Exercise 1

The SpriteTest program demonstrates the use of the `Sprite` class. Modify the program so that it draws multiple instances of the asteroid sprite on the screen, each moving and animating differently.

Exercise 2

Modify the SpriteTest program even further by adding collision testing, such that the asteroids will rebound off one another when they collide.

Chapter 7



Sprite Animation

This chapter adds a significant new feature to your Java toolbox—the ability to load and draw animated sprites and apply that knowledge to an enhanced new sprite class. You will learn about the different ways to store a sprite animation and how to access a single frame in an animation strip, and you will see a new class called `AnimatedSprite` with some serious new functionality that greatly extends the base `Sprite` class.

Here are the key topics we'll cover in this chapter:

- Sprite animation techniques
- Drawing individual sprite frames
- Keeping track of animation frames
- Encapsulating sprite animation in a class

Sprite Animation

Over the years I have seen many techniques for sprite animation. Of the many algorithms and implementations I've studied, I believe there are two essential ways to animate a graphic object on the screen—1) Loading individual frames, each stored in its own bitmap file (in sequence); or 2) Loading a single bitmap containing rows and columns of animation frames (as tiles).

Animation Techniques

First, there is the *sequence* method. This type of animation involves loading a bitmap image for each frame of the animation in sequence, and then animating them on the screen by drawing each image in order. This technique tends to take a long time to load all of the animation frames, especially in a large game with many sprites. There is also the system overhead required to maintain so many images in memory, even if they are small. Figure 7.1 shows an example.

Drawing an animation sequence is somewhat of a challenge when loading individual frames because of the logistics of it. How should you store the images—in an array or a linked list? I've seen some implementations using both methods, and neither is very friendly, so to speak, because the code is so complicated.

The second sprite animation technique is the *tiled* method. This type of animation involves storing an entire animation sequence inside a single bitmap file, also known as an *animation strip*. Inside this bitmap file are the many frames of the animation laid out in a single row or with many columns and rows. Figure 7.2 shows an animation strip on a single row, while Figure 7.3 shows a larger animation with multiple columns and rows.

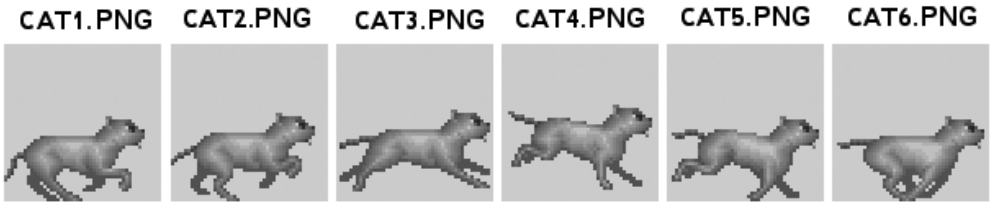


Figure 7.1
An animation sequence with frames stored in individual bitmap files.

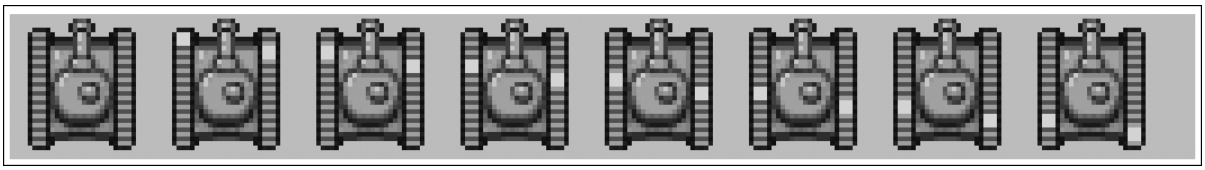


Figure 7.2
An animation strip with a single row. Courtesy of Ari Feldman.

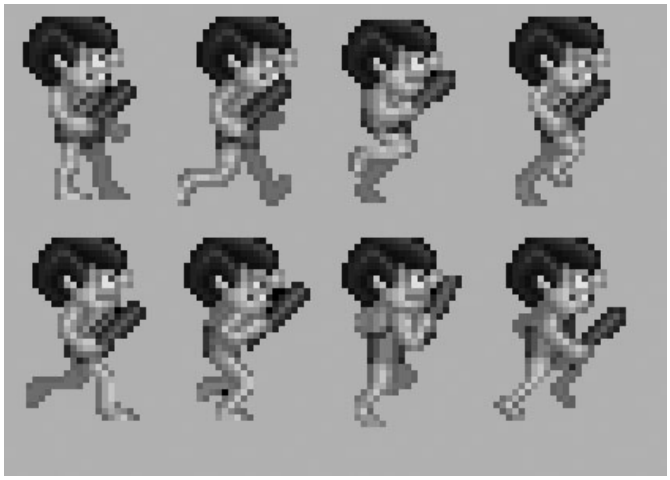


Figure 7.3
An animation strip with four columns and two rows. Courtesy of Ari Feldman.

Drawing Individual Frames

The key to drawing a single frame from an animation sequence stored in a tiled bitmap is to figure out where each frame is located *algorithmically*. It's impossible to manually code the X and Y position for each frame in the image; the very thought of it gives me hives. Not only would it take hours to jot down the X,Y position of every frame, but the bitmap file could easily be modified, thus rendering the manually calculated points irrelevant. This is computer science, after all, so there is an algorithm for almost everything.

You can calculate the column (that is, the number of frames *across*) by dividing the frame number by the number of columns and multiplying that by the height of each frame. This calculation focuses on the *quotient* as the answer we want.

```
frameY = (frameNumber / columns) * height;
```

This will give you the correct *row* down into the image where your desired frame is located, but it will not provide you with the actual *column*, or X value. For that, you need a similar solution. Instead of dividing the frame number by columns, we will use *modulus*. This calculation focuses on the *remainder* as the answer we want.

```
frameX = (frameNumber % columns) * width;
```

As you might have noticed, this looks almost exactly like the formula for calculating `frameY`. Now we're multiplying by width and using the *modulus*

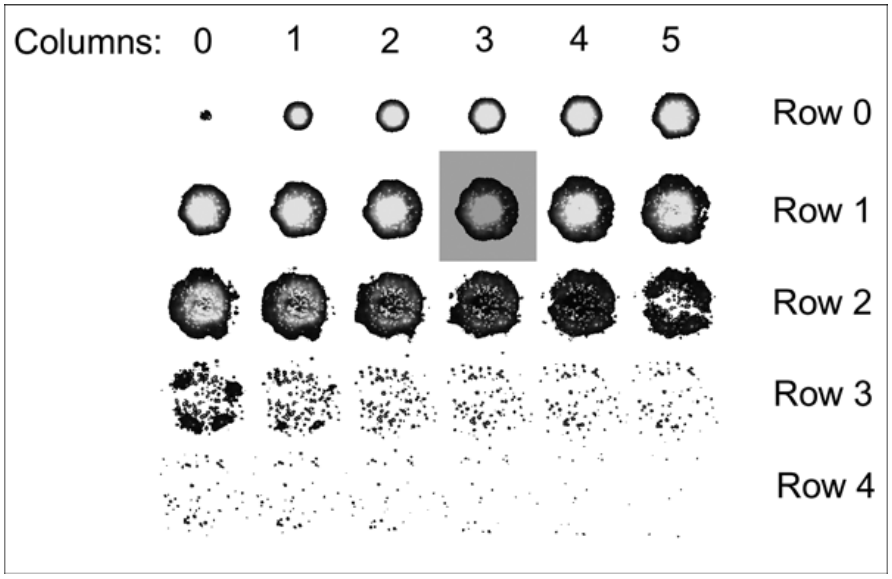


Figure 7.4
Illustration of a specific frame in the sprite sheet.

character instead of the *division* character. Modulus returns the *remainder* of a division, rather than the quotient itself. If you want the Y value, you look at the division *quotient*; if you want the X value, you look at the division *remainder*. Figure 7.4 illustrates how a desired frame is at a certain column and row position in the sprite sheet. See if you can use the division and modulus calculations to figure out where any random frame is located on the sheet on your own!

Here is a complete method that draws a single frame out of an animation sequence. There are a lot of parameters in this method! Fortunately, they are all clearly labeled with descriptive names. It's obvious that we pass it the source `Image`, the destination `Graphics2D` object (which does the real drawing), the destination location (X and Y), the number of columns across, the frame number you want to draw, and then the width and height of a single frame. What you get in return is the desired animation frame on the destination surface (which can be your back buffer or the applet window).

```
public void drawFrame(Image source, Graphics2D dest,
    int destX, int destY, int cols, int frame, int width, int height)
{
    int frameX = (frame % cols) * width;
    int frameY = (frame / cols) * height;
```

```
    dest.drawImage(source, destX, destY, destX+width, destY+height,
        frameX, frameY, frameX+width, frameY+height, this);
}
```

Keeping Track of Animation Frames

Acquiring the desired animation frame is just the first step toward building an animated sprite in Java. After you have figured out how to grab a single frame, you must then decide what to do with it! For instance, how do you tell your program which frame to draw, and how does the program update the current frame each time through the game loop? I've found that the easiest way to do this is with a simple update method that increments the animation frame and then tests it against the bounds of the animation sequence. For instance:

```
currentFrame += 1;
if (currentFrame > 7) {
    currentFrame = 0;
}
else if (currentFrame < 0) {
    currentFrame = 7;
}
```

Take a close look at what's going on in the code here. First, the current frame is incremented by the value 1. To animate in the reverse order, this would be -1. Then, the next line checks the upper boundary (7) and loops back to 0 if the boundary is crossed. Similarly, the lower boundary is checked, setting `currentFrame` to the upper boundary value if necessary. Making this code reusable, we would need three variables:

- `currentFrame`
- `totalFrames`
- `animationDirection`

You would want to call this update code from the thread's `run()` event method. But, speaking of the thread, that does bring up an important issue—timing. Obviously, you don't want every sprite in the game to animate at exactly the same rate! Some sprites will move very slowly, while others will have fast animations. This is really an issue of fine-tuning the gameplay, but you must have some sort of mechanism in place for implementing timing for each animated sprite *separately*.

You can accomplish this by adding a couple more variables to the mix. First, you will need to increment a counter each time through the game loop. If that counter reaches a certain threshold value, then you reset the counter and go through the process of updating the animation frame as before. Let's use variables called `frameCount` and `frameDelay`. The frame delay is usually a smaller value than you would expect—such as 5 to 10, but usually not much more. A delay of 10 in a game loop running at 50 fps means that the object only animates at 5 fps, which is very slow indeed. I often use values of 1 to 5 for the frame delay. Here is the updated animation code with a delay in place:

```
frameCount++;
if (frameCount > frameDelay) {
    frameCount=0;
    currentFrame += animationDirection;
    if (currentFrame > totalFrames-1) {
        currentFrame = 0;
    }
    else if (currentFrame < 0) {
        currentFrame = totalFrames-1;
    }
}
```

The end result is a much *simplified* form of timed animation that assumes the update is taking place within a certain timed function already. If we did not call on this animation code from inside an already-timed function, then the animation would go too fast, and we would need to insert built-in timing into every sprite. We can get away with somewhat *lazy* timing code like this as long as we can assume timing is already handled.

Testing Sprite Animation

I'd like to go through a complete example with you so these concepts will feel more real to you, and so that you can see the dramatic result when a sprite is animated. The AnimationTest program loads a sprite sheet containing 30 frames of an explosion animation (shown in Figure 7.5) and animates it on the screen. Since we are sticking to the subject of animation in this chapter, the program doesn't attempt to do any transforms, such as rotation. But can you imagine the result of an animated sprite that can *also* be rotated? This program will help to determine what we need to do in the animation class coming up next.

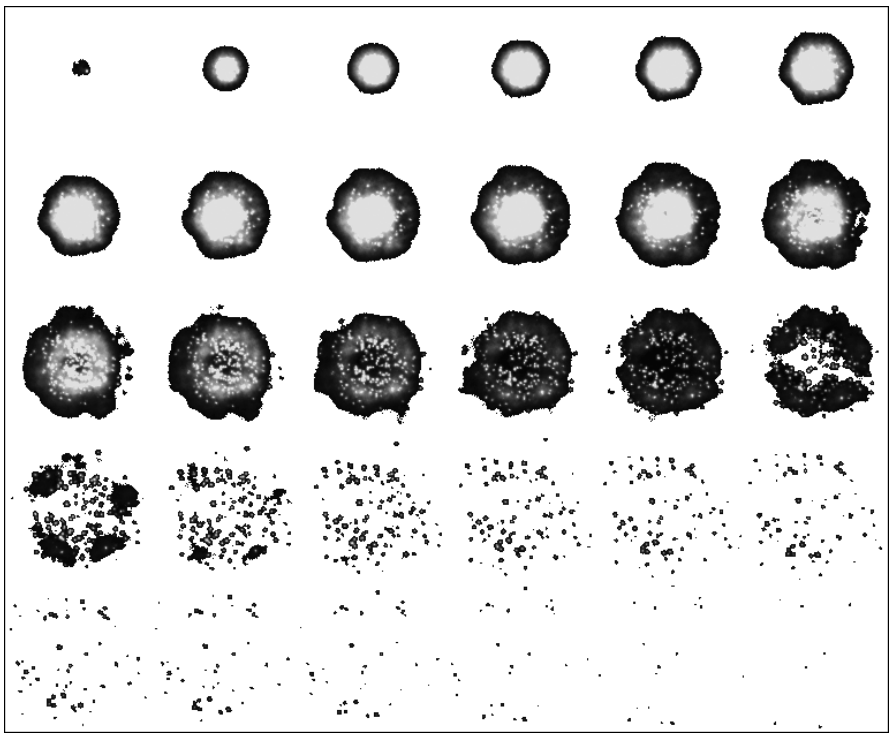


Figure 7.5
An animated explosion with 30 frames.

The output from the program is shown in Figure 7.6, where the single animated sprite is being drawn over a background image. Following is the code listing for the AnimationTest program. I have highlighted key portions of code that are new to this chapter in bold text.

```
// AnimationTest program
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.image.*;
import java.net.*;

public class AnimationTest extends JFrame implements Runnable {
    static int ScreenWidth = 640;
    static int ScreenHeight = 480;
    Thread gameloop;
    Random rand = new Random();
```

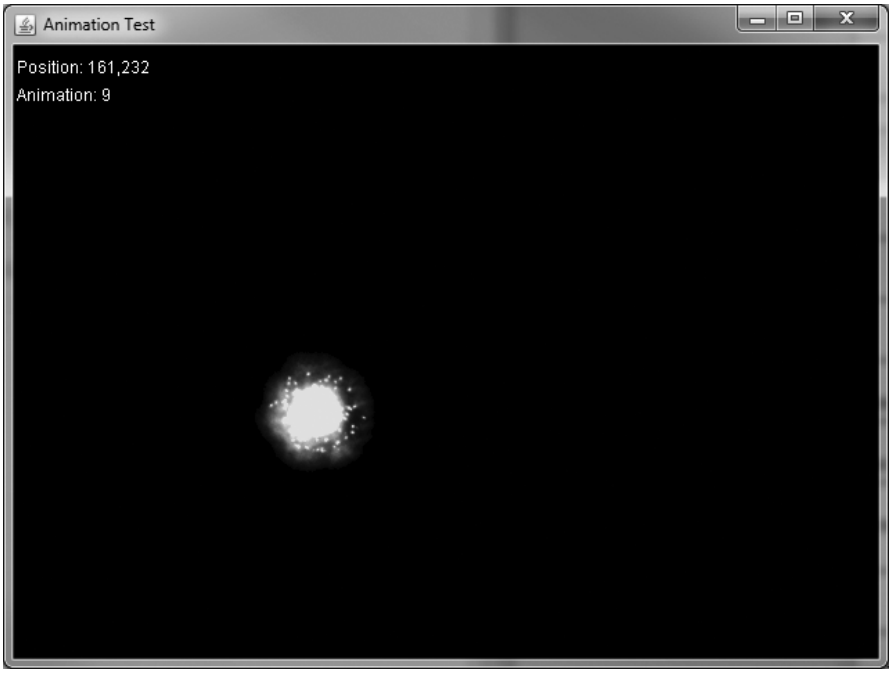


Figure 7.6
The AnimationTest program.

```
//double buffer objects
BufferedImage backbuffer;
Graphics2D g2d;

//sprite variables
Image image;
Point pos = new Point(300,200);

//animation variables
int currentFrame = 0;
int totalFrames = 30;
int animationDirection = 1;
int frameCount = 0;
int frameDelay = 10;

public static void main(String[] args) {
    new AnimationTest();
}
```

```
public AnimationTest() {
    super("Animation Test");
    setSize(ScreenWidth,ScreenHeight);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //create the back buffer for smooth graphics
    backbuffer = new BufferedImage(ScreenWidth, ScreenHeight,
        BufferedImage.TYPE_INT_RGB);
    g2d = backbuffer.createGraphics();

    //load the ball animation strip
    Toolkit tk = Toolkit.getDefaultToolkit();
    image = tk.getImage(getURL("explosion.png"));

    gameloop = new Thread(this);
    gameloop.start();
}

private URL getURL(String filename) {
    URL url = null;
    try {
        url = this.getClass().getResource(filename);
    }
    catch (Exception e) {}
    return url;
}

public void run() {
    Thread t = Thread.currentThread();
    while (t == gameloop) {
        try {
            Thread.sleep(5);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        gameUpdate();
    }
}
```

```
public void gameUpdate() {
    //clear the background
    g2d.setColor(Color.BLACK);
    g2d.fill( new Rectangle(0, 0, ScreenWidth-1, ScreenHeight-1) );

    //draw the current frame of animation
    drawFrame(image, g2d, pos.x, pos.y, 6, currentFrame, 128, 128);

    g2d.setColor(Color.WHITE);
    g2d.drawString("Position: " + pos.x + "," + pos.y, 10, 50);
    g2d.drawString("Animation: " + currentFrame, 10, 70);

    //see if it's time to animate
    frameCount++;
    if (frameCount > frameDelay) {
        frameCount=0;
        //update the animation frame
        currentFrame += animationDirection;
        if (currentFrame > totalFrames - 1) {
            currentFrame = 0;
            pos.x = rand.nextInt(ScreenWidth-128);
            pos.y = rand.nextInt(ScreenHeight-128);
        }
        else if (currentFrame < 0) {
            currentFrame = totalFrames - 1;
        }
    }
    repaint();
}

public void paint(Graphics g) {
    //draw the back buffer to the screen
    g.drawImage(backbuffer, 0, 0, this);
}

//draw a single frame of animation
public void drawFrame(Image source, Graphics2D dest,
    int x, int y, int cols, int frame, int width, int height)
{
    int fx = (frame % cols) * width;
```

```
            int fy = (frame / cols) * height;
            dest.drawImage(source, x, y, x+width, y+height,
                fx, fy, fx+width, fy+height, this);
        }
}
```

Now, after reviewing this code, you might be wondering why we aren't using the `ImageEntity` and `Sprite` classes from the previous chapter, since they would cut down on so much of this code. That's a good question! While learning how to do animation, a single, self-contained example is helpful before we get into a class. Coming up next, we will do that.

Encapsulating Sprite Animation in a Class

There are some significant new pieces of code in the AnimationTest program that we'll definitely need for the upcoming Galactic War project (in Part III). All of the properties can be *stuffed* (that's slang for *encapsulated* or *wrapped*) into a class, and we can reuse that beautiful `drawFrame()` method as well. One really great thing about moving `drawFrame()` into a class is that most of the parameters can be eliminated, as they will be pulled out of the class internally. Setting up an animation will require a few steps up front when the game starts up, but after that, drawing an animated sprite will be an automatic process with just one or two method calls.

This new `AnimatedSprite` class will be completely self-contained. Now that we've seen how inheritance works and how useful it is for reusing code, and it works well for the existing `Sprite` class, we don't need to continue adding new levels to drive the point home. At this time, we will condense everything into just one class to cut down on any confusion that may arise as a result of using the three classes that have been written up to this point: `BaseGameEntity`, `ImageEntity`, and `Sprite`. The properties and methods in these three will be combined into the single `AnimatedSprite` class.

To improve performance, the `AnimatedSprite` class will *not* support affine transforms! This is because of a limitation in the `Graphics2D.drawImage()` function, which can either do animation *or* a transform, but not both in the same function call. So, there are two choices and we can only make one without writing a ton of code: 1) We can draw the current animation frame onto a scratch image and then apply the transforms to it before drawing it; or 2) We

can draw frames of animation directly, but without the benefit of transforms. Since the `Sprite` class in the previous chapter works already with transforms, a good compromise is this: If you want transforms, use `Sprite`; otherwise, if you need animation, use `AnimatedSprite` (but without transforms). A compromise certainly could involve rendering each frame to a scratch image and then applying transforms to it, and perhaps more advanced sprite code would do just that.

Tip

The `Sprite` class supports transforms (rotation and scaling) and manual animation. The `AnimatedSprite` class does automatic animation but *cannot* do any transforms.

Here's the new source code listing for the `AnimatedSprite` class, which is completely self-contained. Probably the most obvious thing about this class is that most variables are declared as `public`, which exposes them to any program that uses the class without any `get` or `set` methods. In a game project, often those `get` and `set` methods just hurt productivity. The important thing is that the class works and is versatile, with variables that are used by the embedded methods in the class. This is not pure object-oriented programming (OOP) by any means—we give up some security for versatility and just count on programmers who use the class to know what they're doing. Getting the job done while writing good, clean code is often the rule in a programming team!

```
// AnimatedSprite class
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import javax.swing.*;
import java.net.*;

public class AnimatedSprite {
    protected JFrame frame;
    protected Graphics2D g2d;
    public Image image;
    public boolean alive;
    public Point position;
    public Point velocity;
    public double rotationRate;
```

```
public int currentState;
public int currentFrame, totalFrames;
public int animationDirection;
public int frameCount, frameDelay;
public int frameWidth, frameHeight, columns;
public double moveAngle, faceAngle;

public AnimatedSprite(JFrame _frame, Graphics2D _g2d) {
    frame = _frame;
    g2d = _g2d;
    image = null;
    alive = true;
    position = new Point(0, 0);
    velocity = new Point(0, 0);
    rotationRate = 0.0;
    currentState = 0;
    currentFrame = 0;
    totalFrames = 1;
    animationDirection = 1;
    frameCount = 0;
    frameDelay = 0;
    frameWidth = 0;
    frameHeight = 0;
    columns = 1;
    moveAngle = 0.0;
    faceAngle = 0.0;
}

public JFrame getJFrame() { return frame; }
public Graphics2D getGraphics() { return g2d; }
public void setGraphics(Graphics2D _g2d) { g2d = _g2d; }

public void setImage(Image _image) { image = _image; }

public int getWidth() {
    if (image != null)
        return image.getWidth(frame);
    else
        return 0;
}
```

```
public int getHeight() {
    if (image != null)
        return image.getHeight(frame);
    else
        return 0;
}

public double getCenterX() {
    return position.x + getWidth() / 2;
}
public double getCenterY() {
    return position.y + getHeight() / 2;
}
public Point getCenter() {
    int x = (int)getCenterX();
    int y = (int)getCenterY();
    return(new Point(x,y));
}

private URL getURL(String filename) {
    URL url = null;
    try {
        url = this.getClass().getResource(filename);
    }
    catch (Exception e) { }
    return url;
}

public Rectangle getBounds() {
    return (new Rectangle((int)position.x, (int)position.y, _
        getWidth(), getHeight()));
}

public void load(String filename, int _columns, int _totalFrames,
    int _width, int _height)
{
    Toolkit tk = Toolkit.getDefaultToolkit();
    image = tk.getImage(getURL(filename));
    while(image.getWidth(frame) <= 0);
    columns = _columns;
```

```
    totalFrames = _totalFrames;
    frameWidth = _width;
    frameHeight = _height;
}

protected void update() {
    //update position
    position.x += velocity.x;
    position.y += velocity.y;

    //update rotation
    if (rotationRate > 0.0) {
        faceAngle += rotationRate;
        if (faceAngle < 0)
            faceAngle = 360 - rotationRate;
        else if (faceAngle > 360)
            faceAngle = rotationRate;
    }

    //update animation
    if (totalFrames > 1) {
        frameCount++;
        if (frameCount > frameDelay) {
            frameCount = 0;
            currentFrame += animationDirection;
            if (currentFrame > totalFrames - 1) {
                currentFrame = 0;
            }
            else if (currentFrame < 0) {
                currentFrame = totalFrames - 1;
            }
        }
    }
}

//draw bounding rectangle around sprite
public void drawBounds(Color c) {
    g2d.setColor(c);
    g2d.draw(getBounds());
}
```

```
    public void draw() {
        update();
        //get the current frame
        int frameX = (currentFrame % columns) * frameWidth;
        int frameY = (currentFrame / columns) * frameHeight;
        //draw the frame
        g2d.drawImage(image, position.x, position.y,
            position.x+frameWidth, position.y+frameHeight,
            frameX, frameY, frameX+frameWidth, frameY+frameHeight,
            getJFrame());
    }

    //check for collision with a rectangular shape
    public boolean collidesWith(Rectangle rect) {
        return (rect.intersects(getBounds()));
    }
    //check for collision with another sprite
    public boolean collidesWith(AnimatedSprite sprite) {
        return (getBounds().intersects(sprite.getBounds()));
    }
    //check for collision with a point
    public boolean collidesWith(Point point) {
        return (getBounds().contains(point.x, point.y));
    }
}
```

Testing the New AnimatedSprite Class

Figure 7.7 shows the output of the program, which you can open up and run from the chapter's resource files if you wish (www.courseptr.com/downloads). Thanks to our new `AnimatedSprite` class, the source code here is quite short compared to previous sprite projects!

```
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.image.*;
import java.net.*;

public class AnimationClassDemo extends JFrame implements Runnable {
    static int ScreenWidth = 640;
```

Figure 7.7
Testing the `AnimatedSprite` class.

```
static int ScreenHeight = 480;
Thread gameloop;
Random rand = new Random();

//double buffer objects
BufferedImage backbuffer;
Graphics2D g2d;

//sprite variables
AnimatedSprite sprite;

public static void main(String[] args) {
    new AnimationClassDemo();
}

public AnimationClassDemo() {
    super("Animation Class Demo");
    setSize(ScreenWidth,ScreenHeight);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        //create the back buffer for smooth graphics
        backbuffer = new BufferedImage(ScreenWidth, ScreenHeight,
            BufferedImage.TYPE_INT_RGB);
        g2d = backbuffer.createGraphics();

        //load the explosion animation
        sprite = new AnimatedSprite(this, g2d);
        sprite.load("explosion.png", 6, 5, 128, 128);
        sprite.position = new Point(300,200);
        sprite.frameDelay = 10;
        sprite.totalFrames = 30;
        sprite.velocity = new Point(1,1);
        sprite.rotationRate = 1.0;

        gameloop = new Thread(this);
        gameloop.start();
    }

    public void run() {
        Thread t = Thread.currentThread();
        while (t == gameloop) {
            try { Thread.sleep(5); }
            catch (InterruptedException e)
            { e.printStackTrace(); }
            gameUpdate();
        }
    }

    public void gameUpdate() {
        //draw the background
        g2d.setColor(Color.BLACK);
        g2d.fill( new Rectangle(0, 0, ScreenWidth-1, ScreenHeight-1) );

        //draw the sprite
        sprite.draw();

        //keep the sprite in the screen boundary
        if (sprite.position.x < 0 || sprite.position.x>ScreenWidth-128)
            sprite.velocity.x *= -1;
        if (sprite.position.y < 0 || sprite.position.y>ScreenHeight-128)
            sprite.velocity.y *= -1;
```

```
            g2d.setColor(Color.WHITE);
            g2d.drawString("Position: " + sprite.position.x + "," +
                sprite.position.y, 10, 40);
            g2d.drawString("Velocity: " + sprite.velocity.x + "," +
                sprite.velocity.y, 10, 60);
            g2d.drawString("Animation: " + sprite.currentFrame, 10, 80);

            repaint();
        }

        public void paint(Graphics g) {
            //draw the back buffer to the screen
            g.drawImage(backbuffer, 0, 0, this);
        }
}
```

What You Have Learned

This chapter tackled the difficult subject of sprite animation. Adding support for animation is not an easy undertaking, but this chapter provided you with the knowledge and a new class called `AnimatedSprite` that will make it possible for you to write your own games without reinventing the wheel every time you need to load an image and draw it. Here are the key topics you learned:

- How an animation is stored in a bitmap file
- How to load and draw an animation strip from a single bitmap file
- How to animate a sprite with timing
- How to put it all together into a reusable class

Review Questions

The following questions will help you to determine how well you have learned the subjects discussed in this chapter. The answers are provided in Appendix A, "Chapter Quiz Answers."

1. What is the name of the animation class created in this chapter?
2. From which class does the new animation class inherit?
3. How many frames of animation were there in the animated ball sprite?