Advanced Java Game Programming

DAVID WALLACE CROFT

Apress™

Advanced Java Game Programming

Copyright © 2004 by David Wallace Croft

ISBN (pbk): 1-59059-123-2

Printed and bound in the United States of America 10987654321

Lead Editor: Steve Anglin

Technical Reviewer: Jack Park

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Sofia Marchant

Copy Manager: Nicole LeClerc

Production Manager: Kari Brooks

Production Editor: Ellie Fountain

Compositor: Gina Rexrode, Point n' Click Publishing, LLC

Proofreader: Susannah Pfalzer

Indexer: Brenda Miller

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 1

Development Setup

A good example is the best sermon. —Benjamin Franklin

In this chapter, you will learn how to configure your development environment to create your own games using the game library and examples described in this book.

Upgrading to 1.4

To compile and run the example games in this book, you need to download and install the Java 2 Software Development Kit (SDK), Standard Edition (J2SE), version 1.4 or later. "Java 2" refers to the Java 2 Platform. "Java 1.4" refers to the specific version of the Java 2 Platform. You can blame the marketing folks at Sun Microsystems for the confusion.

Version 1.4 is required because the game code depends upon new accelerated graphics classes in that version of the J2SE. These new classes greatly increase animation performance. At the time of this writing, J2SE v1.4 is available on multiple platforms including Linux, Mac OS X, Solaris, and Windows. If you do not have version 1.4 or later installed, you can download it from `http://java.sun.com/` for the most common platforms besides Apple. Apple developers can download the latest version of Java from `http://developer.apple.com/java/`.

The game code should run without modification equally well on all platforms that support J2SE v1.4. Having said that, I must now warn you that I have only tested the code on Linux and Windows. I will depend upon the feedback of readers with Mac OS X and Solaris systems to let me know how the code fares on those platforms.

Sticking to the Core

The J2SE defines a standardized application programming interface (API) library with thousands of classes grouped into scores of packages that are considered *core*. The core classes are guaranteed to be pre-installed on any J2SE-compatible system. This makes the life of a developer much easier because it means less code that needs to be downloaded and installed to the client platform. An example of a core package is `java.lang`, which contains the basic classes needed in most Java programs.

Noncore classes are not guaranteed to be pre-installed on the target platform, so you will probably want to think twice before using them. These non-core classes are known as *optional packages*. Until recently, they were called *standard extensions*. They are "extensions" in that they are not distributed with the core, and they are "standard" in that they define standardized interfaces for additional capabilities independent of their underlying implementation. An example of an optional package is `javax.media.j3d`, a library for rendering a 3D scene graph.

It used to be that you could easily tell which packages were core and which were standard extensions because the core package names all started with the `java` prefix and the standard extension names all started with the `javax` prefix. This is no longer the case as the core now contains packages that start with names other than `java` such as `javax.swing`. I am not sure why Sun Microsystems changed this but I suspect it had something to do with the Microsoft lawsuit. I know that the switch continued to confuse even high-level Sun engineers for some time after the decision was made. At some point they will have to give the Java library a complete overhaul. Perhaps they will call it the Java 3 Platform, Version 2.0. When they do so, I hope they consider changing the package naming convention back to the way it was.

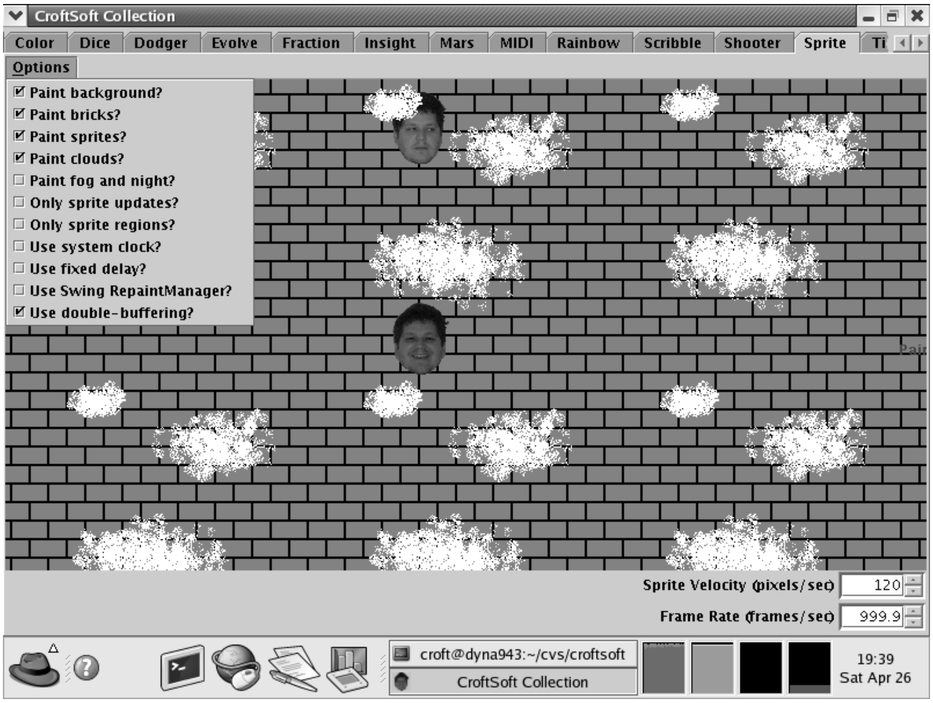
Most of the example game code relies upon the core classes exclusively. This ensures that they will run without additional installation steps on all J2SE platforms. A few of the games, however, do rely upon the optional packages. In these cases, I will warn you.

The subset of the J2SE for smaller platforms such as embedded devices is the Java 2 Micro Edition (J2ME). The superset for heavy-duty platforms such as application servers is the Java 2 Enterprise Edition (J2EE). This book does not cover J2ME and only covers J2EE lightly. Where you need to use J2EE code, I will forewarn you just as I will when you use optional packages.

Playing the Demo Online

The URL for the book web site is `http://www.croftsoft.com/library/books/ajgp/`. At that site, you can test the demonstration as pre-compiled and installed online to get a feel for what is available. From time to time, I will update the demo. Check the book web site for new releases and subscribe to the mailing list for notifications of updates.

The demo contains several animated Java programs including games and simulations. You can scroll the tabs at the top of the screen to the left and right to see the entire list. Of special interest is Sprite, a sprite animation test program. As shown in Figure 1-1, you can change the options and parameters to see the different effects on animation performance.



- ☑ Paint background?
- ☑ Paint bricks?
- ☑ Paint sprites?
- ☑ Paint clouds?
- ☐ Paint fog and night?
- ☐ Only sprite updates?
- ☐ Only sprite regions?
- ☐ Use system clock?
- ☐ Use fixed delay?
- ☐ Use Swing RepaintManager?
- ☑ Use double-buffering?

Sprite Velocity (pixels/sec) 120

Frame Rate (frames/sec) 999.9

Figure 1-1. The Sprite demo

Exploring the Game Library

You can download a snapshot of the source code, graphics, and audio files for the game library as a compressed archive file from the book web site. This snapshot of the code captures the version that is documented in this book. Post-publication archives might also be available. You can also download the most recent development version of the code by using a CVS client. Please see Appendix B for more information.

Once downloaded and decompressed, you can compile and run the example games from the source code. The following sections describe the directory structure of the source code library.

Directory croftsoft

The top-level directory within the archive is `croftsoft/`. It contains the main build file, the `readme.txt` file, and the library subdirectories. The following list describes the subdirectory structure of the library in detail. Some of these subdirectories are not included in the zip archive but are created when you compile the examples.

- `arc/`—packaged archives
- `bin/`—binaries and utilities
- `doc/`—javadoc documentation
- `ext/`—extension libraries
- `lib/`—compiled class library
- `lic/`—licenses
- `res/`—resource files
- `src/`—source code
- `tmp/`—temporary directory

Directory arc

The executable Java archive (JAR) and web application archive (WAR) files are placed in this directory when you package your code for distribution as part of the build process. Creating JAR files is described in greater detail in the next chapter.

Directory bin

The `bin/` subdirectory is a place for your development batch files, scripts, and utilities. My understanding is that *bin* is short for binaries. This is where you would place your compiled binary utility files versus your source code text files. I could be wrong about the origin of the name of the `bin` directory: It might simply stand for bin, a place to put things.

```
path=%path%;C:\home\croft\cvs\croftsoft\bin
```

You want to append this directory to your path environment variable as shown in the preceding code. This directory contains the batch file `javainit.bat` that initializes your development environment. This is a useful batch file to run when you start working.

```
subst J: C:\home\croft\cvs\croftsoft
subst K: C:\home\croft\cvs\croftsoft\src\com\croftsoft\apps
```

When working in Windows, I use this to substitute drive letter `J:` for my working directory. That way my other batch and build files are isolated from changes to the name of my working directory because they use a path such as `J:\src` instead of `C:\home\croft\cvs\croftsoft\src`. I use drive letter `K:` as a shortcut to the application source directory where I do most of my game development.

You can also use `javainit.bat` to initialize environment variables you might need during the development process. On my machine, I have a number of environment variables defined for supplying applications and utilities with directory names such as `JAVA_HOME`, `J2EE_HOME`, `JBOSS_HOME`, and so on. Note that you do not need to define any environment variables at this time to compile or run the examples.

```
javac -deprecation -d J:\lib -sourcepath J:\src -classpath J:\lib;[...] %1
```

The `bin/` directory also contains the batch file `jc.bat` which executes the preceding compile command. I have omitted some of the classpath values here to keep it short. The `%1` is the first argument to the command line and represents the Java source code file name to be compiled. If you want to compile everything in the current directory, you can just type in `jc *.java`.

Directory doc

The `doc/` directory contains the javadoc documentation for your source code. It is not a location for your manually created project documentation as you will want to flush this directory occasionally to get rid of javadoc documentation for deleted classes without fear that you might be getting rid of something that is hard to replace. Because the javadoc utility generates the contents of this directory automatically from the source code, you should not directly edit nor add these files to version control.

Directory ext

The `ext/` directory is a place for the optional package and extension library JAR files that are required for your build. Before JAR files, developers used zip files to archive classes so you occasionally see a `.zip` file name extension in this directory as well.

```
-classpath J:\lib;J:\ext\j2ee.jar;J:\ext\javaws.jar;[...]
```

One of the reasons that I like to collect the extension JAR files in this directory instead of simply using them where they were installed originally is that it makes for a more readable classpath. I would rather use a short argument such as `J:\ext\javaws.jar` instead of a long one such as `C:\Program Files\Java Web Start\javaws.jar`. Also, sometimes space characters in a directory name such as `Program Files` confuse applications that use it in the classpath.

Finally, and probably most importantly, is that it consolidates all of your extension libraries for your project where it is easy to put them under version control. It seems a bit odd to put a third-party binary file under version control when you can usually just download another copy from the third-party web site whenever you need it. This is unreliable, however, as there might be dependencies in the code that require you to use an older version of a library that is no longer distributed from the vendor site. When your `ext/` directory is under version control, you have a snapshot of the versions of all the different extension libraries that are known to work reliably with your project. It also makes it easier on your fellow developers, including you if you ever need to move to another development machine. You can pull the JAR files from the version control system instead of downloading them all over again from multiple vendor web sites.

Directory lib

The `lib/` directory is the destination directory for your compiled Java classes: The files that end with the `.class` file name extension. This is the `-classpath` argument to your `javac` compile command and the `-cp` argument to the `java` execution command.

The abbreviation *lib* stands for library. Keep in mind, however, that this directory is only for the compiled counterparts to the source code files in the `src/` directory. You should keep your third-party libraries in a different directory such as `ext/` as you want to be able to flush the `lib/` directory to get rid of outdated versions of the compiled classes without fear that you will be deleting something hard to replace. Because all the files in the `lib/` directory are derived from the `src/` directory, there is no reason to place them under version control.

Directory lic

The `lic/` directory contains copies of the Open Source software licenses. This is described in more detail later in this chapter.

Directory res

You use the `res/` directory to keep the *resource* files you need to complete your build separate from the Java source code files. It includes JAR manifests, WAR configuration files, multimedia resources such as audio and graphics, and static data files.

Each source code file has an appropriate position within the `src/` directory hierarchy based upon its package prefix. This is not true with other types of files used in the build. Rather than putting those files in the `src/` root directory or somewhere below, such as a directory that corresponds to the package that contains the main class for the application, I just keep them in a separate directory altogether, the `res/` directory.

Directory src

The `src/` directory contains the uncompiled Java source code: the files with the `.java` file name extension. This is also a place for your Hypertext Markup Language (HTML) javadoc descriptions. The javadoc utility looks for the file `package.html` in each directory in the source path and assumes it contains a description of the classes in the package that corresponds to the directory level.

Directory tmp

The `tmp/` directory is created during the build process to hold working files temporarily. The directory is usually destroyed when the build ends. If your build is interrupted due to a compile error, this directory might not get removed. It does not hurt to leave this directory in existence if you find it, as the next successful build first deletes it automatically, creates it anew, then deletes it again. If you do decide to delete it manually, the only restriction is that you do not do it while a build is running.

Introducing XML

To build, package, and deploy your games, you need to understand the basics of Extensible Markup Language (XML). XML is also useful for storing game data and user preferences on the hard drive and passing multi-player messages over the network. This section introduces XML briefly.

XML allows you to describe data using human-readable text. In the following example, note how each data value is encapsulated in an opening and closing element tag pair that names the data field.

```
<book>
  <title>Advanced Java Game Programming</title>
  <author>David Wallace Croft</author>
  <publisher>Apress</publisher>
  <pubdate>2004</pubdate>
</book>
```

Those familiar with HTML will recognize the similarities. It looks like HTML except that I have created my own element names such as `book` and `title`. HTML already defines the element `title`, but here I have given it a new semantic, or meaning, that differs within the context of my `book` definition. Indeed, this ability to define new elements is why XML is considered "extensible."

Despite the flexibility in being able to define new element names, there are enough constraints in the XML format that parsers do not need to be customized uniquely for each definition. Such restrictions include a strict hierarchical nesting of elements. For example, the following XML syntax with overlapping tag boundaries would generate a parsing error.

```
<b><i>Illegal XML</b></i>
```

Although the preceding code might be valid in HTML, the proper way to nest the elements in Extensible Hypertext Markup Language (XHTML)—a definition of XML that replaces HTML—would be as follows.

```
<i><b>Legal XML</b></i>
```

Hundreds, perhaps thousands, of XML definitions now cover everything from genealogy to electronic commerce. Wherever data needs to be exchanged in a transparent manner using standard parsers that are readily available in all the major programming languages, XML provides a solution.

Further information on the subject of XML is easily accessible from a large number of sources as XML has rapidly become a widely adopted technology. As a quickly digestible introduction and handy reference book, I recommend the *XML Pocket Reference*, 2nd edition by Robert Eckstein (O'Reilly & Associates, 2001).

Compiling with Ant

Ant is the Open Source tool that you will use for compiling the example game source code. Ant is more powerful than older tools such as `make` as it is extensible and cross-platform. Although some warn against it, you might want to consider using Ant as a universal scripting language to replace your platform-specific batch or script files. If it lacks a function you need, Ant happily allows you to integrate any Java code you want to write to increase its capabilities.

Ant is distributed from the Apache Jakarta Project web site and has gained rapid and almost universal acceptance by the Java community. If you have not already learned how to use Ant, you should consider doing so. You can start by reading the online documentation at `http://ant.apache.org/`.

```
<project name="myproject" default="compile">
  <target name="compile">
    [...]
  </target>
  <target name="archive" depends="compile">
    [...]
  </target>
</project>
```

Ant takes its instructions on how to compile the source code from an XML build file with a default name of `build.xml`. A build file is organized as a *project* with multiple *targets*. Each target contains zero or more *tasks* that are the commands executed by the Ant processor.

The tasks in a target are usually related in a set that might depend upon the successful completion of one or more previous targets. For example, I might have a target called `archive` that archives the latest version of my source code and moves it to a backup directory. The `archive` target might depend upon another target called `compile` that attempts to compile the entire library. If I run Ant on my project build file with the target `archive` specified as a command-line argument, Ant first attempts to run the `compile` target. If that step fails, it terminates operations without proceeding to `archive`.

You can compile and run most of the example game source code by running Ant on the `build.xml` file in the library root directory using the default target. The following is a line-by-line review of the first part of the file that you use to build the main demonstration program. This is the build code that you could adapt to compile and package your own games.

```
    <project name="croftsoft" default="demo">
```

If this build file is executed without specifying a target as a command-line argument, Ant assumes the default `demo` target.

```
    <property name="arc_dir" value="arc"/>
    <property name="res_dir" value="res"/>
    <property name="src_dir" value="src"/>
    <property name="tmp_dir" value="tmp"/>
```

To prevent the directory names from being hard-coded in the rest of the build file, define them here using property variables. Once a property value is defined, it cannot be redefined later in the build file. You might need to adjust these directory names to match your own preference.

```
<target name="init">
  <mkdir  dir="${arc_dir}"/>
  <delete dir="${tmp_dir}"/>
  <mkdir  dir="${tmp_dir}"/>
  <tstamp>
    <format property="TODAY_ISO" pattern="yyyy-MM-dd"/>
    <format property="YEAR"      pattern="yyyy"/>
  </tstamp>
</target>
```

The default target `demo` indirectly depends on target `init` so I will review the `init` target first. It starts by creating the output and temporary working directories. If the output directory `arc_dir` already exists, the build file presses on without throwing an error. The temporary directory is re-created to make sure no old files are left behind from a previous build attempt.

Task `tstamp` sets a property called `TODAY_ISO` to a value representing the current date in International Standards Organization (ISO) format such as 1999-12-31. Some of the following targets use this property to append the date to archive file names.

```
<target name="shooter_prep" depends="init">
  <copy todir="${tmp_dir}/media/shooter">
    <fileset dir="${res_dir}/apps/shooter/media">
      <include name="shooter_bang.png"/>
      <include name="shooter_boom.png"/>
      <include name="shooter_rest.png"/>
      <include name="bang.wav"/>
      <include name="explode.wav"/>
    </fileset>
  </copy>
</target>
```

In preparation for compiling the demo, the resource files for a game are copied from the resource directory to the temporary working directory. The preceding code demonstrates a variant of the `copy` command that uses a `fileset`. Rather than issue a separate `copy` command for each file, you use the `fileset` tag to copy all the files to the temporary working directory in a single command. You can also use wildcards and pattern matching to describe the files to include in a fileset. I like to specify the files individually by name so that I know exactly what is being packaged.

```
<target name="collection_prep1"
  depends="basics_prep,dodger_prep,fraction_prep,mars_prep"/>

<target name="collection_prep2"
  depends="road_prep,shooter_prep,sprite_prep,tile_prep,zombie_prep"/>

<target name="collection_prepare"
  depends="collection_prep1,collection_prep2">

  <copy file="${res_dir}/apps/images/croftsoft.png"
    todir="${tmp_dir}/images"/>

</target>
```

Target collection_prepare depends on collection_prep1 and collection_prep2. I break it up this way so that the depends tag value does not run too long. The purpose of collection_prepare is to copy all the resource files for the individual games that go into the CroftSoft Collection into the temporary working directory. It also copies any additional resource files, such as croftsoft.png, which you use for the frame icon.

```
<target name="collection_compile" depends="collection_prepare">

  <javac srcdir="${src_dir}" destdir="${tmp_dir}">
    <include
      name="com/croftsoft/apps/collection/CroftSoftCollection.java"/>
    <include name="com/croftsoft/ajgp/basics/BasicsExample.java"/>
    [...]
    <include name="com/croftsoft/apps/zombie/Zombie.java"/>
  </javac>

</target>
```

You use the javac task to compile the source code in the src_dir directory and output the compiled class files to tmp_dir. Normally you would only need to include the main class and javac would be smart enough to compile all the other classes that are required. In this case, however, the individual game applets are linked dynamically instead of statically and you must also name them explicitly. This is described further in the next chapter.

```
<target name="collection_jar" depends="collection_compile">
  <echo
    file="manifest.txt" message=
    "Main-Class: com.croftsoft.apps.collection.CroftSoftCollection" />
  <jar
    basedir="${tmp_dir}"
    destfile="${arc_dir}/collection.jar"
    manifest="manifest.txt"
    update="false"/>
  <delete file="manifest.txt"/>
  <delete dir="${tmp_dir}"/>
</target>

<target name="demo" depends="collection_jar">
  <java fork="true" jar="${arc_dir}/collection.jar"/>
</target>
```

If the `collection_compile` target compiles successfully, the `collection_jar` target executes. The `collection_jar` target packages the temporary working directory contents, which include compiled code and resource files. It overwrites the old JAR file if it exists in the output `arc_dir` directory. When completed, the temporary directory is deleted and the newly created executable JAR is launched in a separate Java Virtual Machine (JVM) for testing.

The rest of the build file contains additional targets that compile demonstrations and examples that require optional packages beyond what the core J2SE library includes. A quick review of the comments embedded in the build file will tell you what optional packages are required to compile and run some of the additional example games not included in the default `demo` target. You can safely ignore these other targets for now as I will cover them in later chapters of the book.

```
ant basics
```

If Ant is properly installed, you should simply be able to enter the command `ant` in the directory containing the `build.xml` file and the default `demo` target will compile and launch. If you want to specify a different target, you simply provide it as a command-line argument as shown in the preceding code. Target `basics`, for example, builds and launches an example game described at the end of this chapter. If you have not already done so, go ahead and build and run the demo using Ant and the example build file.

Using Open Source

Hopefully by this point you have been able to compile and run the demonstration program successfully and are now ready to learn how to modify and incorporate the code library as described in the book to create your own games. Before you get too invested in the library, however, you should know the usage limitations and requirements.

Learning the Basics of Copyright

Copyright used to be a mystery to me until I did a little reading on the subject. Here are a few key basics I think every developer should know.

- You do not need to put a copyright statement on your work nor file paperwork for it to be copyrighted. It is copyrighted automatically by the simple act of creation. Putting the copyright statement on your work and registering your copyright does help you when it comes to receiving damage awards for copyright infringement.

- You no longer need to append “All rights reserved” to your copyright statement. The law changed and it is now assumed.

- If you are an employee, your employer is the copyright holder of works you create on the job by default. If you are an independent contractor, you are the copyright holder by default unless your contract states otherwise. Your contract almost always states otherwise unless you drafted the contract yourself.

- If you create a work with someone else, then you both own the copyright jointly unless an agreement states otherwise. Use and distribution requires unanimous consent and you must share profits from such a work.

- It is possible to create a contract where the work is owned jointly without accountability. In this case, each copyright holder can use the work without accounting to the other. It is as though they each own the copyright independently and do not need to share profits or seek mutual permission. This might be a useful arrangement if you are teaming with others to create a game for your portfolio instead of for profit.

- A copyright holder has the exclusive right to use, distribute, and modify the work. If you create a work that incorporates a work by someone else, either in whole or in part, you have created a derivative work. You must have the permission of the copyright holder of the original work in order to create a derivative work legally.

- Copyright law makes an exception for fair use of a work. This means that under certain circumstances you can use a work without the permission of the copyright holder. This is limited to just a few enumerated uses such as parody, teaching, critical review, and news. There are land mines in the Fair Use doctrine so beware. In general, you should not rely upon Fair Use to incorporate material into your game.

- Works created by the government are not entitled to copyright. This is why the space pictures taken by NASA are in the Public Domain. If you are creating a science fiction game that requires images of stars, planets, space shuttles, rockets, and astronauts, you are in luck. See `http://gimp-savvy.com/PHOTO-ARCHIVE/` for a collection of Public Domain space photos you can use in your games.

- A license is a grant of permission by a copyright holder to use a work. Licenses come in all shapes and sizes. Some are exclusive, meaning that the licensee is the only one who can use the work, and some are non-exclusive, meaning that many can use the work simultaneously. If you grant an exclusive license to your game to someone for a limited time, you retain the copyright but you cannot allow someone else to use the game during that period.

- A common misperception is that if something is on the Web and it does not have a copyright statement on it, it is in the Public Domain and can be used without permission and without attribution. This is simply wrong. In general, unless you read something from the copyright holder that explicitly states otherwise, you cannot use it.

My favorite book on this subject is *Web & Software Development: A Legal Guide* by Stephen Fishman (Nolo Press, 2002). Nolo Press is the publisher of a number of extremely useful self-help legal books. I recommend that you read a book on copyright law at least once early on in your career. It is critical to understanding your employee agreements, your client contracts, and incorporating Open Source software.

Choosing a License

The example game code is licensed to you under the terms of an *Open Source* license. An Open Source license grants you the right to use and modify the source code to create your own games for free. It usually also places restrictions on your ability to sue the creator of the Open Source code for damages if something goes awry. These days, most of the code underlying the Internet infrastructure is distributed under the terms of an Open Source license.

While not officially a trademark, Open Source usually refers exclusively to code distributed under the terms of one of the licenses approved by the organization known as the Open Source Initiative (OSI). You can find a list of these approved licenses as well as additional information on the theory behind Open Source at the OSI web site.[1]

Some Open Source licenses are considered *viral* in that they are said to *infect* your program if you incorporate any of the code licensed under their terms in your game. In this case, you must also release your entire game code—the larger work—under the same licensing terms. The idea is that no one can benefit from the use of this *free software* unless they are also willing to share their enhancements and contributions. Developers might use non-viral licenses, on the other hand, in a program where different parts of the code are distributed under different licensing terms, including closed source commercial.

The reusable game code library and example games are available under the terms of both viral and non-viral Open Source licenses. Look in the license subdirectory of the code distribution for the different licenses you can use. Which license you choose is at your discretion based upon your needs and preferences but you must choose one if you are to use the code at all.

If you are undecided, I recommend the Academic Free License (AFL) version 2.0 by Lawrence E. Rosen. It is possibly the least restrictive. For example, it is non-viral so you have the option of incorporating it into a closed source commercial release of your game. Other choices include the Open Software License (OSL), the GNU General Public License (GPL), and the GNU Lesser GPL (LGPL).

```
My Cool Game v1.0
Copyright 2004 John Doe.

Portions Copyright 2003 CroftSoft Inc.
Licensed under the Academic Free License version 2.0
http://www.croftsoft.com/
```

Note that most, if not all, Open Source licenses require that you maintain the attribution and copyright notice for any included or modified code. If you print something like the preceding code to `System.out` in the background whenever your game starts up, that is good enough for me.

Renaming Modified Code

If you modify one of the files in the game library, I request that you change the package name so the modified class is no longer in the `com.croftsoft` package hierarchy. The general rule is that you should use your unique domain name

1. http://www.opensource.org/

reversed for the beginning of your package prefix to prevent naming conflicts. For example, my domain name is croftsoft.com, so the universally unique package name prefix is `com.croftsoft`.

If you do not have your own domain name you can use, you should consider getting one. Domain name registration now costs as little as $8.95 per year from registrars such as Go Daddy Software.[2] Hosting for your web pages and applets is less than $8 per month from some providers. If you want the domain name but do not want to pay for the web hosting, you can use the domain forwarding service from your name registrar to redirect visitors to the personal web page that comes free with your Internet Service Provider (ISP) account.

I also ask, but do not require, that you retain my name in the modified source code files as one of the authors as indicated by the `@author` javadoc comment. Each time a new contributor modifies the file, the list of `@author` tags should grow.

Sharing the Source

In addition to the CroftSoft Code Library, you might find source code suitable for your game development needs from an Open Source repository. The Open Source repository SourceForge.net, for example, is host to a number of different Java game programming projects.[3] In addition to finer categorizations, it organizes the code by *foundry* such as the Gaming Foundry and the Java Foundry. Look for projects within the intersection of these two sections.

FreshMeat.net has an impressive search-filtering function that allows you, for example, to look for projects that use an OSI-approved Open Source license, use the Java programming language, and have the word "game" in the description. This search includes projects at other sites such as SourceForge.net.

Sun Microsystems has recently launched the "Java Games and Game Technologies Projects" web site which is starting to attract some Open Source contributions.[4] I hope this will become the future center for this type of code, in effect serving as what would be called a "Java Gaming Foundry" if it existed on SourceForge.net. SourceForge.net might continue to dominate, however, as it is much less restrictive as to what projects can be created and how they can be organized.

The Game Developers Java Users Group (GameJUG) has an electronic mailing list, `gamejug-open`, specifically for the discussion of Open Source Java game development efforts.[5] I recommend that you subscribe to one or more of the GameJUG mailing lists such as `gamejug-announce` and `gamejug-news`. As GameJUG volunteers, I and others frequently post useful information and links about the Java game programming industry that you might not find anywhere else.

2. `http://www.godaddy.com/`
3. `http://www.sourceforge.net/`
4. `http://games.dev.java.net/`
5. `http://www.GameJUG.org/`

Finding Multimedia for Your Games

Game development teams are usually composed of programmers, graphic artists, audio engineers, and game designers. If you are creating a game by yourself, you must perform all these roles. This means you need to come up with your own graphics and audio files. Even if you are working with others, you might want to generate the occasional image or audio file just as a temporary placeholder.

Graphics

If you do not have access to an artist and you do not have the skills or time to create the graphics yourself, a couple options are open to you. The first is to use free graphics (see Figure 1-2). All the graphics files used in the example games are available for download from the book web site. These files are dedicated to the Public Domain, which means you may use them in your own games without paying royalties and without even needing to mention where you got them. If you substantially modify these graphics, you own the copyright to the derivative work.

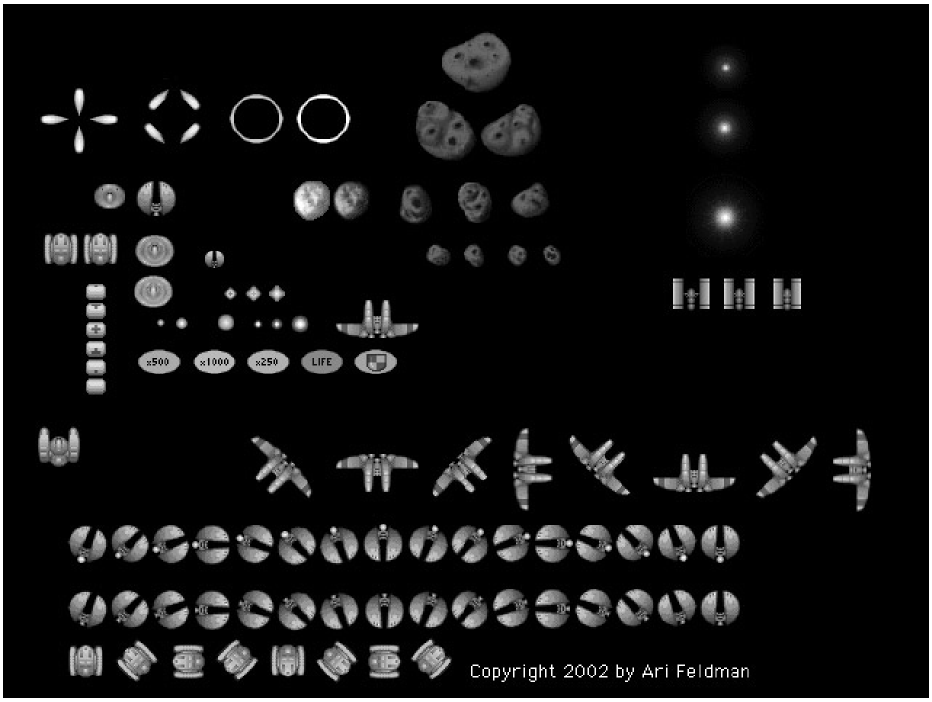


Figure 1-2. Sprite graphics by Ari Feldman. Artwork copyright 2002, 2003 by Ari Feldman.

Ari Feldman, author of the book *Designing Arcade Computer Game Graphics*, provides a free sprite graphics library, SpriteLib GPL, from his web site.[6] See the `License.txt` file within the zip archive for usage terms. Note that the acronym *GPL* in this case does not refer to the GNU General Public License.

James Gholston, president and general partner of the game development company Dimensionality, has released some of his professional graphics to the Public Domain so they could be included in the example games for this book. These graphics are available from the Dimensionality web site and the book web site.[7]

Clip Art collections are another possible source of royalty-free graphics. Additionally, any graphics files distributed with an Open Source game are probably available for reuse under the same licensing terms as the code. When incorporating free graphics into your game, be sure to understand the usage terms or licensing agreement as many sources only permit royalty-free use for limited personal or non-commercial applications.

The second option is to use a digital camera. The images might not be fantastic but it is hard to get more realistic. Simply take a picture of what you want in your game, scale and crop it, and then make the background transparent. An Open Source image-editing tool you might want to use for this purpose is the GNU Image Manipulation Program (GIMP). GIMP is available on Unix and Windows.[8]

You can also use GIMP to create new images from scratch if you do decide to do some of your own artwork. I frequently use the MS Paint accessory that comes with Windows to create simple placeholder graphics. I notice that the most recent version that comes with Windows XP can now save to PNG format. The free image drawing tools Project Dogwaffle and Pixia are other possibilities.[9]

Audio

All the audio files used in the example games are available for download from the book web site. They are also dedicated to the Public Domain and you can use and modify them without accountability and without attribution.

The sound-effects library "6000 Sound Effects" from Cosmi is another possible source of material for your games.[10] This package is available from Amazon.com for $9.99. The vendor web site states that you can use these sounds royalty-free in your commercial computer games. Keep in mind that there might be licensing complications when distributing these files with an Open Source game.

6. `http://www.arifeldman.com/`
7. `http://www.dimensionality.com/free.html`
8. `http://www.gimp.org/`
9. `http://www.squirreldome.com/cyberop.htm`, `http://www.ab.wakwak.com/~knight/`
10. `http://www.cosmi.com/`

If you want to record your own audio, all you need is a microphone. I do not have a lot of experience with audio editors but I have successfully used the shareware version of GoldWave in the past.[11] You will often want to use such an editor to crop your audio recordings to the appropriate sample length for your game.

Basics Example

Assuming your development environment is set up, you should be able to start creating your own games after reading just this first chapter. You can modify the example game shown in Figure 1-3 to get started.

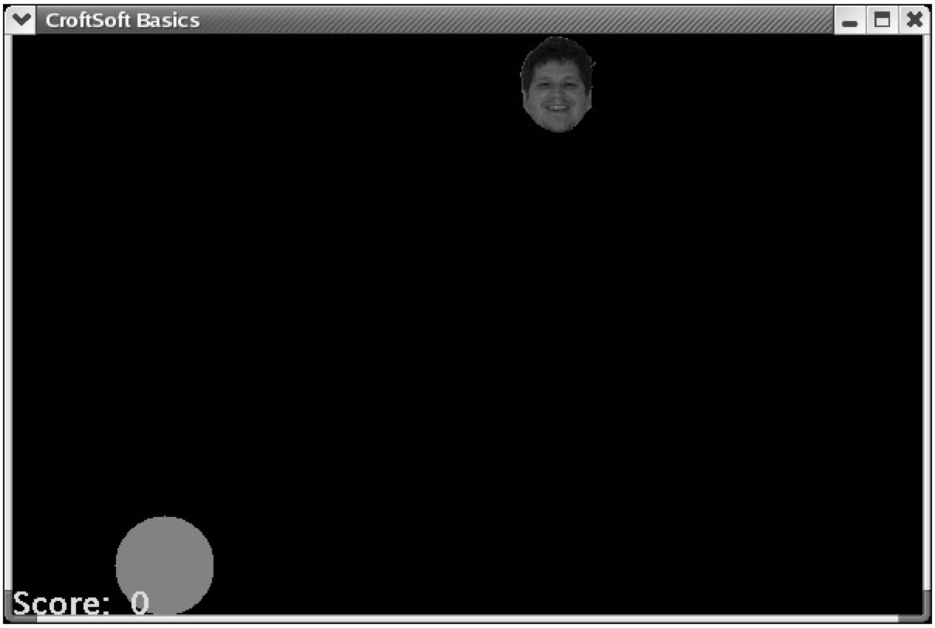


Figure 1-3. Basics example

The object of this simple game is to launch a ball at a moving target at the top of the screen. You can build and launch this example game using the Ant target `basics`. In a minimum amount of code, it demonstrates loading media files, drawing shapes and images, displaying text, playing sounds, random number generation, state maintenance, animation, mouse and keyboard input, collision detection, and dirty rectangle management.

11. `http://www.goldwave.com/`

Modifying the Source

This game is built upon an extensive game library that is described in subsequent chapters. As I review the source code, I will point out which code you can safely modify and which you should leave as is until you have read further. As this example covers most of the Java game programming basics, many readers might find they can simply copy and modify this file to create games successfully without even having to read the next chapter.

```
package com.croftsoft.ajgp.basics;
```

You can find this game in package `com.croftsoft.ajgp.basics`. The first thing you want to modify is the package name.

```
import java.applet.Applet;
import java.applet.AudioClip;
import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.util.Random;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JComponent;

import com.croftsoft.core.CroftSoftConstants;
import com.croftsoft.core.animation.AnimatedApplet;
import com.croftsoft.core.animation.AnimationInit;
```

Remove `import` statements at your own risk. Note that only three custom classes are imported. The rest are all from the Java core.

```
  public final class  BasicsExample
    extends AnimatedApplet
  //////////////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////////////
  {
```

Change the class name but continue to extend the superclass `AnimatedApplet`.

```
  private static final String  VERSION
    = "2003-11-06";

  private static final String  TITLE
    = "CroftSoft Basics";

  private static final String  APPLET_INFO
    = "\n" + TITLE + "\n"
    + "Version " + VERSION + "\n"
    + CroftSoftConstants.COPYRIGHT + "\n"
    + CroftSoftConstants.DEFAULT_LICENSE + "\n"
    + CroftSoftConstants.HOME_PAGE + "\n";
```

Change the values for the `VERSION`, `TITLE`, and `APPLET_INFO`.

```
  private static final Color      BACKGROUND_COLOR
    = Color.BLACK;

  private static final Cursor     CURSOR
    = new Cursor ( Cursor.CROSSHAIR_CURSOR );

  private static final Font       FONT
    = new Font ( "Arioso", Font.BOLD, 20 );

  private static final Double     FRAME_RATE
    = new Double ( 30.0 );

  private static final Dimension  FRAME_SIZE
    = new Dimension ( 600, 400 );

  private static final String     SHUTDOWN_CONFIRMATION_PROMPT
    = "Close " + TITLE + "?";
```

You can specify the background color, mouse cursor type, text font, animation speed, window size, and the shutdown confirmation prompt. These constants are passed to superclass AnimatedApplet. You probably want to keep all these constants but customize the values as required.

```
private static final String  MEDIA_DIR = "media/basics/";

private static final String  AUDIO_FILENAME
  = MEDIA_DIR + "drip.wav";

private static final String  IMAGE_FILENAME
  = MEDIA_DIR + "croftsoft.png";
```

Add or remove media file names as required for your game.

```
private static final long    RANDOM_SEED = 0L;

private static final Color   BALL_COLOR = Color.RED;

private static final Color   SCORE_COLOR = Color.GREEN;

private static final int     VELOCITY = 3;
```

Add or remove additional constants as required for your game. You use the RANDOM_SEED with the random number generator. The VELOCITY is the speed of the bowling ball in pixels per frame.

```
private final Rectangle  componentBounds;

private final Random     random;
```

The variable componentBounds is a Rectangle for storing the width and height of the game screen. Variable random is the random number generator. You probably need these in most of your games.

```
private final Rectangle  ballRectangle;

private final Rectangle  targetRectangle;
```

Variables ballRectangle and targetRectangle are used for collision detection. These variables are game-specific.

```
private boolean    componentResized;

private KeyEvent   keyEvent;
private Point      mousePoint;

private boolean    mousePressed;
```

These variables are used for flagging or storing user input events. You will want to use these in most of your games.

```
private AudioClip  audioClip;

private Icon       icon;
```

You will have an `audioClip` or `icon` for each media file you use.

```
private boolean    rolling;

private int        score;
```

These are game-specific state variables. Variable `rolling` indicates whether the ball has been launched.

```
public static void  main ( String [ ]  args )
//////////////////////////////////////////////////////////////////////
{
  launch ( new BasicsExample ( ) );
}
```

In the `main()` method, change the `BasicsExample` to the name of your new class.

```
private static AnimationInit  createAnimationInit ( )
//////////////////////////////////////////////////////////////////////
{
  AnimationInit  animationInit = new AnimationInit ( );

  animationInit.setAppletInfo ( APPLET_INFO );

  animationInit.setCursor ( CURSOR );

  animationInit.setFont ( FONT );

  animationInit.setFrameIconFilename ( IMAGE_FILENAME );

  animationInit.setFrameRate ( FRAME_RATE );
```

```
      animationInit.setFrameSize ( FRAME_SIZE );

      animationInit.setFrameTitle ( TITLE );

      animationInit.setShutdownConfirmationPrompt (
        SHUTDOWN_CONFIRMATION_PROMPT );

      return animationInit;
    }
```

The static method `createAnimationInit()` sets a number of parameters commonly used by the superclass `AnimatedApplet`. You probably will not need to modify this method.

```
    public  BasicsExample ( )
    //////////////////////////////////////////////////////////////////////
    {
      super ( createAnimationInit ( ) );

      componentBounds = new Rectangle ( );

      random = new Random ( RANDOM_SEED );
```

Change the name of this constructor method to the new name of your class. Keep the initial call to the superclass constructor as is. You almost always need the `componentBounds` variable. Many games use a random number generator.

```
      animatedComponent.addComponentListener (
        new ComponentAdapter ( )
        {
          public void  componentResized (
            ComponentEvent  componentEvent )
          {
            componentResized = true;
          }
        } );

      animatedComponent.addKeyListener (
        new KeyAdapter ( )
        {
          public void  keyPressed  ( KeyEvent  ke )
          {
            keyEvent = ke;
          }
        } );
```

```
    animatedComponent.addMouseListener (
      new MouseAdapter ( )
      {
        public void  mousePressed ( MouseEvent  mouseEvent )
        {
          mousePressed = true;
        }
      } );

    animatedComponent.addMouseMotionListener (
      new MouseMotionAdapter ( )
      {
        public void  mouseMoved ( MouseEvent  mouseEvent )
        {
          mousePoint = mouseEvent.getPoint ( );
        }
      } );
```

These methods attach user input event listeners to the `animatedComponent`. Variable `animatedComponent` is inherited from the superclass `AnimatedApplet`. You normally want to have these event listeners in all your games.

```
    ballRectangle   = new Rectangle ( );

    targetRectangle = new Rectangle ( );
  }
```

You can initialize many of your game-specific objects in your constructor method.

```
  public void  init ( )
  //////////////////////////////////////////////////////////////////////
  {
    super.init ( );

    animatedComponent.requestFocus ( );

    componentResized = true;
```

As described in Chapter 2, method `init()` is called when your game first starts to perform any additional initialization not performed in your constructor method. You generally always want to call the superclass `init()` method, request the keyboard focus, and set `componentResized` to `true`. Setting the `componentResized` flag is necessary to load the screen width and height into the `componentBounds` variable for the first time.

```
    ClassLoader  classLoader = getClass ( ).getClassLoader ( );

    audioClip = Applet.newAudioClip (
      classLoader.getResource ( AUDIO_FILENAME ) );

    icon = new ImageIcon (
      classLoader.getResource ( IMAGE_FILENAME ) );
```

For reasons explained in subsequent chapters, the creation of some of your media objects must be delayed until the initialization method `init()` is called, otherwise a `NullPointerException` might be thrown. This example demonstrates how to load audio and image file data into memory.

```
    ballRectangle.width    = icon.getIconWidth  ( );

    ballRectangle.height   = icon.getIconHeight ( );

    targetRectangle.width  = icon.getIconWidth  ( );

    targetRectangle.height = icon.getIconHeight ( );
  }
```

The dimensions of the ball and the target are equal to the image dimensions.

```
  public void  update ( JComponent  component )
  //////////////////////////////////////////////////////////////////////
  {
```

As explained in Chapter 3, the `update()` method is where you perform all your state updates. It is called each time a new frame of animation is required by the game loop. The game loop logic is provided by the superclass `AnimatedApplet`.

```
    if ( componentResized )
    {
      componentResized = false;

      component.repaint ( );

      component.getBounds ( componentBounds );
    if ( !rolling )
    {
      ballRectangle.y = componentBounds.height - ballRectangle.height;
    }
  }
```

When the player resizes the desktop window containing the game, an event is generated. This event is intercepted by an event listener created in the constructor method. The processing of user input events should be delayed until the `update()` method is called. For this reason, all the event listeners created in the constructor method simply flag or store the events instead of processing them immediately.

One of the event listeners raises the boolean flag `componentResized` to indicate that the size of the game screen has changed. You almost always want to handle this event in your `update()` method implementation by resetting the flag to `false`, requesting a repaint of the entire screen, and storing the new dimensions in `componentBounds` as shown in the preceding code.

You might then want to provide additional game-specific logic to adjust state coordinates based upon the new dimensions. In this example, the position of the ball is placed at the bottom edge of the screen if it has not already been launched.

```
  boolean  rollRequested = false;

  if ( mousePressed )
  {
    mousePressed = false;

    rollRequested = true;
  }
```

When users press a mouse button, it generates an event that is intercepted by one of the mouse listeners created in the constructor. The mouse listener raises the boolean flag `mousePressed` so it can be processed later when the `update()` method is called. In this example, pressing the mouse causes the variable `rollRequested` to be set to `true` indicating that the ball should be launched at the target.

```
  int  ballMove = 0;

  if ( keyEvent != null )
  {
    int  keyCode = keyEvent.getKeyCode ( );
    switch ( keyCode )
    {
      case KeyEvent.VK_SPACE:

        rollRequested = true;

        break;

      case KeyEvent.VK_LEFT:
      case KeyEvent.VK_KP_LEFT:

        ballMove = -1;

        break;

      case KeyEvent.VK_RIGHT:
      case KeyEvent.VK_KP_RIGHT:

        ballMove = 1;

        break;
    }

    keyEvent = null;

    mousePoint = null;
  }
```

You can also launch the ball at the target by pressing the spacebar on the keyboard. Pressing the left or right arrows sets the `ballMove` variable to -1 or +1, respectively. After the `update()` method processes a keyboard event, `keyEvent` is set to null so that the same keyboard event will not be processed again the next time the `update()` method is called by the game loop. The `mousePoint` is set to null whenever a `keyEvent` is generated as keyboard events have priority over mouse events.

```
  if ( mousePoint != null )
  {
    if ( mousePoint.x
      < ballRectangle.x + ballRectangle.width / 2 - VELOCITY)
    {
      ballMove = -1;
    }
    else if ( mousePoint.x
      > ballRectangle.x + ballRectangle.width / 2 + VELOCITY )
    {
      ballMove = 1;
    }
    else
    {
      mousePoint = null;
    }
  }
```

Before it is launched, you can also move the ball left and right using the mouse. When the player moves the mouse, an event listener created in the constructor stores the position of mouse cursor as the `mousePoint`. Each time the `update()` method is called, it will move the center of the ball closer to the most recently generated `mousePoint.x` value. Once the center of the ball has reached the `mousePoint.x` value, or close enough that any further movement might cause an overshoot, the `mousePoint` is set to null.

```
  if ( rollRequested )
  {
    if ( !rolling )
    {
      audioClip.play ( );

      rolling = true;
    }
  }
```

If the player has requested by either pressing the mouse or pressing the spacebar on the keyboard that the ball be launched at the target, a sound will be played if the ball is not already `rolling`.

```
  component.repaint ( targetRectangle );
```

As explained in a subsequent chapter, the `update()` method must request a repaint of the objects on the screen in their old and new positions in order to generate the animation. In this example, a repaint of the area containing the old position of the target is requested. This is done every time the `update()` method is called because the target is always moving.

```
  if ( rolling )
  {
    component.repaint ( ballRectangle );

    ballRectangle.y -= VELOCITY;
```

If the ball is rolling toward the target, a repaint of the screen area containing its old position is requested and its `y` coordinate is decremented by its `VELOCITY`.

```
boolean  reset = false;

if ( targetRectangle.intersects ( ballRectangle ) )
{
  reset = true;

  targetRectangle.x = -targetRectangle.width;

  audioClip.play ( );

  score++;

  component.repaint ( );
}
```

If the rolling ball collides with the target, the game is reset, a sound is played, the score is incremented, and a repaint of the entire screen is requested.

```
else if ( ballRectangle.y + ballRectangle.height < 0 )
{
  reset = true;

  if ( score > 0 )
  {
    score--;
  }

  component.repaint ( );
}
```

If the ball is rolling and it misses the target, the game is reset and the score is decremented. A repaint of the entire screen is requested so that the updated score will be painted.

```
if ( reset )
{
  ballRectangle.y = componentBounds.height - ballRectangle.height;

  rolling = false;
}
```

If the game was reset either because the ball hit or missed the target, the ball is moved back to the launching position at the bottom of the screen and the `rolling` state flag is reset to `false`.

```
        component.repaint ( ballRectangle );
      }
```

Because the ball is rolling, it must be repainted in its new position.

```
      else if ( ballMove != 0 )
      {
        component.repaint ( ballRectangle );

        ballRectangle.x += ballMove * VELOCITY;

        if ( ballRectangle.x < 0 )
        {
          ballRectangle.x = 0;
        }

        if ( ballRectangle.x
          > componentBounds.width - ballRectangle.width )
        {
          ballRectangle.x
            = componentBounds.width - ballRectangle.width;
        }

        component.repaint ( ballRectangle );
      }
```

If the ball is not rolling toward the target, it might be moving left or right as the player aims it in the launching area. This is indicated by a non-zero value for `ballMove`. The ball cannot move off the left or right edges of the screen. Repaints of the screen in the old and new positions of the ball are requested.

```
      if ( score > 1 )
      {
        targetRectangle.x += random.nextInt ( score ) * VELOCITY;
      }
      else
      {
        targetRectangle.x += VELOCITY;
      }
```

The horizontal velocity of the target becomes more random as the player score increases. This makes the game more of a challenge for the player.

```
    if ( targetRectangle.x >= componentBounds.width )
    {
      targetRectangle.x = -targetRectangle.width;
    }
```

If the target moves off the right edge of the screen, it continues on from the left side so that it is moving continuously across the top of the screen from left to right.

```
    component.repaint ( targetRectangle );
  }
```

The final act of the `update()` method is to request a repaint of the screen area containing the new position of the target. The repaint request for the old position was made earlier in the `update()` method.

```
  public void  paint (
    JComponent  component,
    Graphics2D  graphics )
  //////////////////////////////////////////////////////////////////////
  {
```

As described in Chapter 3, the `paint()` method is called whenever the screen needs to be repainted, usually once per game loop during animation. The calling of the `paint()` method is managed by the superclass `AnimatedApplet` and is triggered by the repaint requests made in the `update()` method.

```
    graphics.setColor ( BACKGROUND_COLOR );

    graphics.fill ( componentBounds );
```

The background is always drawn first.

```
    icon.paintIcon ( component, graphics, targetRectangle.x, 0 );
```

The target image is painted.

```
    graphics.setColor ( BALL_COLOR );

    graphics.fillOval (
      ballRectangle.x,
      ballRectangle.y,
      ballRectangle.width,
      ballRectangle.height );
```

The ball is drawn.

```
    graphics.setColor ( SCORE_COLOR );

    graphics.drawString (
      "Score:  " + Integer.toString ( score ),
      0, componentBounds.height );
  }
```

Finally the score text is drawn. Because the score is drawn after the ball in the `paint()` method, the score cannot be occluded by the ball.

Notice that no game state is updated in the `paint()` method and no graphics are painted in the `update()` method. It is a sin to do otherwise.

Modifying the Build

After playing with the game example and reviewing its source code, you might want to go ahead and use it as a template for a game of your own invention. I recommend that you do so using an incremental approach, building and testing after each small change. The first change you might want to make is to change the package name for the class. Even this one-line change to the code can be frustrating if you do not consider the simultaneous changes that need to be made to the Ant build file. The following code documents the required changes.

```
<target name="basics_prep" depends="init">
  <copy todir="${tmp_dir}/media/basics">
    <fileset dir="${res_dir}/ajgp/basics/media">
      <include name="croftsoft.png"/>
      <include name="drip.wav"/>
    </fileset>
  </copy>
</target>
```

If you add or remove media files for the game, you must modify target `basics_prep`.

```
<target name="basics" depends="basics_prep">
  <javac srcdir="${src_dir}" destdir="${tmp_dir}">
    <include name="com/croftsoft/ajgp/basics/BasicsExample.java"/>
    </javac>
    <echo
      file="manifest.txt"
      message="Main-Class: com.croftsoft.ajgp.basics.BasicsExample" />
    <jar
      basedir="${tmp_dir}"
      destfile="${arc_dir}/basics.jar"
      manifest="manifest.txt"
      update="false"/>
    <delete file="manifest.txt"/>
    <delete dir="${tmp_dir}"/>
    <java fork="true" jar="${arc_dir}/basics.jar"/>
  </target>
```

If you change the package or class name, you must modify target `basics`.

This wraps up the Java game programming basics example. If any of it is a bit of a mystery, be assured that all will be explained in subsequent chapters. Later chapters will also make the point that while this code serves as a useful example and an initial template, there are a number of improvements that could be made.

Summary

In this chapter, you learned how to compile the example code used throughout the book using the development build tool Ant. You also learned about other development tools such as image and audio editors. I identified sources for code and multimedia you can incorporate into your games, and explained Open Source licensing terms. I documented the source code for an example game demonstrating the Java game programming basics for reuse as a template. In the following chapters, I describe the classes available to you within the reusable game library in detail.

Further Reading

Eckstein, Robert. *XML Pocket Reference*, 2nd edition. Sebastopol, CA: O'Reilly & Associates, 2001.

Feldman, Ari. *Designing Arcade Computer Game Graphics*. Plano, TX: Wordware Publishing, 2000.

Fishman, Stephen. *The Public Domain: How to Find & Use Copyright-Free Writings, Music, Art & More*. Berkeley, CA: Nolo Press, 2001.

Fishman, Stephen. *Web & Software Development: A Legal Guide*, 3rd edition. Berkeley, CA: Nolo Press, 2002.