

**Open Source Physics:<sup>1</sup>  
A User's Guide with Examples  
(Draft)**

**Wolfgang Christian**

Includes

Physics Curricular Material by Mario Belloni

*Tracker* Video Analysis and OSP XML by Doug Brown

*BQ* Database by Anne Cox and William Junkin

*Easy Java Simulations* by Francisco Esquembre

March 20, 2017

<sup>1</sup>Open Source Physics is supported in part by National Science Foundation grants DUE-0126439 and DUE-0442481.

## OPEN SOURCE PHYSICS

The Open Source Physics project is a synergy of curriculum development, computational physics, and physics education research. One goal of the project is to make a large number of Java simulations available for education using the GNU Open Source model. This Guide describes some of the classes and interfaces that are being used in this project.

Portions of Open Source Physics are being incorporated into the following projects:

- The third edition of *An Introduction to Computer Simulation Methods* by Harvey Gould, Jan Tobochnik, and Wolfgang Christian.
- The *Statistical and Thermal Physics* project by Harvey Gould and Jan Tobochnik.
- The *Easy Java Simulations* high-level modeling tool by Francisco Esquembre.
- The *Tracker* video analysis program by Doug Brown.

Programmers wishing to adopt Open Source Physics tools for their projects are encouraged to do so, provided that they release their source code under the GNU Open Source GPL license. Open Source Physics code is being distributed as compressed archives (zip files) from the Open Source Physics server hosted at Davidson College and using CVS from the *SourceForge* web site. Additional curricular material and links are available on the Open Source Physics server.

<http://www.opensourcephysics.org>  
<http://sourceforge.net/projects/os-physics/>

## LICENSE

Open Source Physics software is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation; either version 2 of the License, or(at your option) any later version.

Code that uses any portion of the code in the org.opensourcephysics package or any subpackage (subdirectory) of this package must also be released under the GNU GPL license.

A copy of the GNU General Public License is included on the CD accompanying this book. If this CD is not available, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston MA 02111-1307 USA or view the license online at

<http://www.gnu.org/copyleft/gpl.html>

## AUDIENCE

This Guide is intended as a guide to the tools and philosophy of the Open Source Physics project. It does not intend to teach computational physics, numerical analysis, or Java programming. There are many books for these purposes.

Prose descriptions of a class are inexact, and there is no substitute for a short demonstration program. This Guide provides many such programs. Another reason for the Guide is to document the test suite we use to maintain the integrity of the Open Source Physics library before distributing it on the Internet.

## SUPPORT

To keep the objects simple, the Open Source Physics Library does not include advanced features, such as serialization and synchronization. Although we have tried to make the Open Source Physics software as bug-free as possible, we cannot guarantee that the Library is free of bugs. If you find a bug and would like to contact the authors(s), email is your best option. The following email address can be used:

wochristian@davidson.edu

All reasonable questions will be answered and bugs fixed as time allows. But please do not make incredible demands and expect them to be fulfilled.

## ACKNOWLEDGEMENTS

There are very many people and institutions that have contributed to the Open Source Physics project, and I take great pleasure in acknowledging their support and their interest. The Open Source Physics project would not have been possible without the support and collaboration of my longtime friend and Davidson colleague, Mario Belloni. Harvey Gould at Clark University and Jan Tobochnik at Kalamazoo College have pioneered the teaching of computational physics to both undergraduate and graduate students. This Guide would not have been possible without their early adoption of the Open Source Physics libraries in their teaching and curriculum development projects. Anne Cox has provided many insightful and thoughtful comments about the Guide and the physics examples and has hosted an Open Source Physics developers workshop at Eckerd College.

Douglas Brown at Cabrillo College and Francisco Esquembre at the University of Murcia, Spain, have contributed code, fixed bugs, and made suggestions on the OSP architecture. Joshua Gould of the Broad Institute at MIT and Harvard and Kipton Barros of Boston University were invaluable in helping to design and test the Open Source Physics libraries. Aaron Titus and Matt DesVoigne tested the OSP libraries and suggested numerous improvements to this Guide. Glenn Ford

developed the JOGL implementation of the OSP 3D API. We are especially thankful to students and faculty at Davidson College, Clark University, and Kalamazoo College who have generously commented on the Open Source Physics project as they suffered through early versions of the library.

The Open Source Physics project makes use of a number of software packages released under open source licenses. In particular we acknowledge the following authors:

- Fast Fourier Transformation, FFT, package (`jnt.fft`) by Bruce Miller.
- Java expression parser package (`org.nfunk.jep`) by Nathan Funk.
- A function plotting program by Yanto Suryono that is the basis for the Contour and Surface Plot components.
- Differential equation solvers by Andrew Gusev and Yuri. B. Senichenkov at Saint Petersburg Polytechnic University, Russia.

These optional packages are included on the Open Source Physics Software CD, but newer versions may be available on the web or from their respective authors.

The Open Source Physics project is indebted to Leigh Brookshaw for publishing his graphics package under the GNU GPL license. Portions of this package were used in developing our own x-axis and y-axis components. Many thanks to Paul Mutton for making the `EpsGraphics2D` package available under the GPL license. This package makes it easy to create high quality EPS graphics for use in documents and papers.

Davidson College has supported the Open Source Physics in many ways including the hosting of the Open Source Physics web server and by providing an academic home for the author of this Guide. OSP would not be possible without this generous support.

The Open Source Physics project is supported in part by the National Science Foundation grants DUE-0126439 and DUE-0442481. These grants have helped us to write books, to provide workshops at professional meetings, and to develop curricular materials for distribution on the internet that use Open Source Physics simulations.

Finally, I thank my wife Barbara and my children Beth, Charlie, and Konrad Rudolf for their love and support. Although such acknowledgements may appear gratuitous, anyone who has ever written a book knows otherwise.

# Contents

<b>Open Source Physics</b>	ii
<b>Audience</b>	iii
<b>Support</b>	iii
<b>Acknowledgements</b>	iii
<b>1 ■ Introduction to Open Source Physics</b>	1
1.1    Introduction 1	
1.2    MVC Design Pattern 3	
1.3    Simulations and Pedagogy 3	
1.4    Using Launcher 5	
1.5    Installing Java 7	
1.6    Packages 11	
1.7    Eclipse 14	
1.8    Building Programs with Ant 16	
1.9    Documentation 18	
1.10   Programs 18	
<b>2 ■ A Tour of Open Source Physics</b>	23
2.1    Object Oriented Programming 23	
2.2    OSP Frames 23	
2.3    Function Plotter 25	
2.4    SHO 28	
2.5    Three-Dimensional Framework 32	
2.6    Programs 34	
<b>3 ■ Frames Package</b>	36

**vi**

3.1	Overview	36
3.2	Display Frame	38
3.3	Plot Frame	40
3.4	Scalar and Vector Fields	43
3.5	Complex Functions	45
3.6	Fourier Transformations	48
3.7	Display 3D Frame	51
3.8	Tables	54
3.9	Programs	55

**4 ■ Drawing** **57**

4.1	Overview	57
4.2	Drawables	59
4.3	Drawing Panel and Frame	61
4.4	Inspectors	62
4.5	Message Boxes	63
4.6	Scale	64
4.7	Measurable	66
4.8	Interactive	67
4.9	Affine Transformations	70
4.10	Shapes	72
4.11	Interactive Shapes	75
4.12	Text	77
4.13	Measured Images and Snapshots	78
4.14	Effective Java	81
4.15	Programs	83

**5 ■ Controls and Threads** **85**

5.1	Overview	85
5.2	Control Interface	87
5.3	Calculations	90
5.4	Threads	92
5.5	Animations	97
5.6	Simulations	100
5.7	Ejs Controls	103
5.8	Programs	109

**6 ■ Plotting** **112**

6.1	Overview	112
6.2	Dataset	113

6.3	Dataset Manager	114
6.4	Markers	117
6.5	Axes	119
6.6	Log scale	121
6.7	Dataset Tables	122
6.8	Programs	127
<b>7</b>	<b>■ Animation, Images, and Buffering</b>	<b>129</b>
7.1	Animation	129
7.2	Buffered Panels	132
7.3	Buffer Strategy	135
7.4	Simulation Architecture	136
7.5	Drawable Buffer	138
7.6	Data Raster	139
7.7	Programs	141
<b>8</b>	<b>■ Visualization of Two Dimensional Fields</b>	<b>143</b>
8.1	Overview	143
8.2	Images and Rasters	147
8.3	Lattices	150
8.4	Plot2D Interface	153
8.5	Scalar Fields	156
8.6	Vector Fields	159
8.7	Complex Fields	161
8.8	Data Models	163
8.9	Overlays	166
8.10	Programs	168
<b>9</b>	<b>■ Differential Equations and Dynamics</b>	<b>171</b>
9.1	Overview	171
9.2	ODE Interface	173
9.3	ODE Solver Interface	175
9.4	Algorithms	176
9.5	Adaptive Step Size	181
9.6	Multi-Stepping	184
9.7	High-Order ODE Solvers	186
9.8	Conservation Laws	188
9.9	Collisions and State Events	193
9.10	Programs	198

<b>10 ■ Numerics</b>	<b>200</b>
10.1 Functions 200	
10.2 Derivatives 203	
10.3 Integrals 206	
10.4 Parsers 209	
10.5 Finding the Roots of a Function 211	
10.6 Polynomials 214	
10.7 Fast Fourier Transform (FFT) 220	
10.8 Programs 226	
<b>11 ■ Three Dimensional Modeling</b>	<b>228</b>
11.1 Overview 228	
11.2 Simple 3D 229	
11.3 Drawing Panel 3D 231	
11.4 Cameras 233	
11.5 Elements and Groups 235	
11.6 Interactions 238	
11.7 Vectors 244	
11.8 Transformations 247	
11.9 Matrix Transformations 249	
11.10 Angle-Axis Representation 251	
11.11 Euler Angle Representation 254	
11.12 Quaternion Representation 255	
11.13 Rigid Body Dynamics 258	
11.14 JOGL 261	
11.15 Programs 267	
<b>12 ■ XML Documents</b>	<b>269</b>
12.1 Basic XML 269	
12.2 Object Properties 271	
12.3 Object Loaders 274	
12.4 OSP Applications 277	
12.5 Clipboard Data Transfer 280	
12.6 Programs 281	
<b>13 ■ Video</b>	<b>283</b>
13.1 Overview 283	
13.2 Animated Gifs 284	
13.3 Playing QuickTime Video 286	

13.4	Controlling Video Display	288
13.5	Video Capture Tool	290
13.6	Image and Video Analysis	291
13.7	Programs	294

<b>14 ■ Utilities</b>	<b>295</b>
-----------------------	------------

14.1	Resource Loader	295
14.2	Custom Menus	298
14.3	Chooser	301
14.4	Static Methods	302
14.5	Inspectors	303
14.6	Logging	305
14.7	Programs	306

<b>15 ■ Authoring Curricular Material</b>	<b>308</b>
---	------------

15.1	Overview	308
15.2	Launcher	309
15.3	LaunchBuilder	310
15.4	Curriculum Development Overview	320
15.5	Classical Mechanics: Orbits	321
15.6	Electromagnetism: Radiation	326
15.7	Quantum Mechanics: Superpositions	331

<b>16 ■ Tracker</b>	<b>337</b>
---------------------	------------

16.1	Overview	337
16.2	Video Analysis	337
16.3	Examples	338
16.4	Installing and Using Tracker	341
16.5	User Interface	341
16.6	Videos	346
16.7	Coordinate System	349
16.8	Tracks	353
16.9	Track Types	355
16.10	Views	360
16.11	Tracker XML Documents	362
16.12	Glossary	363

<b>17 ■ Easy Java Simulations</b>	<b>366</b>
-----------------------------------	------------

17.1	Introduction	366
------	--------------	-----

x

17.2	The Model–View–Control Paradigm Made Simpler	368
17.3	Inspecting an existing simulation	370
17.4	Running a Simulation	382
17.5	Modifying a Simulation	386
17.6	A global vision	394

## 18 ■ Dissemination And Databases

396

18.1	Overview	396
18.2	Disseminating Programs Using The Internet	396
18.3	The BQ-OSP Database	399
18.4	Searching and Browsing	400
18.5	Integration with Digital Libraries	402
18.6	Editing Curricular Material	403
18.7	Editing Animation by Changing XML	403
18.8	Uploading New Materials	405
18.9	Database Installation	406

## CHAPTER

## 1

## Introduction to Open Source Physics

We describe the tools and philosophy of the Open Source Physics project.

### 1.1 ■ INTRODUCTION

The switch from procedural to object-oriented (OO) programming has produced dramatic changes in professional software design. However, object-oriented techniques have not been widely adopted by scientists teaching computational physics and computer simulation methods. Although most scientists are familiar with procedural languages such as Fortran, few have formal training in computer science and therefore few have made the switch to OO programming. The continued use of procedural languages in education is due, in part, to the lack of up-to-date curricular materials that combine science topics with an OO framework. Although there are many good books for teaching computational physics, most are not object-oriented. The Open Source Physics project was initially intended to be a small object-oriented library that would provide input/output and graphing capabilities for the third edition of *An Introduction to Computer Simulation Methods* by Harvey Gould, Jan Tobochnik, and Wolfgang Christian. After showing this library to other educational software developers, it became apparent that it has a potentially wider audience than computational physicists. What is needed by the broader science education community is not a computational physics, numerical analysis, or Java programming book (although such books are essential for discipline-specific practitioners) but a synthesis of curriculum development, computational physics, computer science, and physics education that will be useful for scientists and students wishing to write their own simulations and develop their own curricular material. The Open Source Physics (OSP) project was established to meet this need.

Open Source Physics is a National Science Foundation-funded curriculum development project that seeks to develop and distribute a code library, programs, and examples of computer-based interactive curricular material. The goal of the project is to create and disseminate a large number of ready-to-run Java simulations for education using the GNU open source model for code distribution. The OSP project also maintains a web site that serves as in-depth guide to the tools, philosophy, and programs developed as part of this project.

<http://www.opensourcephysics.org>

The Open-Source Physics project is based, in part, on a collection of Java applets written by Wolfgang Christian known as *PhySlets*. Although PhySlets are written in Java, they are not open source. We often receive requests for PhySlet source code but have declined to distribute this code. PhySlets are compiled Java

## Chapter 1 Introduction to Open Source Physics

applets that are embedded into html pages and controlled using Java-Script. This paradigm works well for general-purpose programs such as a Newton's law simulation, but fails for more sophisticated one-of-a-kind simulations that require advanced discipline-specific expertise. Users and developers of these types of programs often have specialized curricular needs that can only be addressed by having access to the source code. However, anyone who has ever written a program in Java knows that writing code for opening windows and creating buttons, text fields, and graphs can be tedious and time consuming. Moreover, it is not in the spirit of object-oriented programming to rewrite these methods for each application or applet that is developed. The Open Source Physics project solves this problem by providing a consistent object-oriented library of Java components for anyone wishing to write their own simulation programs. In addition, the Open Source Physics library provides a framework for embedding these programs into an html page and for controlling these programs using JavaScript.

The basic Open Source Physics library includes the following packages:

- The `controls` package builds user interfaces and defines the OSP xml framework.
- `display` draws 2D objects and plots graphs on the `DrawingPanel` class using the `Drawable` interface.
- The `display2d` package displays two-dimensional data using contour and surface plots.
- The `display3d` package displays three-dimensional objects.
- The `ejs` package builds custom user-interface components.
- The `numerics` package contains numerical analysis tools such as ordinary differential equation solvers.
- The `tools` package contains small programs, such as data analysis programs, that collect data from other Open Source Physics programs. These tools are often added to a menu bar.

Open Source Physics libraries are based on Swing and Java 1.4. Although the focus of the basic libraries is traditional computational physics, they have already been extended to include topics not covered in computational physics texts such as video analysis. These packages are available from other OSP developers and from the OSP web site.

A number of curriculum authors have adopted OSP for their projects and have agreed to let the OSP project distribute their programs as examples. These projects include:

- *An Introduction to Computer Simulation Methods* (3 ed) by Harvey Gould, Jan Tobochnik, and Wolfgang Christian.
- *Statistical and Thermal Physics* by Harvey Gould and Jan Tobochnik.

1.3 Simulations and Pedagogy 3

- *Easy Java Simulations* high-level modeling tool by Francisco Esquembre.
- *Tracker* video analysis program by Doug Brown.

The development of these programs has enabled the Open Source Physics project to improve the core library in many ways. The API has been improved and made more consistent, bugs have been found and squashed, and we have learned what tools are useful to the developer community. But most importantly, the development of Tracker and EJS have resulted in great tools for the physics education community.

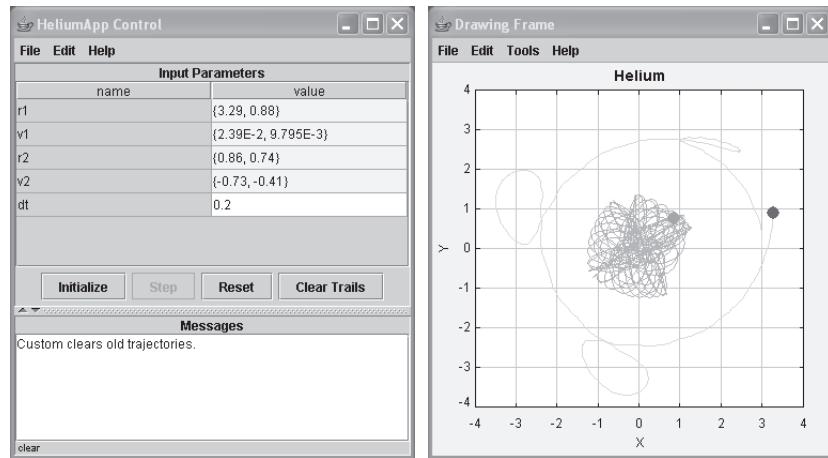
## 1.2 ■ MVC DESIGN PATTERN

Experienced programmers approach a programming project not as a coding task but as a design process. They look for data structures and behaviors that are common to other problems that they have solved. Separating the physics from the user interface and the data visualization is one such approach. In keeping with computer science terminology, we refer to the user interface as the *control*. It is responsible for handling events and passing them on to other objects. Plots, tables, and other visual representation of data are examples of *views*. Finally, the physics can be thought of as a *model* which maintains data that defines a state and provides methods by which that data can change. There is usually only one model and often only one control. It is, however, common to have multiple views. We could, for example, show a plot and a table view of the same data. Separating programs into models, views, and controls is known as the Model-View-Control (MVC) design pattern. The various OSP frameworks are designed to support this pattern.

The MVC design pattern is one of the most successful software architectures ever devised. Java is an excellent language with which to implement this pattern because it allows us to isolate the model, the control, and the view in separate classes. Object-oriented programming makes it easy to reuse these classes and easy to add new features to a class without having to change the existing code. These features are known as encapsulation, code reuse, and polymorphism. They are the hallmarks of object-oriented programming.

## 1.3 ■ SIMULATIONS AND PEDAGOGY

Good programming practice is best taught by having students modify, compile, test, and debug their own code. A typical exercise in *An Introduction to Computer Simulation Methods* begins with a discussion of theory, the presentation of an algorithm, and the necessary Java syntax. We implement the algorithm in a sample application and then ask the reader to test it with various parameters. The user then modifies the model, adds visualizations such as graphs and tables, and performs further analysis.



(a) A simple user interface.

(b) Helium electron trajectories.

**FIGURE 1.1** A classical model of two electrons orbiting about a nucleus. From the third edition of *An Introduction to Computer Simulation Methods* by Gould, Tobochnik, and Christian.

Consider the classical three-body model of Helium shown in Figure 1.1. The book narrative asks the reader to do the following:

- Modify the program `PlanetApp.java` to simulate the classical Helium atom. Let the initial value of the time step be  $\Delta t = 0.001$ . Some of the possible orbits are similar to those we have already studied in our mini-solar system.
- The initial condition  $r_1 = (1.4, 0)$ ,  $r_2 = (-1, 0)$ ,  $v_1 = (0, 0.86)$ , and  $v_2 = (0, -1)$  gives “braiding” orbits. Most initial conditions result in unstable orbits in which one electron eventually leaves the atom (autoionization). Make small changes in this initial condition to observe autoionization.
- The classical helium atom is capable of very complex orbits (see Figure 3). Investigate the motion for the initial condition  $r_1 = (3, 0)$ ,  $r_2 = (1, 0)$ ,  $v_1 = (0, 0.4)$ , and  $v_2 = (0, -1)$ . Does the motion conserve the total angular momentum?
- Choose the initial condition  $r_1 = (2, 0)$ ,  $r_2 = (-1, 0)$ , and  $v_2 = (0, -1)$ . Then vary the initial value of  $v_1$  from  $(0.6, 0)$  to  $(1.3, 0)$  in steps of  $v_x = 0.02$ . For each set of initial conditions calculate the time it takes for autoionization. Assume that ionization occurs when either electron exceeds a distance of six from the nucleus. Run each simulation for a maximum time

## 1.4 Using Launcher

5

equal to 2000. Plot the ionization time versus  $v_{1x}$ . Repeat for a smaller interval of v centered about one of the longer ionization times.

As the Helium example shows, the focus is on both the physics and programming. The user interface need not be very sophisticated, and a simple control that includes a few buttons and a table for data entry is all that is needed for a computational physics textbook. Modifications to the program, such as a custom user interface or web delivery, allow the program to be used with different pedagogies such as in-class demonstrations, Peer Instruction, traditional homework, and Just-in-Time Teaching. Because the code is released under a GNU GPL license, users can write their own narrative for other contexts such as astronomy and classical mechanics.

Another goal of the OSP project is to make Java simulations widely available for physics education. We do this by converting the applications such as those in the Java edition of *An Introduction to Computer Simulation Methods* into web-deliverable applets. Compiled applets, sample scripts, and source code are available on the OSP web server.

## 1.4 ■ USING LAUNCHER

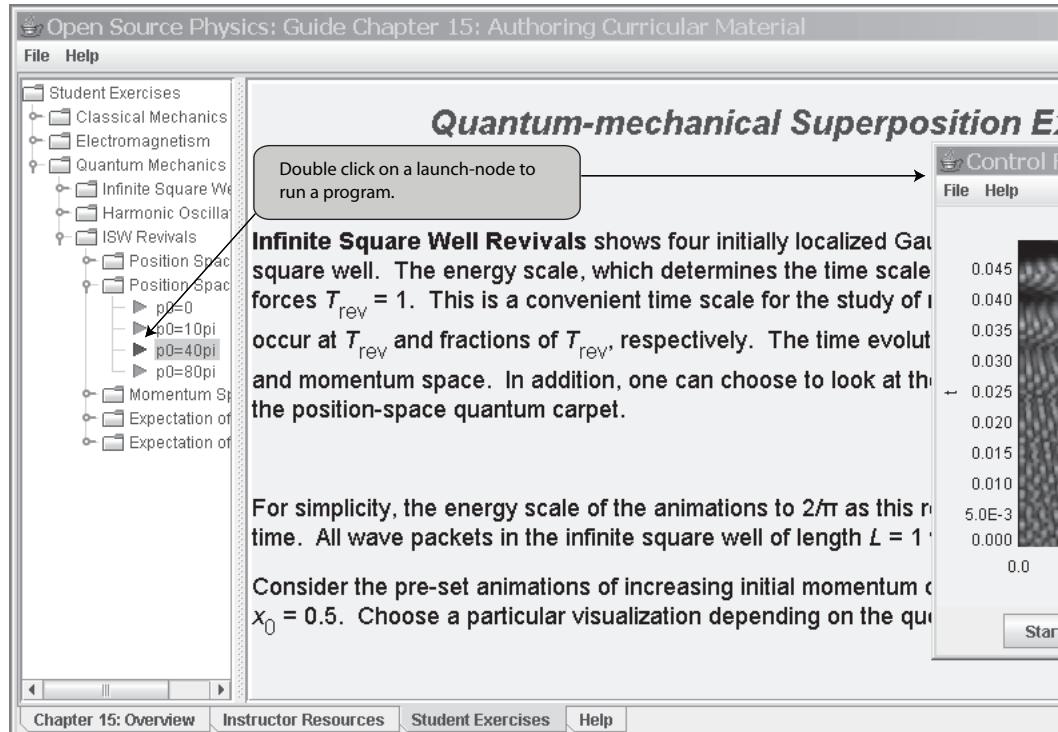
Java programs are usually distributed in archives that contain compiled code, resources such as images, security information, and a *manifest* that specifies the path to the class file containing the main method. These archives end with the extension \*.jar and are referred to as *jar* files. If a Java VM is installed on a computer and if a Java application has been properly packaged in a jar file, then the program specified in the jar file's manifest can be executed from the command line by passing the name of the jar file to the Java VM. (See Section 1.5 for Java installation instructions.) For example, the osp.jar archive can be executed by typing the following commands in the console (terminal) after the jar file has been downloaded from the OSP web site and copied into the apps directory. You will, of course, need to invoke the change directory cd /apps command (or cd \apps on Windows) to navigate to this directory.

```
1 cd /apps
2 java -jar osp-demo.jar
```

Operating systems with graphical user interfaces allow users to easily execute a jar file by double clicking on the jar's icon displayed within a filesystem browser. As far as the user is concerned, an archived Java program behaves like any other installed application.

If the jar file contains more than one program, the user can override the manifest's target class using other command line parameters. For example, the following command adds the osp-demo.jar archive to the *classpath* and executes the FirstPlotApp program in the archive.

```
1 java -classpath osp-demo.jar demo.FirstPlotApp
```



**FIGURE 1.2** The *Launcher* program executes other Java programs.

To execute the Helium example we copy the `osp_guide.jar` into the current directory and type:

```
1 java -classpath osp_guide.jar
2 org.opensourcephysics.manual.ch09.HeliumApp
```

Although it is possible to distribute programs in a single jar file or multiple jar files this, approach is not well suited for the distribution of curricular material. Few users want to deal with the complexities of command line syntax and resource management. Although jar files can contain resources such as html-based documentation, images, and sound, they are difficult to create and modify by casual users, particularly if the archive has been digitally signed. A large curriculum development project creates hundreds of programs and each program may be used in multiple contexts with different initial conditions. The *Launcher* and *Launch Builder* programs developed by Doug Brown at Cabrillo College solve many of these problems. *Launcher* shown in Figure 1.2 is a Java application that can launch (execute) other Java programs. We use *Launcher* to organize and distribute collections of ready to run programs, documentation, and curricular material in a single easily modifiable package.

## 1.5 Installing Java

7

Executing the `osp_demo.jar` archive starts an instance of *Launcher* containing curricular material described in Chapter 15. Double-clicking on a leaf-node in the Launcher’s left-hand “tree” launches a Java application. The right hand panel shows html-based documentation for the given program. Other Launcher packages are available on the CD and from the OSP website. The `osp_guide.jar` archive contains examples from this Guide and the `osp_csm.jar` file contains examples from *An Introduction to Computer Simulation Methods*.

The *Launcher*’s configuration, application data, and associated documentation are organized using an Extensible Markup Language (xml) text file named `launcher_default.xset`. Although this file is often packaged inside a Java archive, it can also be distributed as a stand-alone file.<sup>1</sup> *Launcher* uses the external configuration file if it is placed in the same directory as the jar file. Because this file can be modified without recompiling code or rebuilding the jar file, teachers and authors can easily adapt Open Source Physics curricular material for their own needs.

Although *Launcher* configuration xml files can opened with any text editor, they are most easily created and edited using the *Launch Builder* program as discussed in Chapter 15. The impatient reader can selecting the *Edit* menu item under the *Launcher* frame’s *File* menu to explore the capabilities of *LaunchBuilder*.

## 1.5 ■ INSTALLING JAVA

Java developers know how finicky Java can be. An incorrect installation, forgotten environment variables, or out of date libraries can cause pages of error messages. Many beginning programmers report that installing the Java Development Kit (JDK) and running their first program is the most difficult part of learning to use Java and Open Source Physics. Even if the the JDK is properly installed, finding a simple syntax error can be frustrating when using the Java compiler from he command line. For this reason, almost all Java developers use an integrated development environment such as *Eclipse* as described in Section 1.7. You can almost certainly begin Java programming by “double clicking” on file icons to install the JDK and Eclipse. But it is always useful to know what is going on under the hood and this section will help you debug your installation should something go wrong.

Java programs can be readily compiled and run on all standard operating systems. Although some operating system vendors, such as Apple, ship with Java pre-installed, users of most operating systems will need to install Java. Users wishing to only run programs should install the Java runtime environment but developers should install the complete Java Development Kit. Up-to-date versions of the JDK are available from Sun Microsystems for Solaris, Linux, and Windows at the following web site:

<http://java.sun.com/j2se/>

<sup>1</sup>Files in Java archives can be extracted and examined using the *jar utility* program from Sun or by using a decompression program such as *WinZip* or *StuffIt Expander*.

**TABLE 1.1** Java archive (jar) files available on the CD.

<b>Ready-to-run Open Source Physics programs.</b>	
<code>osp.jar</code>	Runs Launcher and provides access to LaunchBuilder using the edit menu item in Launcher. This archive contains the basic OSP library without any programs or curricular material. This jar is also used to show how to execute programs from the command line.
<code>osp_csm.jar</code>	Examples from the third edition of <i>An Introduction to Computer Simulation Methods</i> by Gould, Tobochnik, and Christian.
<code>osp_demo.jar</code>	A small collection of OSP-based physics curricular material developed by Mario Belloni and Wolfgang Christian at Davidson College for demonstrations and modification at faculty development workshops.
<code>osp_guide.jar</code>	Examples from this Guide.
<code>osp_stp.jar</code>	Statistical and thermal physics examples from Clark University and Kalamazoo College developed by Harvey Gould, Jan Tobochnik, and their students.
<code>tracker.jar</code>	Tracker video analysis program by Doug Brown.
<b>Optional libraries.</b>	
<code>osp_ode.jar</code>	High-order differential equation solvers developed by Andrew Gusev and Yuri B. Senichenkov.
<code>osp_jogl.jar</code>	A Java for Open GL (JOGL) implementation of the OSP 3D API developed by Glenn Ford.
<code>osp_media.jar</code>	Video tools developed by Doug Brown.

Look for version 5.0 or later and download the Java 2 Standard Edition (J2SE) in contrast to the Java 2 Enterprize Edition (J2EE) or the Java 2 Micro Edition (J2ME). Currently, you can also download an integrated development environment (IDE) named *NetBeans* which includes the JDK but this is not necessary. This IDE can be downloaded and installed at a later time from <http://netbeans.org/>. The *Eclipse* IDE is an alternative to *NetBeans* and is described in Section 1.7.

After downloading, double clicking on the downloaded archive (zip, gzip, or tar) in a filesystem browser is all that is required to install the JDK on most platforms. However, because Java has Unix origins, we recommend that users not install Java programs directories containing spaces in the file name. For example, Windows users may wish to install the jdk in `c:\jdk5.0` rather than the default `c:\program files\java\jdk5.0`.

## 1.5 Installing Java

9

**TABLE 1.2** Typical subdirectories within a JDK installation.

<b>JDK directory structure</b>	
bin	Compiler and other developer tools.
demo	Java demonstration programs from Sun.
docs	Java documentation from Sun.
include	Tools for compiling native methods.
jre	Java runtime environment files.
lib	Compiled libraries.
sample	Examples of Java-related programs and tools.
src	Source code extracted from <code>src.zip</code> .

The `jdk` directory contains the source code for the JDK in the `src.zip` archive. Because we will want to peek under the hood to see how Sun implements a particular class, it is a good idea to unpack the source code archive in the `jdk` directory. Create a directory named `src` and unpack the archive in this directory. You may also want to download and install Sun's Java documentation from <http://java.sun.com/docs>. A typical `jdk` directory structure is shown in Table 1.2.

In order to run and compile programs without typing the complete path to the programs in the `jdk` directory, it is convenient to add the `jdk/bin` directory to the search (execution) path by setting the *path* environment variable. This variable is set automatically by the JDK installer, but it can also be set from the console.

Setting environment variables is operating system dependent and you should consult your operating system manual to determine the proper syntax. On Unix/Linux environment variables are set by editing a user's environment file. On Windows XP environment variables are set using the advanced tab in the "System Properties" dialog box in the control panel. A good way to determine if the path contains the location of your JDK is to examine the current setting by entering the following in the command in the Windows console.

```
set path
```

You can modify the existing path by appending directories to the current path.

```
set PATH=%PATH%;c:\jdk1.5.0_04
```

If the Java VM is properly installed, you should now be able to open a console (terminal) window and type the `java` command to test the runtime installation.

```
java -version
```

The `javac` command tests the compiler in the JDK installation

```
javac -version
```

In a Windows console the `java -version` responds with output similar to the following:

```

1 Z:\>java -version
2
3 java version "1.5.0_04" Java(TM) 2 Runtime Environment,
4 Standard Edition (build 1.5.0_03-b07)
5 Java HotSpot(TM) Client VM
6 (build 1.5.0_04-b05, mixed mode, sharing)
7
8 Z:\>

```

Note that Windows uses the backslash character \ as a path delimiter while Unix and Linux use a forward slash /. Java applications recognize the slash / as a delimiter even if these applications are run on a Java VM in a Windows environment.

To further check the JDK installation, it is useful to compile and run a simple Java application. We follow tradition and create a `HelloWorldApp` test program. Create a project directory named `hello` and create two subdirectories named `src` and `classes`. Create a subdirectory named `hello` in `src`. Use a simple text editor to create `HelloWorldApp.java` code file shown in Listing 1.1 and save this file in the `hello` subdirectory. Note how the directory we have created matches the package declaration at the beginning of the code.

Listing 1.1 The `HelloWorldApp` is used to test the JDK installation.

```

1 package hello;
2 public class HelloWorldApp {
3     public static void main(String[] args) {
4         System.out.println("Hello World");
5     }
6 }

```

Compile the `HelloWorldApp` program by executing the `javac` command within a console after using the `cd` command to navigate to the project file.

```

1 cd \hello_project
2 javac -d classes/ src/hello/HelloWorldApp.java

```

Entering the `javac -help` command explains the various command line options including the `-d` option used above to specify where the compiler should place its output. The specified directory is the root of an output directory hierarchy that matches the package names. The compiler creates directories in this hierarchy automatically as needed.

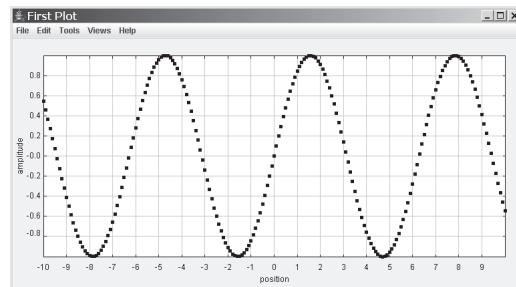
If you have not made any typing mistakes (Java is case sensitive), the compiler silently processes the `HelloWorldApp.java` file and produces the `HelloWorldApp.class` file in the `classes/hello` directory. You can now run the `HelloWorldApp` program using the `java` command.

```

1 java -classpath classes/ hello/HelloWorldApp

```

Again, entering the `java -help` command explains the various command line options such as `-classpath`.

**1.6 ■ PACKAGES****FIGURE 1.3** A simple Open Source Physics graph.

Because of the large number of Java source files, both Sun Microsystems and the Open Source Physics project organize code using Java packages. A Java package is a collection of related files located in a single directory. Code is given privileged access to other code within the same package because it is assumed that a package is designed and written by a programmer or by a small group of programmers who understand the code's interdependencies. Access to a class's data and methods from outside the package is far more restrictive and is only granted if the programmer has added either the `protected` or `public` keywords to a variable or method definition.

We often refer to a group of packages as a library and we refer to a group of files designed for a specific task as a framework. For example, the Open Source Physics core library contains a 3D drawing framework, and this framework uses files from multiple packages such as `ElementBox.java` in the `simple3d` package and `Transformation.java` in the `numerics` package as shown in the following code fragments.

```

1 // import a concrete implementation of OSP 3D
2 import org.opensourcephysics.display3d.simple3d.*;
3 import org.opensourcephysics.numerics.*;
4 // create a 3D world
5 DrawingPanel3D panel = new DrawingPanel3D();
6 DrawingFrame3D frame = new DrawingFrame3D(panel);
7 // place an Element into the 3D world
8 Element box = new ElementBox();
9 Transformation transformation =
10    Matrix3DTransformation.rotationX(Math.PI/6);
11 box.setTransformation(transformation);
12 panel.addElement(box);

```

The Open Source Physics library is organized in directories (packages) starting at the `org.opensourcephysics` root directory. Examples referred to in this

**TABLE 1.3** Packages (directories) within the core Open Source Physics library.

<b>org.opensourcephysics</b>	
display	A drawing framework based on the <code>Drawable</code> interface and the <code>DrawingPanel</code> class.
display2d	Visualization package for two-dimensional data such as contour and surface plots.
displayejs3d	A three dimensional modeling framework.
controls	A framework based on the <code>Control</code> interface for building graphical user interface, GUI, components.
numerics	A numerical analysis package containing tools such as Fourier analysis and ordinary differential equation, ODE, solvers.
tools	Small programs, such as video capture, that are designed to work with other OSP programs.
resources	Text files and configuration data.

Guide can be found in the `org.opensourcephysics.manual` directory. Open Source Physics programs written by students and faculty at Davidson College can be found in `org.opensourcephysics.davidson` and statistical and thermal physics programs from Clark University and Kalamazoo College can be found at `org.opensourcephysics.clark`.

Directories in the core Open Source Physics library are shown in Table 1.3. The compiled core library is available in a Java archive `osp.jar` on the CD and from the Open Source Physics website. Source code for the core library is available in the `osp_core.zip` file. Additional libraries, such as high precision differential equation solvers by Andrew Lvovitch Gusev and Yuri Senichenkov at Saint-Petersburg Polytechnic University, Russia, are available from the OSP website.

Although Java does not formally recognize the concept of a *subpackage* we often refer to subdirectories as such. Because it is awkward to refer to the code within the `org.opensourcephysics.display` package using the fully qualified package name, we often refer to a package using a single directory such as the `display` package when the complete path can be inferred from the context.

To compile and run an Open Source Physics program, first download the OSP core library archive `osp_core.zip` and place it in a project directory (folder) such as `osp_project`. Unpack the archive and notice the directory called `src` which in turn contains a directory called `org` which in turn contains a directory called `opensourcephysics` which in turn contains directories for the various packages. All directories except the `resources` directory contain code files. The `resources` directory contains non-code files such as images and text. Files with a `properties` extension contain text data such as button and menu labels to configure the user interface and operating system information such as default file paths. Placing non-code resources in a separate directory makes it easy to locate text

## 1.6 Packages

## 13

data and to internationalize the Open Source Physics library by translating this text into other languages. Because the Java compiler does not process text files, resource files must be copied to the output directory. In the `osp-project` directory create a new directory called `classes` which contains a directory called `org` which contains a directory called `opensourcephysics`. This directory is where the compiler will place the byte code (binary output). Copy the resources from the source `src/org/opensourcephysics/resources` to the output `classes/-org/opensourcephysics/resources` directory. We are now ready to test the core Open Source Physics library.

The project directory now contains two subdirectories: `src` and `classes`. Create the subdirectory named `plot` in `src`. Use a simple text editor to create `FirstPlotApp.java` code file shown in Listing 1.2 and save this file in the `plot` subdirectory. Figure 1.4 shows the package structure for this project.

Listing 1.2 The `FirstPlotApp` tests the Open Source Physics library installation.

```

1 package plot;
2 import org.opensourcephysics.frames.*;
3 import javax.swing.JFrame;
4
5 public class FirstPlotApp {
6     public static void main(String[] args) {
7         PlotFrame frame = new PlotFrame("position",
8                                         "amplitude", "First Plot");
9         frame.setSize(400, 400);
10        for(double x = -10, dx = 0.1;x<10;x += dx) {
11            frame.append(0, x, Math.sin(x));
12        }
13        frame.setVisible(true);
14        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15    }
16}
```

Open the console (terminal) application, change directory (`cd`) to the project directory, and compile the project by executing the `javac` command.

```
1 javac -d classes/ -sourcepath src/ src/plot/FirstPlotApp.java
```

Run the `FirstPlotApp` program by executing the `java` command.

```
1 java -classpath classes/plot/FirstPlot
```

A window similar to Figure 1.3 should appear if the OSP library is properly installed.

You can specify multiple code files in a single `javac` command but this is usually not necessary. The Java compiler is very helpful and automatically searches for and compiles classes referenced by our code. Examine the files produced in the output directory tree. Because the `FirstPlotApp` example references the

## Chapter 1 Introduction to Open Source Physics

PlotFrame class, this class and all dependent classes in the OSP library are automatically compiled.

Occasionally, classes are not compiled because they are not directly referenced in the code but these classes are later needed when a program is run, resulting in a *ClassNotFoundException* error message. If this runtime error occurs because the necessary class has not been compiled, go to the project directory and compile the entire package.

```
javac -d classes / org / opensourcephysics / packagename / *.java
```

Examples from the Open Source Physics Guide are distributed in the `osp-guide.zip` archive. The code is organized in packages named by chapter number and these packages are located in the `org.opensourcephysics.manual` package. Create a new project directory with `src` and `classes` subdirectories as before and download and unpack the examples into the `src` directory. You can copy the Open Source Physics core library into this project's source directory but this is not necessary. We now demonstrate how to compile and run programs using already compiled byte-code in the `osp.jar` file.

Open Source Physics programs are compiled and run without recompiling the core library by using a command line parameter that specifies the *classpath* to the byte code. Download and copy the `osp.jar` into the project directory as shown in Figure 1.5. We can now compile programs by adding this jar file to the classpath. For example, in order to compile and run the `SH03DApp` program in the `ch02` code package, we open the console and navigate to the Guide's project directory before executing the following commands.<sup>2</sup>

```
-classpath osp.jar src / org / opensourcephysics / manual / ch02
```

The compiler assumes that the `osp.jar` file is in the console's current directory and that the subdirectory structure for the examples has a directory structure that matches the java package names. Although locations of the source and output files can be overridden with command line options, it is best to keep things simple and locate everything in a single project directory when starting out.

To make it easy to identify OSP programs, we have adopted the convention that classes that contain a *main* method and can therefore be run as a Java application end with *App*. For example, the ubiquitous “HelloWorld” program was defined in a file named `HelloWorldApp.java`. This convention makes it easy to find example programs in the Guide by searching for files matching the pattern `**App.java`.

## 1.7 ■ ECLIPSE

The command line examples we have shown are simple and useful as a starting point for building shell (command) scripts for compiling and running Java

<sup>2</sup>Some operating systems use a colon rather than a semicolon to delimit the search path.

## 1.7 Eclipse

15

**TABLE 1.4** Code archives (zip) files available on the CD and on the Open Source Physics website.

<code>osp_core.zip</code>	The basic Open Source Physics library without any programs or curricular material.
<code>osp_csm.zip</code>	Examples from the third edition of <i>An Introduction to Computer Simulation Methods</i> by Gould, Tobochnik, and Christian.
<code>osp_guide.zip</code>	Examples from this Guide.
<code>jep.zip</code>	Java expression parser by Nathan Frank.
<code>osp_ode.zip</code>	High-order differential equation solvers developed by Andrew Gusev and Yuri. B. Senichenkov.
<code>osp_jogl.zip</code>	A Java for Open GL (JOGL) implementation of the OSP 3D API developed by Glenn Ford.
<code>osp_media.zip</code>	Video tools developed by Doug Brown.

programs. Large development projects, however, are more efficiently managed using an integrated development environment (IDE) such as *Eclipse*, Borland's *JBuilder*, Sun's *NetBeans*, or Apple's *XTools*. Good IDEs provide color-coded syntax highlighting and syntax checking as well as access to tools such as the `jar` archive builder and the `javadoc` documentation generator. They can easily incorporate the entire OSP source code library thereby allowing debugging and single stepping through an entire project. And most importantly, an IDE provides easy access to documentation. Highlighting a method or variable name and right-clicking (control or option clicking in some operating systems) takes the user directly to the source code and documentation for the given object or variable.

Eclipse is available for Mac OS X, Windows, Linux, and Solaris. Although it supports multiple programming languages, it is written in Java and is an excellent Java IDE. Eclipse is open source and is freely available from <http://eclipse.org>.

Although it takes time to fully master Eclipse, it is not difficult to get started. An Eclipse *project* organizes the code and resources needed to compile, run, and distribute a program. Download and install Eclipse and then select the *create project* item from the file menu after starting the Eclipse application as shown in Figure 1.6. Select the check box to use separate directories for the source code and the compiler output and enter the name of the project and the location of the source code and the compiler output. We typically use `src` and `classes` for the names of the the output directories.

Although it is possible to copy the Open Source Physics code library into the source directory, it is preferable to use the import menu item to obtain the source code. Using the Eclipse menu copies the source code into the correct directory and insures that internal variables are up to date. You should press the *F5* function key to refresh the Eclipse display if you copy the source code directly.

Eclipse displays the project in a graphical interface known as the *Eclipse*

*workspace* shown in Figure 1.7. Source code is accessed using the tree-like *package explorer* in the left-hand side of the workspace. Expand a node in the Eclipse package explorer by double clicking and select a class definition containing a Java application. The application's code appears in an *editor* pane near the center of the workspace. Run the application using either the run menu or the tool-bar button with the white triangle inside the green circle.

When you edit code in the Eclipse editor, an asterisk appears in front of the file name in the title bar indicating that the code has not yet been saved. If the *Build Automatically* option under the project menu is selected, the edited code is automatically compiled when the code is saved using either the file menu or the standard <control>-s keyboard shortcut. Errors, warnings, and program output are displayed in a tabbed view near the bottom of the workspace.

## 1.8 ■ BUILDING PROGRAMS WITH ANT

*Ant* is an open source alternative to generic shell scripts and a powerful complement to Java IDEs such as Eclipse. Version 3.1 of Eclipse comes with Ant 1.6.5, and there is an extensive Ant interface in Eclipse. To understand how Ant operates within Eclipse, it is useful to learn how to run Ant from the command line.

*Ant* is a Java application that uses Extensible Markup Language (xml) to build Java projects. Ant invokes other programs known as tasks and the most common Ant task is, of course, to compile and run Java programs using the Java compiler javac and the Java runtime environment java. But there are other tasks such as creating documentation using JavaDoc, creating Java Archive (jar) files, signing jar files with security keys, and deploying applications to Web servers. Because *Ant* does this and much more and because *Ant* is platform independent, we recommend that developers distribute an *Ant* build file with their projects even if they use other development environments for their day-to-day work. Although it is not our intention to teach *Ant* programming, a basic build file shown in Listing 1.3 is not difficult to decipher. The latest *Ant* distribution and online documentation is available at <http://ant.apache.org>.

*Ant* takes its instructions from a build file that contains one or more *targets*. Each target has a name such as *compile* or *run*. If *Ant* is properly installed, you execute a target by passing the target's name to *Ant*. For example, the command line ant *compile* executes the *compile* target in the default build file, *build.xml*. An *Ant* file that compiles and runs an example is shown in Listing 1.3. Note how we include the *osp.jar* archive in the *classpath*.

Listing 1.3 A simple Ant build file for compiling and running an OSP program. The default Ant build file is named *build.xml*.

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <project name="osp" default="all" basedir=".">
3   <property name="author" value="W. Christian"/>
4   <property name="classes.dir" value=".//classes"/>
```

## 1.8 Building Programs with Ant

17

```

5   <property name="srcdir.dir" value=".src"/>
6   <property name="lib.dir" value=".lib"/>
7   <path id="run.class.path">
8     <pathelement location=".classes"/>
9     <pathelement location="${lib.dir}/osp.jar"/>
10    </path>
11
12  <!--compiles the source files-->
13  <target name="compile" >
14    <delete dir="${classes.dir}"/>
15    <mkdir dir="${classes.dir}"/>
16    <javac srcdir="${srcdir.dir}"
17           destdir="${classes.dir}"
18           classpath="${lib.dir}/osp.jar" />
19  </target>
20
21  <!--runs the PlotterApp-->
22  <target name="run" depends="compile">
23    <echo>Running Program.</echo>
24    <java
25      classname="plot.FirstPlotApp"
26      classpathref="run.class.path"
27      fork="true">
28    </java>
29  </target>
30 </project>

```

Targets within the build file contain tasks that are executed by *Ant*. These tasks typically invoke programs that take input and produce output. The `compile` target executes a task that invokes the Java compiler `javac` to generate class files from Java source files. The `compile` target shown in Listing 1.3 executes two additional (nested) tasks to delete and make the output directory.

Names, such as the locations of source and output directories, are usually defined near the beginning of an ant file using *property* variables. These variables are referenced by enclosing the property name with a dollar sign and braces. Defining properties makes it easy to adjust build files to match the local installation. Note how the path to the `osp.jar` library is specified.

*Ant* tasks often depend on the successful completion of one or more previous targets. Adding the `depends="compile"` attribute to the `run` target insures that the Java code is compiled before the program is executed. This compilation is done automatically and it is not necessary to invoke the `compile` target first. The user simply types

```
ant run
```

to compile and run the `PlotterApp` program from the build file.

An *Ant* build file to manage the entire Open Source Physics library is available on the OSP website. The `build_osp.xml` file defines targets to compile the code

and to create the documentation and the `osp.jar` archive. Entering the following command line executes a lengthy task that removes old files and produces a clean copy of the core OSP library. You should edit the build file's property values to match the directory structure and Java version on your computer.

```
ant -buildfile build\osp.xml all
```

You can import the `build_osp.xml` file into an Eclipse project and execute targets from within the workspace as shown in Figure 1.8. An icon for the Ant build file appears in the Package Explorer and the xml code is displayed in a color coded editor view to highlight declarations, attributes, and keywords. Eclipse catches syntax errors when you edit the build file and provides an explanation of the error. Clicking on an Ant target in the outline view executes that target and sends output to the Eclipse Console view.

Other Ant IDEs exist, but the big players are behind Eclipse and support for Ant in Eclipse is extensive and robust. The combination of Ant and Eclipse makes an ideal development environment for Java programmers.

## 1.9 ■ DOCUMENTATION

Although Java code should be documented using `javadoc` comments, we assume that readers can access OSP code and documentation in an IDE and have therefore removed most block comments from this Guide in order to compact the listings.<sup>3</sup> Complete *Java Documentation Generator* output is available on the Guide's CD and on the OSP web site. This official OSP documentation is generated using a custom `javadoc` add-on *doctlet* that creates Universal Modeling Language *UML* diagrams as shown in Figure 1.9.

## 1.10 ■ PROGRAMS

### **FirstPlotApp**

`FirstPlotApp` in the `plot` package tests the Open Source Physics library by creating a plot of the sine function. See Section 1.6.

### **HeliumApp**

`HeliumApp` described Section 1.3 in discussed Chapter 9.

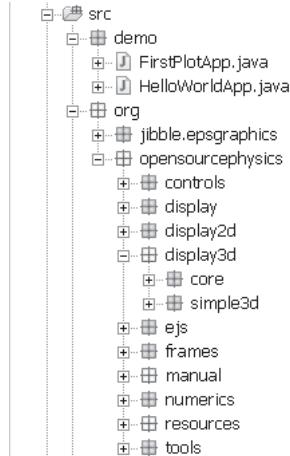
### **HelloWorldApp**

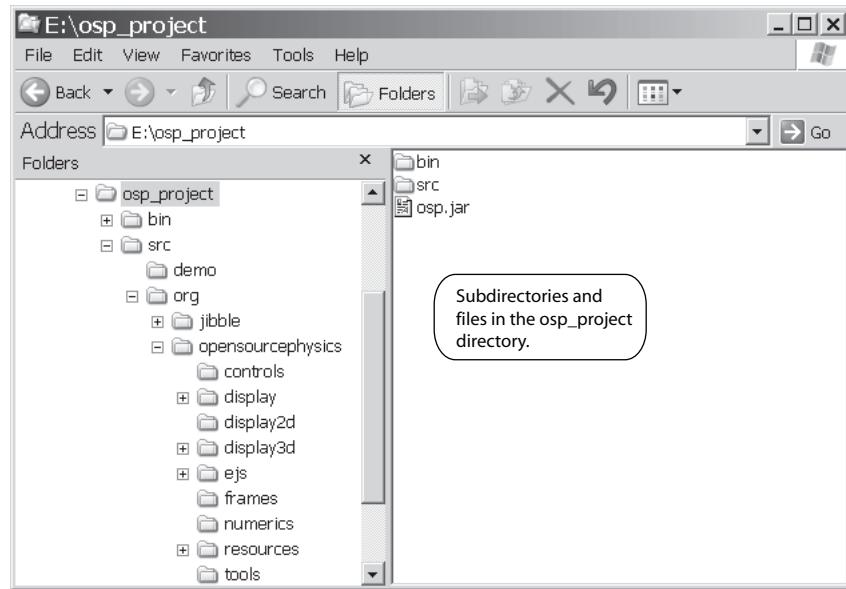
`HelloWorldApp` in the `hello` package tests the Java installation by printing a string in the console. See Section 1.5.

<sup>3</sup>The Java 2 Software Development Kit (J2SDK) provides a Java API documentation generator known as `javadoc`. The `javadoc` program generates html-based documentation by processing the Java source code and extracting information such as comments, method names, and class variables.

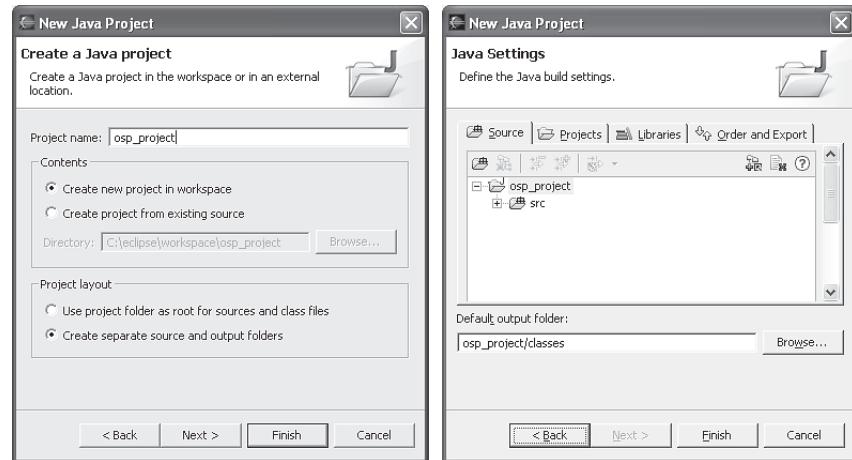
## 1.10 Programs

19





**FIGURE 1.5** A typical directory structure for compiling programs without recompiling the core library.



(a) Creating a project.

(b) Setting project directories.

**FIGURE 1.6** Dialog boxes guide the user through the process of creating a project in Eclipse.

## 1.10 Programs

21

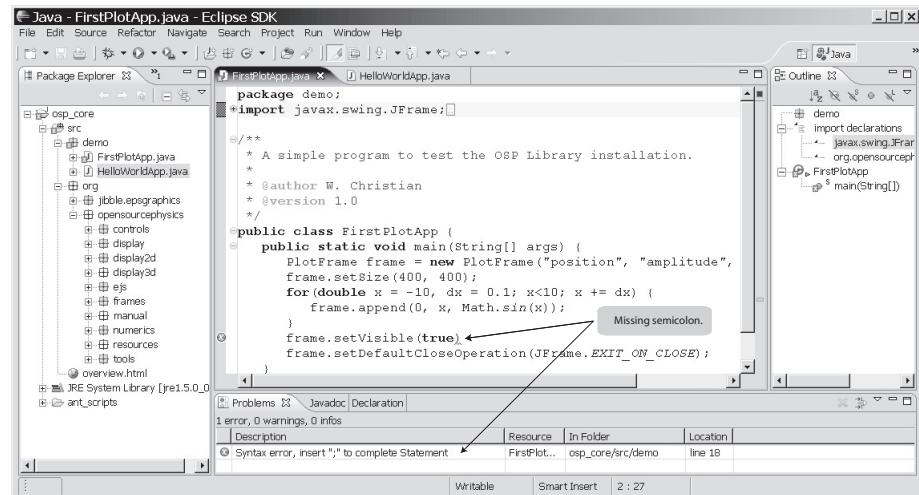


FIGURE 1.7 The Eclipse development environment.

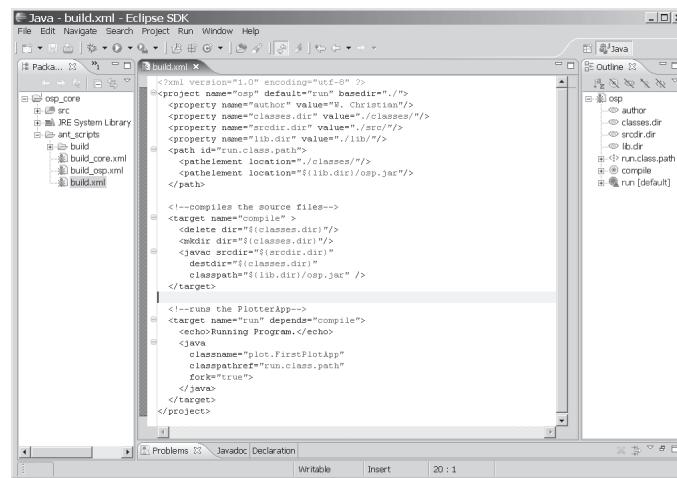
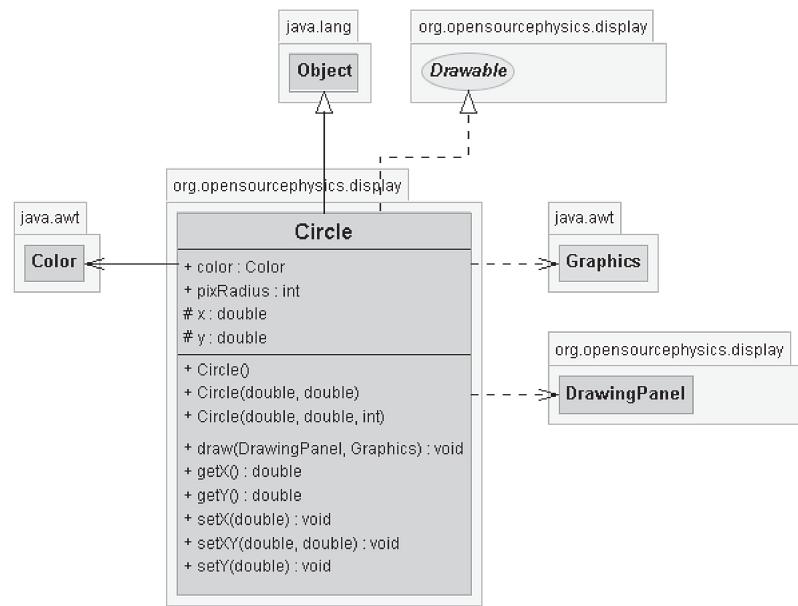


FIGURE 1.8 Eclipse has well-integrated and extensive support for Ant.



**FIGURE 1.9** A Universal Modeling Language (UML) diagram of the circle class in the display framework.

## CHAPTER

## 2

## A Tour of Open Source Physics

©2005 by Wolfgang Christian

The Open Source Physics project provides high-level components for drawing, numerical analysis, and user interfaces. This chapter provides an overview of these capabilities.

### 2.1 ■ OBJECT ORIENTED PROGRAMMING

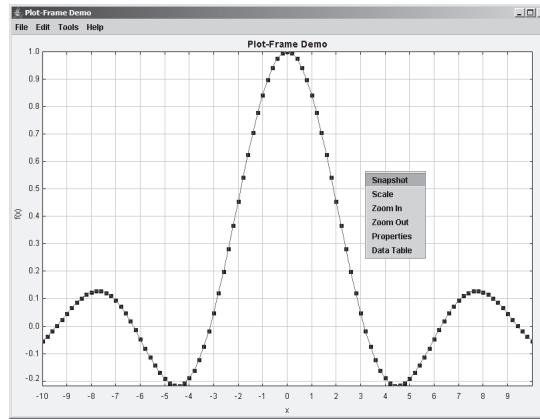
Object-oriented programming (OOP) enables us to represent the real world using a computer. We are fortunate that in physics, computer objects can more nearly reproduce real-world behavior than in most other disciplines. This chapter presents an overview of some important *abstractions* in the Open Source Physics library by building a function plotter and by modeling simple phenomena using first-order differential equations. These examples make use of functions, differential equation solvers, and strip-chart plots without worrying how they are implemented. Because our programs frequently make use of graphical user interface (GUI) components such as buttons, text fields, and check boxes, we introduce *Easy Java Simulations* (EJS) components. EJS components can be used without being an expert in the Java Swing library. You do not, for example, need to know much about a Java JButton to understand that the following statement creates a button and adds it to a graphical user interface named gui.

```
gui.add ("Button", "parent=inputPanel; text=Plot;
action=doPlot");
```

The button displays the text label Plot and (without additional programming) invokes the program's doPlot method when it is pressed.

### 2.2 ■ OSP FRAMES

We have come to expect programs with attractive graphical user interfaces that support common tasks such as printing, disk access, and copy-paste data exchange with other applications. These features can be implemented using packages in the Java Development Kit (JDK), but it usually requires much programming and a good knowledge of the Java API. For example, adding a “print” menu item requires importing eight classes in three different packages. Learning the API is essential for software developers but even experienced programmers cut and



**FIGURE 2.1** OSP frames, such as the `PlotFrame` shown here, are composite objects that contain a menu bar and a pop-up menu that allow users to access the frame's data.

paste this type of boilerplate code. To enable users to quickly begin writing their own programs while they are learning the core Java and the Open Source Physics APIs and in order to provide examples of how these APIs should be used, we have defined a collection of high-level objects in the frames package for routine data visualization tasks<sup>1</sup>. These frames inherit from the `JFrame` class in the Java `javax.swing` package and add the functionality needed to plot datasets, display vector and scalar fields, and do numerical analysis such as fast Fourier transformations.

One such frame is the `PlotFrame` class that plots points in one or more datasets. A `Dataset` object contains an array of points and drawing attributes such as color and marker shapes. Data points are added directly to a `PlotFrame` using the `append` method and the frame automatically creates the necessary `Dataset` object as shown in Listing 2.1. The first parameter in the `append` method identifies the dataset. The second and third parameters specify the point's coordinates. The `append` method has other signatures that accept arrays of data points and data points with error estimates.

Listing 2.1 The `PlotFrameApp` program displays an x-y plot of the  $\sin x/x$  function.

```

1 package org.opensourcephysics.manual.ch02;
2 import org.opensourcephysics.frames.PlotFrame;
3 import javax.swing.JFrame;
4
5 public class PlotFrameApp {
6     public static void main(String[] args) {

```

<sup>1</sup>An application programming interface (API) define how a piece of software is used. In other words, an API specifies how a method (subroutine) is invoked (called) and what it does.

## 2.3 Function Plotter

25

```

7  PlotFrame frame = new PlotFrame("x", "f(x)", "Plot-Frame"
8      Demo");
9  frame.setConnected(true); // sets default to connect dataset
10     points
11  frame.setXYColumnNames(0, "x",
12      "sin(x)/x");           // sets names for first dataset
13  double dx = 0.2;
14  for(double x = -10;x<=10;x += dx) {
15      frame.append(0, x, Math.sin(x)/x);
16  }
17  frame.setVisible(true);
18  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19 }
20 }
```

OSP frames, such as the `PlotFrame` in the example, are composite objects with a great deal of functionality. Right (control)-clicking within the display area shows a pop-up menu. The default menu allows users to create a snapshot (a gif image) of the frame's contents, set the scale, zoom, and examine the frame's internal data. The Data Table item displays data points in a Java `JTable` and the Properties item presents a xml tree view of the frame and its contents. (See Chapter 12 for an introduction to xml.)

OSP frames have a menu bar that gives additional options. A typical menu bar has a *file* menu that allows users to print, save, and inspect the contents of the frame; an edit menu allows users to cut and paste between OSP frames; and a tools menu that contains links to other OSP programs such as data analysis tools.

### 2.3 ■ FUNCTION PLOTTER

We now use the `PlotFrame` introduced in Section 2.2 to build a simple function plotter with a graphical user interface. This program will be assembled using components from multiple packages and is outlined in Listing 2.2. It builds a custom user interface using components from the `ejs` package and draws the plot using components from the `display` package. Mathematical abstractions, such as a function  $f(x)$ , are defined in the `numerics` package.

Listing 2.2 A function plotter program outline.

```

1 import org.opensourcephysics.display.*;
2 import org.opensourcephysics.ejs.control.*;
3 import org.opensourcephysics.numerics.*;
4
5 public class PlotterApp {
6     PlottingPanel plot =
7         new PlottingPanel("x", "f(x)", "y = f(x)");
8     Dataset dataset = new Dataset(); // (x,y) data
9     EjsControl gui; // graphical user interface
```

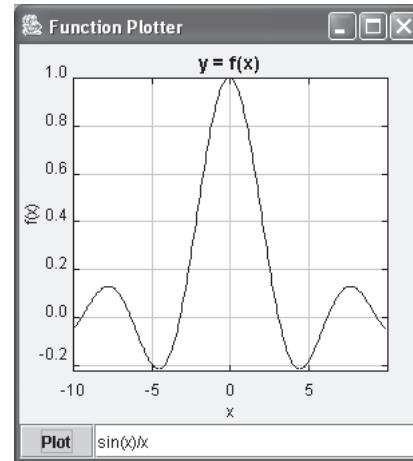


FIGURE 2.2 A function plotter with a custom interface.

```

10 Function function; // function that will be plotted
11
12 public PlotterApp() {
13     dataset.setConnected(true); // connect with lines
14     dataset.setMarkerShape(Dataset.NOMARKER);
15     plot.addDrawable(dataset); // add dataset to panel
16     buildUserInterface(); // creates user interface
17     doPlot(); // plots default function
18 }
19
20 public void doPlot() {
21     // code omitted for compactness
22 }
23
24 void buildUserInterface() {
25     // code omitted for compactness
26 }
27
28 public static void main(String[] args) {
29     new PlotterApp(); // creates program
30 }
31 }
```

The PlotterApp program creates a plotting panel with cartesian axes, a dataset to store  $(x, y)$  points, and identifiers for the user interface and the function. The function and the user interface are created later in the doPlot and buildUserInterface methods, respectively. The program's constructor sets various display parameters, creates the PlottingPanel object, and adds the Dataset object to the panel so that it can be drawn. It then builds the graphical

## 2.3 Function Plotter

27

user interface and plots a default function. The static main method instantiates the application.

The doPlot method shown below does what you would expect. It reads a string from the user interface and attempts to convert the string's characters into a mathematical function using a parser (see Section 10.4). If a user mistypes a character or enters an invalid function, the function is set to zero using a utility class in the numerics package. The program then clears the old data, reads the domain of the independent variable, and evaluates the function at a suitable number of points. Although the plot's default  $x$ -scale is  $[-10, 10]$ , this range can be changed by the user at runtime by right (control)-clicking within the plot.<sup>2</sup> Setting the scale and other routine operations are implemented in the plotting panel.

```

1  public void doPlot(){
2      try{ // read input and parse text
3          function =
4              new ParsedFunction(gui.getString("fx"), "x");
5      }catch(ParserException ex){ // input errors are common
6          // set f(x)=0 if there is an error
7          function= Util.constantFunction(0);
8      }
9      dataset.clear(); // removes old data
10     double xmin=plot.getXMin(), xmax=plot.getXMax();
11     double dx=(xmax-xmin)/200;
12     for(double x=xmin; x<=xmax; x+=dx){ // loop generates data
13         double y=function.evaluate(x);
14         dataset.append(x,y);
15     }
16     plot.repaint(); // new data so repaint
17 }
```

Although the Open Source Physics library contains a number of simple general purpose graphical user interfaces, it also defines tools that enable us to quickly design and create a custom user interface. The buildUserInterface method constructs a custom user interface using the EJS framework developed by Francisco Esquembre at the University of Murcia, Spain. The complete *Easy Java Simulations* (EJS) program is a high-level modelling tool that builds Java applications and applets using a drag and drop metaphor. The core OSP library includes a subset of this framework. The buildUserInterface method in the function plotter example creates a user interface consisting of a text field and a button as shown in Figure 2.2.

EJS builds controls without any preconditions or assumptions as to the type of object that will be controlled. The user invokes methods in the PlotterApp because a reference to the model this is passed to the EjsControl constructor.

```
1 void buildUserInterface() {
```

<sup>2</sup>Users with a one-button mouse should press the Control, Alt, or Option key while clicking to modify mouse actions.

```

2   gui = new EjsControl(this); // Ejs user interface
3   gui.add("Frame", "name=controlFrame;title=Plotter;
4     layout=border; exit=true;size=pack");
5   gui.addObject(plot, "Panel", "name=contentPanel;
6     parent=controlFrame; position=center");
7   gui.add ("Panel", "name=inputPanel; parent= controlFrame;
8     layout=hbox; position=south");
9   gui.add ("Button", "parent=inputPanel; text=Plot;
10    action=doPlot;");
11  gui.add ("TextField", "parent=inputPanel;variable=fx;
12    value=sin(x); size=125,15");
13 }
```

EJS supports a wide variety of user interface components including labels, radio buttons, check boxes, and text fields. These components are created in a script-like style by invoking the control's add method using the following signature:

```
1 add(String type, String property_list); .
```

The add method's first parameter specifies the type of object that is being created. In the plotter program we create a frame, a button, and a one-line text field. The second parameter is a semicolon delimited list of properties. These properties depend on the type of object being created. For example, the button's action property specifies the method in the plotter program that will be invoked. The text field's variable property, fx, is used later in the doPlot method to read the function string. Note that the ejs package provides a convenient way to add the preexisting plot component to an EjsControl using the addObject method.

The function plotter program makes use of three core OSP packages. The display package contains the definition of the Dataset class and the PlottingPanel class. The numerics package defines the Function interface and the ParsedFunction class. The controls package defines the Control interface and the ejs.control package provides a concrete implementation of this interface. These packages must, of course, be imported into the program.

```

1 import org.opensourcephysics.display.*;
2 import org.opensourcephysics.numerics.*;
3 import org.opensourcephysics.ejs.control.*;
```

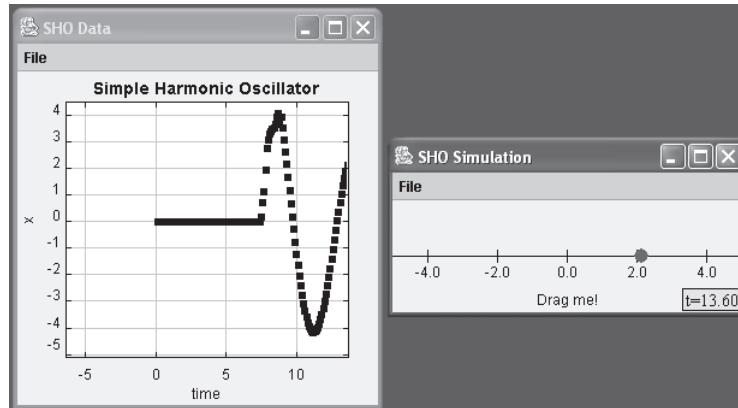
The listing for the entire function plotter program can be found in the `org.opensourcephysics.manual.ch02` package.

## 2.4 ■ SHO

The simple harmonic oscillator (SHO) occurs in many different educational contexts. Our simulation of this phenomenon requires two classes SHO and SHOApp to define and to display the evolution of the model, respectively. The SHOApp class

## 2.4 SHO

29



**FIGURE 2.3** A simple harmonic oscillator simulation with a position graph.

constructs the object that draws a picture of the oscillating body, the object that plots position as a function of time, and the object that contains the dynamical equations of motion. The `SHO` class shown in Listing 2.3, contains the physics.

Listing 2.3 The `SHO` class defines the dynamical equations of motion for a damped simple harmonic oscillator.

```

1 package org.opensourcephysics.manual.ch02;
2 import org.opensourcephysics.display.InteractiveCircle;
3 import org.opensourcephysics.numerics.*;
4
5 public class SHO extends InteractiveCircle implements ODE {
6     // initial state values = {x, v, t}
7     double[] state = new double[] {0.0, 0.0, 0.0};
8     double k = 1;    // spring constant
9     double b = 0.2; // damping constant
10    ODESolver ode_solver = new RK4(this);
11
12    public double getTime() {
13        return state[2];
14    }
15
16    public double[] getState() {
17        // insure that the state matches the screen position
18        state[0] = getX();
19        return state;
20    }
21
22    public void getRate(double[] state, double[] rate) {
23        rate[0] = state[1]; // dx/dt = v
24        double force = -k*state[0]-b*state[1];

```

```

25     rate[1] = force; // dv/dt = force
26     rate[2] = 1;      // dt/dt = 1
27 }
28
29 public void setXY(double x, double y) {
30     super.setXY(x, 0); // y is always zero
31     state[0] = x;
32 }
33
34 public void stepTime() {
35     ode_solver.step();
36     setX(state[0]);
37 }
38 }
```

The SHO class extends `InteractiveCircle` so that it can respond to mouse actions and can paint itself as a circle on the computer screen. A user just drags the circle to set the oscillator's initial position. Because mouse actions and painting are defined in the oscillator's *superclass*, we need not concern ourselves with the details here. It is sufficient to know that the oscillator inherits this capability.

The oscillator's dynamics is implemented via the `ODE` interface to define a system of first-order ordinary differential equations. The OSP numerics package defines a number of differential equation solvers that make use of the `ODE` interface. In this example, we instantiate a fourth-order Runge-Kutta solver to advance the dynamical system using variables that are stored in the `state` array. The spring and damping constants are parameters that do not evolve and are therefore not included in this array. Because a user of the `SHO` class may not know the order of the variables in the array, we implement the `getTime` convenience method for clarity. The `ode` solver calls the oscillator's `getState` and `getRate` methods as needed when the solver's `step` method is invoked. In the spirit of object-oriented programming, we again do not concern ourselves with the details. We do assume that the differential equation solver has implemented the numerical method correctly.

The `SHOApp` class defines a concrete implementation of an `AbstractAnimation` by implementing the `doStep` method. It also creates the necessary drawing and plotting panels to show the oscillator and the generated data. Drawing and plotting follow similar paradigms. Panels are created and added to frames. In the constructor objects such as the oscillator and the strip-chart dataset are added to the panels. The only requirement is that these objects implement the `Drawable` interface defined in the `display` package.

The `startAnimation` method in the application's `AbstractAnimation` superclass creates a `Thread` that invokes the `doStep` method approximately ten times per second. In this example, the `doStep` method advances the oscillator's state by stepping a differential equation. It then retrieves the position and time values and appends a data point to the strip-chart dataset. The drawing and the plot are then repainted because objects within these panels have changed state.

**Listing 2.4** The `SHOApp` class creates the objects needed for a simple harmonic

## 2.4 SHO

31

oscillator simulation.

```

1 package org.opensourcephysics.manual.ch02;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.*;
4 import org.opensourcephysics.display.axes.XAxis;
5
6 public class SHOApp extends AbstractAnimation {
7     PlottingPanel plot = new PlottingPanel("time", "x",
8                                         "Simple Harmonic Oscillator");
9     DrawingFrame plottingFrame = new DrawingFrame("SHO Data",
10                                                plot);
11    DrawingPanel drawing = new InteractivePanel();
12    DrawingFrame drawingFrame = new DrawingFrame("SHO Simulation",
13                                                 drawing);
14    Dataset stripChart = new Stripchart(20, 10); // strip chart of
15    x(t)
16    SHO sho = new SHO();                                // simple
17    harmonic oscillator
18
19    public SHOApp() {
20        drawing.setPreferredMinMax(-5, 5, -1, 1);
21        drawing.addDrawable(new XAxis("Drag me!"));
22        drawing.addDrawable(sho);
23        drawingFrame.setSize(300, 150);
24        drawingFrame.setVisible(true);
25        drawingFrame.setDefaultCloseOperation(
26            javax.swing.JFrame.EXIT_ON_CLOSE);
27        plot.addDrawable(stripChart);
28        plottingFrame.setLocation(400, 300);
29        plottingFrame.setVisible(true);
30        plottingFrame.setDefaultCloseOperation(
31            javax.swing.JFrame.EXIT_ON_CLOSE);
32    }
33
34    protected void doStep() {
35        sho.stepTime();
36        stripChart.append(sho.getTime(), sho.getX());
37        drawing.setMessage("t=" + decimalFormat.format(sho.getTime()));
38        drawing.repaint();
39    }
40
41    public static void main(String[] args) {
42        new SHOApp().startAnimation();
43    }
44}

```

## 2.5 ■ THREE-DIMENSIONAL FRAMEWORK

The Java 2D drawing API is designed to represent and manipulate two-dimensional objects. The `DrawingPanel` and `PlottingPanel` classes assume this model. However, the physical world is three-dimensional, and we have therefore defined a number of high-level abstractions for manipulating 3D models. Listing 2.5 shows that it is not much more difficult to define and manipulate a three-dimensional ball than a two-dimensional ball using the `display3d.simple3d` package. The most significant change is that the program instantiates a `DrawingPanel3D` and adds `Element` objects to this panel.

The `Element` interface defines objects that can be added to a `Display3DFrame`. This interface contains methods that allow us to work with elements in a way similar to the way that we work with physical objects. Using concrete implementations of `Element` in the `display3d.simple3d` package, we can create and manipulate objects such as arrows, cylinders, and ellipsoids in space. The OSP 3D API defines objects that:

- have a position, and size in space,
- have visual properties such as color,
- can change their visibility status,
- post interaction events and can have one or more interaction targets,
- and can be grouped with other elements.

Listing 2.5 Particles can be displayed in three dimensions using the `display3d.simple3d` package.

```

1 package org.opensourcephysics.manual.ch02;
2 import org.opensourcephysics.controls.AbstractSimulation;
3 import org.opensourcephysics.display3d.core.Camera;
4 import org.opensourcephysics.display3d.core.Resolution;
5 import org.opensourcephysics.display3d.simple3d.*;
6 import org.opensourcephysics.ejs.control.EjsControl;
7 import org.opensourcephysics.frames.Display3DFrame;
8 import java.awt.Color;
9
10 public class Ball3DApp extends AbstractSimulation {
11     EjsControl gui;
12     Display3DFrame frame = new Display3DFrame("3D Ball");
13     Element ball = new ElementEllipsoid();
14     double
15         time = 0, dt = 0.05;
16     double vz = 0;
17
18     public Ball3DApp() {
19         frame.setPreferredMinMax(-5.0, 5.0, -5.0, 5.0, 0.0, 10.0);

```

## 2.5 Three-Dimensional Framework

## 33

```

20    ball.setZ(9);
21    ball.setSizeXYZ(
22        1, 1,
23        1); // The bob will be displayed in 3D as a planar
24        ellipse of size (max(dx,dy),dz)
25    frame.addElement(ball);
26    Element block = new ElementBox();
27    block.setXYZ(0, 0, 0);
28    block.setSizeXYZ(4, 4, 1);
29    block.getStyle().setFillColor(Color.RED);
30    block.getStyle().setResolution(new Resolution(5, 5,
31        2)); // divide the block in subblocks.
32    frame.addElement(block);
33    buildUserInterface();
34}
35
36protected void doStep() {
37    time += dt;
38    double z = ball.getZ() + vz * dt - 4.9 * dt * dt;
39    vz -= 9.8 * dt;
40    if (vz < 0 && z < 1.0) {
41        vz = -vz;
42    }
43    ball.setZ(z);
44    frame.setMessage("t=" + decimalFormat.format(time));
45}
46
47public void set3d() {
48    if (gui.getBoolean("3d")) {
49        frame.setProjectionMode(Camera.MODE_PERSPECTIVE);
50    } else {
51        frame.setProjectionMode(Camera.MODE_PLANAR_YZ);
52    }
53    frame.repaint();
54}
55
56void buildUserInterface() {
57    gui = new EjsControl(this); // use Easy Java Simulation
58        components to build a user interface
59    gui.addObject(
60        frame, "Frame",
61        "name=controlFrame;title=Bouncing
62            Ball;location=400,0;exit=true;size=pack");
63    gui.add(
64        "Panel",
65        "name=inputPanel; parent= controlFrame; layout=hbox;
66            position=south");
67    gui.add("Button",
68        "parent=inputPanel; text=Start;

```

```

65         action=startAnimation;" );
66     gui.add("Button",
67             "parent=inputPanel; text=Stop;
68                 action=stopAnimation;" );
69     gui.add("Button",
70         "parent=inputPanel; text=Step;
71             action=stepAnimation;" );
72     gui.add(
73         "CheckBox",
74         "parent=inputPanel;variable=3d;text=
75             3D;selected=true;action=set3d;" );
76   }
77 }
```

## 2.6 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch02` package.

### **Ball3DApp**

`Ball3DApp` creates a bouncing ball simulation by extending `AbstractSimulation` and implementing the `doStep` method. See Section 2.5.

### **PlotFrameApp**

`PlotFrameApp` demonstrates how to use a `PlotFrame`. See Section sec:tour/frames.

### **PlotterApp**

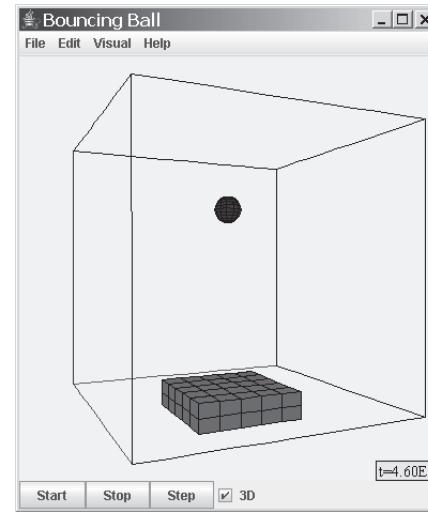
`PlotterApp` reads a string from a text field, converts this string to a function, and plots the function on the interval  $-10$  to  $10$ . See Section 2.3.

### **SHOApp**

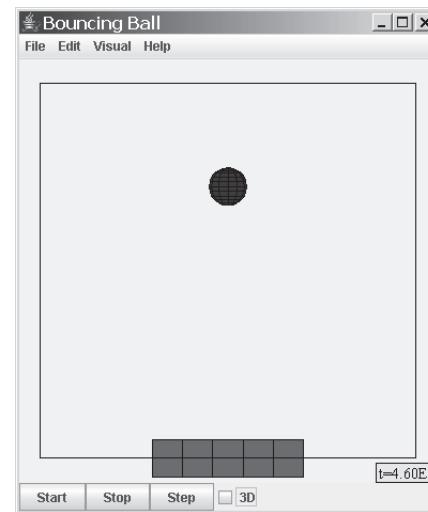
`SHOApp` creates a harmonic oscillator simulation with a 2D view by extending `AbstractAnimation` and implementing the `doStep` method. See Section 2.5.

## 2.6 Programs

35



(a) 3D Visualization.



(b) 2D Projection.

**FIGURE 2.4** Objects can be represented in three dimensions using the OSP 3D framework.

# CHAPTER

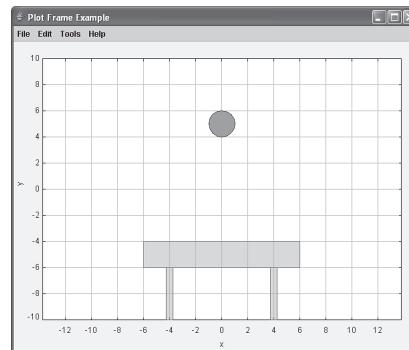
# 3

# Frames Package

©2005 by Wolfgang Christian

Because visualization is a very common requirement in the sciences, the Open Source Physics library defines frames that contain high-level data visualization and data analysis objects. Examples are located in the `org.opensourcephysics.manual.ch03` package unless otherwise stated.

## 3.1 ■ OVERVIEW



**FIGURE 3.1** A ball bouncing off of a table can be represented using a circle and three rectangles within a `DisplayFrame` as shown in Listing 3.1.

To enable users to quickly begin writing their own programs while they are learning the Java and the Open Source Physics APIs and in order to provide examples of how these APIs are used, we have defined a collection of high-level objects for routine visualization tasks. These objects are defined in the frames package. They extend the `DrawingFrame` superclass and add the functionality needed to plot datasets, display vector and scalar fields, and do numerical analysis such as Fourier transformations. Table 3.1 lists the classes in the package.

Frames in the `org.opensourcephysics.frames` package are high-level composite objects that perform methods by forwarding them to instances of other objects such as `Dataset` or `ContourPlot`. This chapter describes the classes in the frames package. Subsequent chapters in this Guide will show how these

## 3.1 Overview

37

**TABLE 3.1** The frames package contains plotting and data analysis objects.

<b>org.opensourcephysics.frames</b>	
ComplexPlotFrame	Displays $x$ - $y$ plots of complex datasets. Data points are added using the <code>append</code> method. Real and imaginary components can be drawn as separate curves or as a single function using color to show phase.
Complex2DFrame	Displays a complex scalar field. The complex field is defined using two-dimensional arrays for the real and imaginary components.
DisplayFrame	Draws objects such as images, circles, and rectangles.
Display3DFrame	Displays three-dimensional objects using the OSP 3D implementation in the <code>simple3d</code> package.
FFTFrame	Displays the fast Fourier transform of a complex dataset.
FFT2DFrame	Displays the fast Fourier transform of a two-dimensional scalar field.
HistogramFrame	Sorts the data into bins and plots the bins as bars.
LatticeFrame	Displays a two-dimensional array of integers.
PlotFrame	Displays $x$ - $y$ plots.
RasterFrame	Converts a scalar field into an image.
Scalar2DFrame	Displays a scalar field defined using a two-dimensional array of <code>double</code> .
TableFrame	Displays a table. Elements in any row must be of the same data type but elements in different rows can contain different data types. Rows are not required to be the same length.
Vector2DFrame	Displays a vector field. The scalar field is defined using a two-dimensional array for each vector component.

frames are composed of objects defined in other Open Source Physics packages. Users should consult the code listings and each frame's JavaDoc documentation for additional information as they gain familiarity with the Open Source Physics library.

The frames described in this chapter are designed to provide convenient general-purpose data visualization and analysis tools for *An Introduction to Computer Simulation Methods* by Gould, Tobochnik, and Christian and these classes may not meet the needs of other Open Source Physics projects. Ease of use and not computation speed or efficient use of memory motivated the frames package API. However, it is not difficult for readers to create their own optimized and cus-

tomized visualizations. Subsequent chapters describe how the core OSP classes are implemented so that computer resources can be used most efficiently.

### 3.2 ■ DISPLAY FRAME

The Open Source Physics library defines objects such as circles, rectangles, images, and single-line text that can be rendered on an output device such as a monitor or printer. These objects are referred to as being drawable because they implement the `Drawable` interface by defining a `draw` method. They are drawn in the order that they are added to an OSP drawing frame. Although drawable objects can be added to any frame in the frames package, this section uses the simplest of these frames, the `DisplayFrame` class, to demonstrate how drawable objects are displayed and manipulated.

The `DisplayFrame` class defines a general purpose frame and is shown in Figure 3.1. The frame's constructor has signatures to create a display area with and without axes. The axes labels and title are set when the frame is created, but these strings can later be changed using accessor methods such as `setXLabel`.

```

1 // constructor with axes
2 DisplayFrame frame =
3     new DisplayFrame("x axis label","y axis label", "Plot Frame
Title");
4 // constructor without axes
5 DisplayFrame frame = new DisplayFrame("Plot Frame Title");

```

The instantiated `DisplayFrame` is initially empty (blank) and is now ready to be customized by adding drawable objects. A typical use of this frame is to display a physical model. For example, a ball bouncing off of a table can be represented using a circle and three rectangles as shown in Listing 3.1. Note how the axes change as the frame is resized.

Listing 3.1 `BallAndTableApp` creates a visualization of a ball and a table.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.display.DrawableShape;
3 import org.opensourcephysics.frames.DisplayFrame;
4 import java.awt.Color;
5 import javax.swing.JFrame;
6
7 public class BallAndTableApp {
8     public static void main(String[] args) {
9         DisplayFrame frame = new DisplayFrame("x", "y",
"Plot Frame Example");
10        // DisplayFrame frame=new DisplayFrame("Plot Frame Example");
11        DrawableShape circle = DrawableShape.createCircle(0.0, 5.0,
12                2);
13        circle.setMarkerColor(new Color(128, 128, 255), Color.BLUE);

```

## 3.2 Display Frame

39

```

14    frame.addDrawable(circle);
15    DrawableShape rectangle = DrawableShape.createRectangle(-4,
16                  -8.0,
17                  0.5, 4);
18    frame.addDrawable(rectangle); // left leg
19    rectangle = DrawableShape.createRectangle(4, -8.0, 0.5, 4);
20    frame.addDrawable(rectangle); // right leg
21    rectangle = DrawableShape.createRectangle(0.0, -5.0, 12, 2);
22    frame.addDrawable(rectangle); // table top
23    frame.setVisible(true);
24    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25 }
}

```

Some, but not all, drawable objects can respond to mouse events and are called *interactive* objects. The `InteractiveShape` class in the display package defines interactive objects that can be moved by clicking and dragging. The `BoundedShape` class defines full-featured objects that can respond to additional mouse actions so that they can be rotated and resized. Run the program in Listing 3.2 to see the capabilities of these interactive objects. Note that you must double-click a `BoundedShape` to change its properties whereas you can simply click-drag an `InteractiveShape`. It is usually not a good idea to mix `BoundedShape` and `InteractiveShape` objects in a drawing panel, because users will not know if they should click-drag an object or double-click to select and then click-drag.

Listing 3.2 `InteractionApp` creates objects that can be manipulated using a mouse.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.frames.DisplayFrame;
4
5 public class InteractionApp {
6     public static void main(String[] args) {
7         DisplayFrame frame = new DisplayFrame("x", "y", "Drawable
8             Shapes");
9         frame.addDrawable(DrawableShape.createCircle(0.0, 0, 4));
10        frame.addDrawable(InteractiveShape.createCircle(-5.0, -5.0,
11                  4));
12        BoundedShape circle = BoundedShape.createBoundedCircle(-5.0,
13                  5.0,
14                  4);
15        circle.setWidthDrag(true);
16        circle.setHeightDrag(true);
17        frame.addDrawable(circle);
18        BoundedShape rectangle =
19            BoundedShape.createBoundedRectangle(0,
20                  -2, 4, 8);
21    }
22 }

```

**TABLE 3.2** The Open Source Physics library defines `Drawable` objects with varying capabilities.

<b>org.opensourcephysics.display.Drawable</b>	
AWT drawables	Objects, such as <code>Circle</code> , that use the original pixel-based drawing mechanism introduced in the Java 1.0 <i>Abstract Windows Toolkit</i> , AWT. Size properties are usually specified in pixel units.
Swing drawables	Objects, such as <code>DrawableShape</code> , that use the Java 2D API introduced in Java 1.2 to transform themselves from world to pixel coordinates. Properties are specified in world units.
Interactive drawables	Objects, such as <code>InteractiveShape</code> or <code>BoundedShape</code> , that support mouse actions to adjust their properties.

```

17    rectangle.setRotateDrag(true);
18    frame.addDrawable(rectangle);
19    BoundedShape arrow = BoundedShape.createBoundedArrow(-4,
20              -8.5, 8,
21              0);
22    arrow.setRotateDrag(true);
23    frame.addDrawable(arrow);
24    frame.setVisible(true);
25    frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
26 }
```

Right (control)-clicking within the display area shows a pop-up menu. The *Properties* menu item allows a user to examine the properties of objects in the frame. Other menu items enable a user to control the display area's scale.

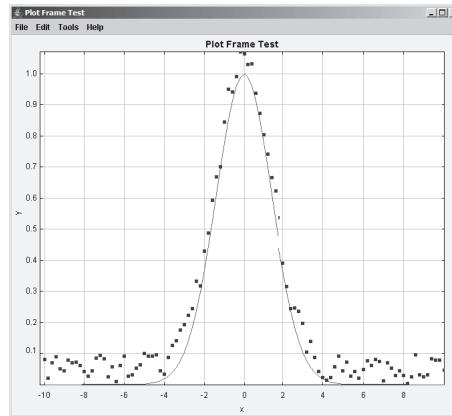
Table 3.2 gives an overview of some simple predefined geometric shapes that can be added to a `DrawingFrame`. These are just samples. Almost any geometric shape can be created using Java 2D as described in Chapter 4. Composite drawables, such as contour plot, vector field plot, and datasets are described in subsequent chapters. Programs should, of course, use the simplest drawable object that implements the required functionality.

### 3.3 ■ PLOT FRAME

The `PlotFrame` class is designed to create *x-y* plots by adding points to one or more *datasets* as shown in Figure 3.2. A *Dataset* is an object that contains an array of points and drawing attributes such as color and marker shapes. The details of dataset creation and management are hidden inside the frame. Data

## 3.3 Plot Frame

41

**FIGURE 3.2** A `PlotFrame` that displays a spectral line and data with random noise.

points are added directly to a `PlotFrame` using the `append` method. The frame creates `Dataset` objects as needed and appends points to the appropriate dataset.

```

1 PlotFrame frame = new PlotFrame("x", "y", "Plot Frame Test");
2 frame.append(0, 0.2, 3.1); // point (x,y)=(0.2, 3.1)
3 frame.append(1, new double[] {-2,-1,0}, new double[] {4,5,6});
// x-y arrays

```

The first parameter in the frame's `append` method is an index that identifies the dataset. The second and third parameters specify coordinates. In the above code fragment, dataset zero contains a single data point and dataset one contains three data points. Note that the `append` method has signatures that accept points, point arrays, and points with error estimates.

The `PlotFrame` class creates datasets with rectangular data point *markers* having different colors. A dataset's marker shape, size, and color, as well as a boolean that determines if markers should be connected with line segments, can be set using accessor methods. Listing 3.3 uses two datasets to simulate a Gaussian spectral line with random noise.

**Listing 3.3** `PlotFrameApp` displays two datasets.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.display.Dataset;
3 import org.opensourcephysics.frames.PlotFrame;
4
5 public class PlotFrameApp {
6     public static void main(String[] args) {
7         PlotFrame frame = new PlotFrame("$\\Delta$f", "intensity",
8                                         "Gaussian Lineshape");
9         frame.setConnected(0, true);
10        frame.setMarkerShape(0, Dataset.NO_MARKER);

```

```

11  for (double x = -10;x<10;x += 0.2) {
12    double y = Math.exp(-x*x/4);
13    frame.append(0, x, y); // datum
14    frame.append(1, x, y+0.1*Math.random()); // datum + noise
15  }
16  frame.setVisible(true);
17  frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
18  frame.setXPointsLinked(true); // default is true
19  frame.setXYColumnNames(0, "frequency", "theory");
20  frame.setXYColumnNames(1, "frequency", "experiment");
21  frame.setRowNumberVisible(true);
22}
23}

```

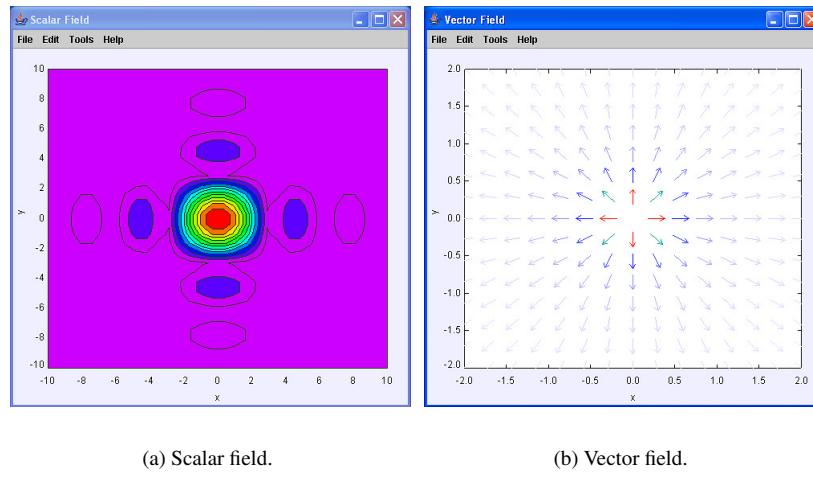
It is often convenient to examine data in a table and the `PlotFrame` class implements this feature using a *data table*. A data table can be created using the frame's file menu or using a right (control) mouse click to show a popup menu within the display area. Because it is common for points in multiple datasets to have the same independent variable, a `PlotFrame` assumes that *x* coordinates in multiple datasets are linked and suppresses all but the first column of *x* values in a data table. This behavior can be changed using the `setXPointsLinked` accessor method. In addition, the table's column names can be set as shown in Listing 3.3.

row	frequency	theory	experiment
31	-3.8	0.027	0.031
32	-3.6	0.039	0.115
33	-3.4	0.056	0.11
34	-3.2	0.077	0.151
35	-3	0.105	0.183
36	-2.8	0.141	0.188
37	-2.6	0.185	0.241
38	-2.4	0.237	0.313
39	-2.2	0.298	0.314
40	-2	0.368	0.385
41	-1.8	0.445	0.53
42	-1.6	0.527	0.599
43	-1.4	0.613	0.652
44	-1.2	0.698	0.744
45	-1	0.779	0.856
46	-0.8	0.852	0.948
47	-0.6	0.914	0.962
48	-0.4	0.961	1.033
49	-0.2	0.99	1.072

FIGURE 3.3 A `PlotFrame` can display its data in a table.

### 3.4 Scalar and Vector Fields

43



**FIGURE 3.4** Visualizations of two-dimensional fields.

## FIGURE

**FIGURE 3.4** Visualizations of two-dimensional fields.

### 3.4 ■ SCALAR AND VECTOR FIELDS

Imagine a plate that is heated at an interior point and cooled along its edges. In principle, the temperature of this plate can be measured at every point. A scalar quantity, such as temperature, pressure, or light intensity, that is defined throughout a region of space is known as a *scalar field*. Other physical quantities, such as the wind velocity or the force on a particle near Earth, require that a vector be measured at every point and therefore define a *vector field*. The Open Source Physics library contains tools that help us visualize two-dimensional scalar and vector fields. These low-level components are defined in the `display2d` package. The `frames` package defines high-level `Scalar2DFrame` and `Vector2DFrame` components that are not as flexible but are easier to use.

The `Scalar2DFrame` class allows us to view 2D scalar fields using representations such as contour plots and 3D surface plots. Listing 3.4 shows a `Scalar2DFrame` being used to visualize the diffraction pattern from a rectangular aperture. We plot the square root of the light intensity to favor the regions of low intensity.

**Listing 3.4** A scalar field test program.

```
1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.frames.Scalar2DFrame;
3 import javax.swing.JFrame;
4
5 public class Scalar2DFrameApp {
6     public static void main(String[] args) {
7         Scalar2DFrame frame = new Scalar2DFrame("x", "y", "Scalar
```

**TABLE 3.3** Scalar2DFrame visualizations are available from the frame's Views menu.

org.opensourcephysics.frames.Scalar2DFrame	
Contour Plot	Draws contour lines and colors the area between contours.
Grayscale Plot	Renders a 2D scalar field as a grayscale image.
Grid Plot	Renders a scalar field using multi-colored rectangles.
Interpolated Plot	Renders a scalar field by coloring pixels using values that are interpolated between grid points.
Surface Plot	Renders a scalar field by drawing a 3D surface whose height is proportional to the field's value.

```

8     Field");
9 // generate sample data
10 double[][] data = new double[32][32];
11 frame.setAll(data, -10, 10, -10, 10); // initialize field
12     and scale
13 for(int i = 0, nx = data.length; i<nx; i++) {
14     double x = frame.indexToX(i);
15     double ax = (x==0) ? 1 : Math.abs(Math.sin(x)/x);
16     for(int j = 0, ny = data[0].length; j<ny; j++) {
17         double y = frame.indexToY(j);
18         double ay = (x==0) ? 1 : Math.abs(Math.sin(y)/y);
19         data[i][j] = ax*ay; // square root of intensity
20     }
21 }
22 frame.setAll(data); // set new values
23 frame.setVisible(true);
24 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25 }

```

Run the scalar field example and note the various types of visualizations available under the frame's *Views* menu. Many of the visualizations shown in Table 3.3 produce useful representations even if the grid is small. Some visualizations may look better using a smaller grid size.

The frames package contains the `Vector2DFrame` class for displaying two-dimensional vector fields. To use this class we instantiate a multi-dimensional array to store vector components. The first array index is zero or one in order to specify the  $x$  or  $y$  vector component, respectively. The second array index iterates over the  $x$  coordinate and the third array index iterates over the  $y$  coordinate. As for scalar fields, the vectors are set by passing the data array to the frame using the `setAll` method. The program in Listing 3.5 demonstrates how a `Vector2DFrame` is used by displaying the electric field due to a unit charge located at the origin.

## 3.5 Complex Functions

45

Listing 3.5 A vector field test program.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.frames.Vector2DFrame;
3 import javax.swing.JFrame;
4
5 public class VectorFrameApp {
6     public static void main(String[] args) {
7         Vector2DFrame frame = new Vector2DFrame("x", "y", "Vector
8             Field");
9         double a = 2; // world units for vector field
10        int nx = 15, ny = 15;
11        // generate the data
12        double[][][] data = new double[2][nx][ny]; // vector field
13        frame.setAll(data, -a, a, -a, a); // initialize field and
14        scale
15        for(int i = 0;i<nx;i++) {
16            double x = frame.indexToX(i);
17            for(int j = 0;j<ny;j++) {
18                double y = frame.indexToY(j);
19                double r2 = x*x+y*y; // distance squared
20                double r3 = Math.sqrt(r2)*r2; // distance cubed
21                data[0][i][j] = (r2==0) ? 0 : x/r3; // x component
22                data[1][i][j] = (r2==0) ? 0 : y/r3; // y component
23            }
24        }
25        frame.setAll(data); // vector field displays new data
26        frame.setVisible(true);
27        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28    }
29}
```

The arrows representing the vectors have a fixed length that is chosen to match the grid spacing. The arrow's color represents the field's magnitude. The frame's legend item in the tools menu shows this mapping.

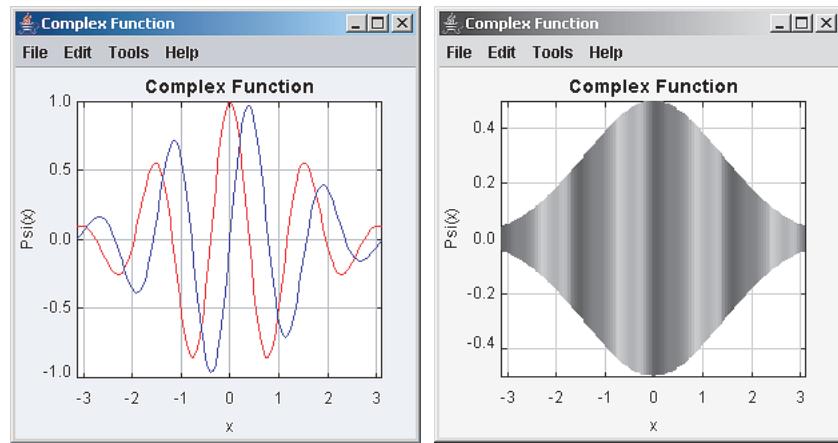
## 3.5 ■ COMPLEX FUNCTIONS

Complex functions are essential in many areas of physics such as quantum mechanics, and the frames package contains classes for displaying and analyzing these functions. Listing 3.6 uses a `ComplexPlotFrame` to display a one-dimensional complex wave function.

Listing 3.6 `ComplexPlotFrameApp` displays a one-dimensional quantum wave packet with a phase modulation.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.frames.ComplexPlotFrame;
```



(a) Real and imaginary.

(b) Amplitude and phase.

**FIGURE 3.5** Two representations of complex wave functions.

```

3 import javax.swing.JFrame;
4
5 public class ComplexPlotFrameApp {
6     public static void main(String[] args) {
7         ComplexPlotFrame frame = new ComplexPlotFrame("x", "Psi(x)",
8                                         "Complex Function");
9         int n = 128;
10        double
11            xmin = -Math.PI, xmax = Math.PI;
12        double
13            x = xmin, dx = (xmax-xmin)/n;
14        double[] xdata = new double[n];
15        double[] zdata = new double[2*n]; // real and imaginary
16        values alternate
17        int mode = 4;
18        for(int i = 0; i<n; i++) {
19            double a = Math.exp(-x*x/4); // wave function amplitude
20            // function is  $e^{-(x*x/4)} e^{(i*mode*x)}$  where  $x=[-pi, pi]$ 
21            zdata[2*i] = a*Math.cos(mode*x);
22            zdata[2*i+1] = a*Math.sin(mode*x);
23            xdata[i] = x;
24            x += dx;
25        }
26        frame.append(xdata, zdata);
27        frame.setVisible(true);
28        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

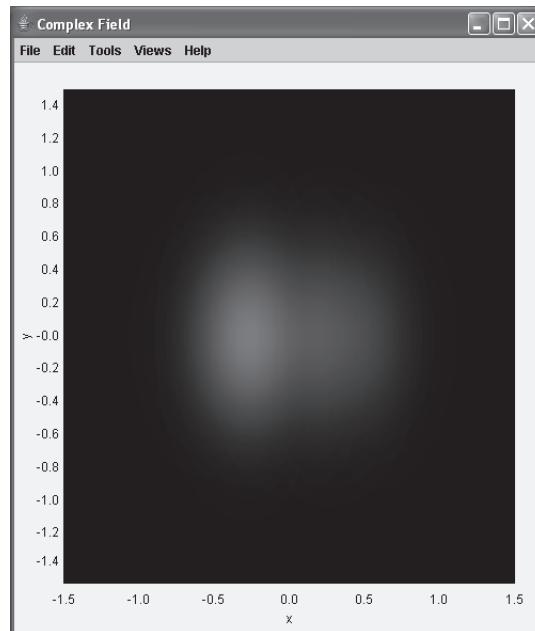
```

## 3.5 Complex Functions

47

28 }  
29 }

Figure 3.5 shows two representations of a one-dimensional complex (quantum) wave function. The real and imaginary representation displays the real and imaginary parts of the wave function  $\Psi(x)$  by drawing two curves. The amplitude and phase representation uses height to show wave function magnitude and color to show phase. Other wave function visualizations can be selected at runtime using the Views menu or programmatically using convert methods such as `convertToPostView` and `convertToReImView`. The Views menu also allows the user to display a data table and a legend to examine the wave function's values and to display the color to phase relationship, respectively.



**FIGURE 3.6** A Complex2DFrame that shows a two-dimensional Gaussian wave packet with a phase modulation.

The `Complex2DFrame` class is designed to display two-dimensional complex wave functions. This class is designed to plot a two-dimensional array containing the field's real and imaginary components. The `Complex2DFrameApp` example is available on the CD.

### 3.6 ■ FOURIER TRANSFORMATIONS

An arbitrary periodic function  $f(t)$  can be approximated as a series of sinusoidal functions. In other words, a function  $f(t)$  of period  $T$  can be approximated as a sum of  $N$  sine and cosine functions:

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^N (a_k \cos \omega_k t + b_k \sin \omega_k t), \quad (3.1)$$

where

$$\omega_k = k\omega_0 \text{ and } \omega_0 = 2\pi/T. \quad (3.2)$$

The quantity  $\omega_0$  is the fundamental frequency and the sum is called a Fourier series. Because it is mathematically convenient to work with complex numbers, the sine and cosine functions of the same frequency are often combined into a single complex exponential using Euler's formula<sup>1</sup>:

$$e^{i\omega_k t} = \cos \omega_k t + i \sin \omega_k t. \quad (3.3)$$

By using (3.3) we can express an arbitrary periodic complex function  $f(t)$  as

$$f(t) = \sum_{k=-N/2}^{N/2} c_k e^{i\omega_k t}, \quad (3.4)$$

where the expansion coefficients  $c_k$  are complex numbers.

The process of approximating a function by computing the expansion coefficients  $c_k$  is called *Fourier analysis* and the most efficient algorithm for performing this computation is known as the fast Fourier transformation (fft). Computing Fourier coefficients using one of the many publicly available fft implementations is straightforward but may require a fair amount of bookkeeping. To simplify the process, we have defined the `FFTFrame` class in the frames package to perform a one-dimensional complex fft and display the coefficients. This utility class accepts either data arrays or functions as input parameters to the `doFFT` method. The code shown in Listing 3.7 transforms an input array containing the complex exponential  $e^{imt}$ . The mode number  $m$  determines the fourier coefficient.

Listing 3.7 `FFTFrameApp` displays the coefficients of the function  $e^{2\pi mt}$  where  $m$  is the harmonic (mode).

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.frames.FFTFrame;
3 import javax.swing.JFrame;
4

```

<sup>1</sup>Euler's formula is a special case of De Moivre's formula  $(\cos x + i \sin x)^n = \cos nx + i \sin nx$  that preceded it.

## 3.6 Fourier Transformations

49

```

5  public class FFTFrameApp {
6    public static void main(String[] args) {
7      FFTFrame frame = new FFTFrame("omega", "amplitude",
8                                    "FFT Frame Test");
9      int n = 16; // number of data points
10     double
11       tmin = 0, tmax = 2*Math.PI;
12     double
13       t = tmin, delta = (tmax-tmin)/n;
14     double[] data = new double[2*n];
15     frame.setDomainType(FFTFrame.OMEGA);
16     int mode = 2; // function is  $e^{(i*mode*x)}$  where  $x=[0,2\pi]$ .
17     for(int i = 0; i < n; i++) {
18       data[2*i] = Math.cos(mode*t);
19       data[2*i+1] = Math.sin(mode*t);
20       t += delta;
21     }
22     frame.doFFT(data, tmin, tmax);
23     frame.setVisible(true);
24     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25   }
26 }
```

The extension of the ideas of Fourier analysis to two dimensions is simple and direct. If we assume a complex function of two variables  $f(x, y)$ , then a two-dimensional series is constructed using harmonics of both variables. The expansion functions are the products of one-dimensional functions  $e^{ixq_x}e^{iyq_y}$  and the Fourier series is written as a sum of these harmonics:

$$f(x, y) = \sum_{n=-N/2}^{N/2} \sum_{m=-M/2}^{M/2} c_{n,m} e^{iq_n x} e^{iq_m y}, \quad (3.5)$$

where

$$q_n = \frac{2\pi n}{X} \quad \text{and} \quad q_m = 2\pi \frac{m}{Y}. \quad (3.6)$$

The function  $f(x, y)$  is assumed to be periodic in both  $x$  and  $y$  with periods  $X$  and  $Y$ , respectively.

Because of the large number of coefficients  $c_{n,m}$ , the discrete two-dimensional Fourier transform is also implemented using the fft algorithm. `FFT2DFrameApp` shows how to compute and display a 2D fft using the `FFT2DFrame` class.

The Fourier analysis implementation in the OSP library is based on FFT routines contributed to the GNU Scientific Library (GSL) by Brian Gough and adapted to Java by Bruce Miller at NIST. We initialize our data array to conform to the GSL API using a one-dimensional array such that rows follow sequentially. This ordering is known as row-major format. Because the input function is assumed to be complex, the array has dimension  $2N_x N_y$ , where  $N_x$  and  $N_y$

are the number of grid points in the  $x$  and  $y$  direction, respectively. One disadvantage of using row-major format is that the number of rows must be specified as an additional parameter in order to interpret the data. The `FFT2DFrame` object transforms and displays the data when the `doFFT` method is invoked. Note that the `doFFT` methods is invoked with a position space scale and that this scale is used to compute the spatial frequencies of the Fourier components.

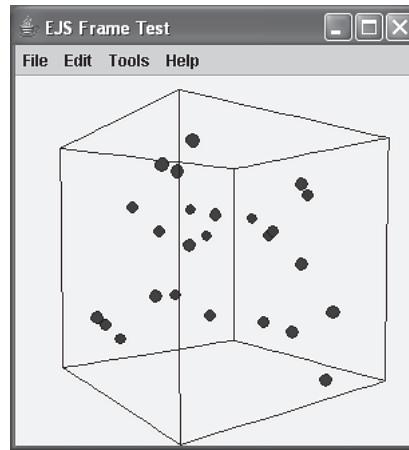
Listing 3.8 `FFT2DFrameApp` displays the coefficients for the two-dimensional Fourier analysis of the complex function  $e^{i2\pi nx} e^{i2\pi my}$ .

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.frames.FFT2DFrame;
3 import javax.swing.JFrame;
4
5 public class FFT2DFrameApp {
6     public static void main(String[] args) {
7         FFT2DFrame frame = new FFT2DFrame("k_x", "k_y", "2D FFT");
8         double
9             xmin = 0, xmax = 2*Math.PI, ymin = 0, ymax = 2*Math.PI;
10        // generate data on a grid of size (nx, ny)
11        int nx = 5, ny = 5;
12        int xMode = -1, yMode = -1;
13        double[] zdata = new double[2*nx*ny]; // field stored in
14        // row-major format with array size 2*nx*ny
15        // test function is  $e^{(i*xmode*x)}e^{(i*ymode*y)}$  where where x
16        // and y limits are [0,2pi].
17        double
18            y = 0, yDelta = 2*Math.PI/ny;
19        for(int iy = 0; iy<ny; iy++) {
20            int offset = 2*iy
21                *nx; // offset to beginning of a row; each
22                // row is nx long
23            double
24                x = 0, xDelta = 2*Math.PI/nx;
25            for(int ix = 0; ix<nx; ix++) {
26                zdata[offset+2*ix] = // real part
27                    Math.cos(xMode*x)*Math.cos(yMode*y)
28                    -Math.sin(xMode*x)*Math.sin(yMode*y);
29                zdata[offset+2*ix+1] = // imaginary part
30                    Math.sin(xMode*x)*Math.cos(yMode*y)
31                    +Math.cos(xMode*x)*Math.sin(yMode*y);
32                x += xDelta;
33            }
34            y += yDelta;
35        }
36        frame.doFFT(zdata, nx, xmin, xmax, ymin, ymax);
37        frame.setVisible(true);
38        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39    }

```

37 { }

**3.7 ■ DISPLAY 3D FRAME**

**FIGURE 3.7** A `Display3DFrame` displaying particles at random locations.

There are a number of APIs available for three-dimensional visualizations using Java. Although Sun has developed *Java 3D*, this library is currently not included in the standard Java runtime environment. The *Java bindings for OpenGL* (JOGL) library is a popular alternative because it is based on the OpenGL language. Because the basic OSP library should not rely on other 3D libraries to be downloaded and installed on a client computer and because we want a three-dimensional visualization framework designed for physics simulations, we have developed our own framework that relies only on the standard Java API (see Figure 3.7). This 3D framework uses packages (subdirectories) in the OSP `display3d` directory.

The OSP 3D API is based on Java interfaces defined in the `org.opensourcephysics.display3d.core` package. Using interfaces allows us to develop multiple implementations using various graphics packages. The OSP 3D API is similar to the OSP 2D API except that 3D objects are referred to as `Elements` rather than `Drawables`. Three-dimensional drawable objects are created and added to a 3D container using the `addElement` method as shown in Listing 3.9.

The `Display3DFrame` class in the `frames` package uses concrete implementations of 3D `Elements` defined in the `org.opensourcephysics.display3d.simple3d` package. This package uses only the standard Java library and is well

## Chapter 3 Frames Package

suited for displaying small objects that do not intersect. It avoids advanced rendering techniques by decomposing large objects into smaller polygons. You may notice artifacts at the polygon intersections but large objects can be decomposed into smaller pieces and the results are almost always satisfactory for simple physical models such as collections of particles and vectors.

Listing 3.9 `Display3DFrameApp` displays twenty five interactive particles using the OSP 3D framework.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.display3d.simple3d.*;
3 import org.opensourcephysics.frames.Display3DFrame;
4 import javax.swing.JFrame;
5
6 public class Display3DFrameApp {
7     public static void main(String[] args) {
8         Display3DFrame frame = new Display3DFrame("Random Circles");
9         for(int i = 0; i < 25; i++) {
10             Element ball = new ElementCircle();
11             ball.setSizeXYZ(0.1, 0.1, 0.1);
12             ball.setXYZ(-1.0+2*Math.random(), -1.0+2*Math.random(),
13                         -1.0+2*Math.random());
14             frame.addElement(ball);
15         }
16         frame.setVisible(true);
17         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     }
19 }
```

We will not study the `simple3d` implementation in detail here, but will describe only key OSP 3D concepts (see Chapter 11). `Display3DFrameApp` creates a simple three-dimensional visualization. Run the program and drag the mouse within the panel. Line and perspective details are hidden from the user as the viewing (camera) position is changed.

Figure 3.8 shows another example, a 3D visualization of an electromagnetic wave. The wave is constructed using arrows as shown in Listing 3.10.

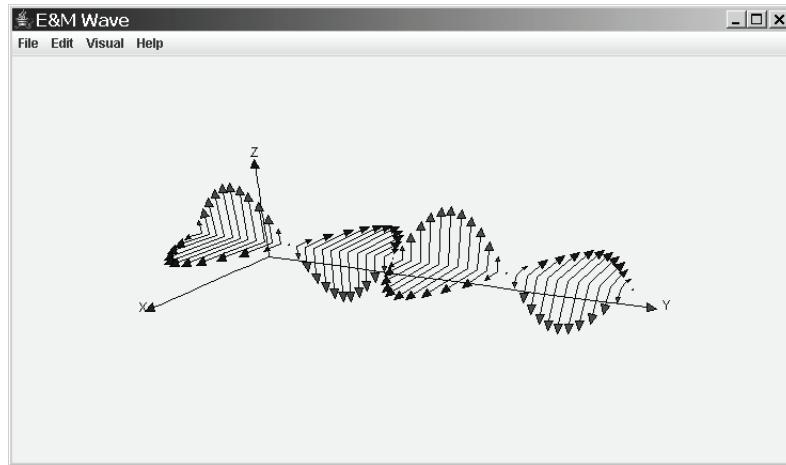
Listing 3.10 A 3D visualization of an electromagnetic wave is constructed using arrows.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.display3d.simple3d.*;
3 import org.opensourcephysics.frames.Display3DFrame;
4 import javax.swing.JFrame;
5
6 public class Wave3DApp {
7     public static void main(String[] args) {
8         // Data for the wave
9         int n = 48; // number of arrows
```

## 3.7 Display 3D Frame

53

**FIGURE 3.8** A 3D visualization of an electromagnetic wave.

```

10   double
11     time = 0.0, dt = 0.1, E0 = 100.0, Vy = 40.0, B0 = E0;
12   double
13     period = 5.0, omega = 2.0*Math.PI/period, k = omega/Vy;
14   double[] y = new double[n];
15   Display3DFrame frame = new Display3DFrame("E&M Wave");
16   frame.setPreferredMinMax(-100, 100, 0, 400, -100, 100);
17   frame.setDecorationType(VisualizationHints.DECORATION_AXES);
18   ElementArrow[] fieldE = new ElementArrow[n];
19   ElementArrow[] fieldB = new ElementArrow[n];
20   for(int i = 0;i<n;i++) {
21     y[i] = i*400.0/n;
22     fieldE[i] = new ElementArrow();
23     fieldE[i].getStyle().setFillColor(java.awt.Color.RED);
24     fieldE[i].setXYZ(0, y[i], 0);
25     fieldE[i].setSizeXYZ(0, 0, 0);
26     frame.addElement(fieldE[i]);
27     fieldB[i] = new ElementArrow();
28     fieldB[i].getStyle().setFillColor(java.awt.Color.BLUE);
29     fieldB[i].setXYZ(0, y[i], 0);
30     fieldB[i].setSizeXYZ(0, 0, 0);
31     frame.addElement(fieldB[i]);
32   }
33   frame.setSquareAspect(false);
34   frame.setVisible(true);
35   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36   while(true) { // animate until the program exits
37     try {
38       Thread.sleep(100);

```

```

39 } catch(InterruptedException ex) {}
40 time += dt;
41 for(int i = 0;i<n;i++) {
42     fieldE[i].setSizeZ(E0*Math.sin(k*(y[i]-Vy*time)));
43     fieldB[i].setSizeX(B0*Math.sin(k*(y[i]-Vy*time)));
44 }
45 frame.render();
46 }
47 }
48 }
```

Additional OSP 3D implementations that use advanced rendering packages such as Java 3D and JOGL are under development. All that will be required for a program to use these more sophisticated libraries is to install the library and import the corresponding OSP 3D package.

### 3.8 ■ TABLES

The `TableFrame` class is designed to show a table without the overhead of creating a plot. Data are appended one row at a time using the `appendRow` method.

```

1 TableFrame tableFrame = new TableFrame("Root Table");
2 tableFrame.appendRow(new String[]{"one","two"});
3 tableFrame.appendRow(new int[]{2,3,4,5});
```

Each row is a one-dimensional array of double, integer, string, or byte data, but these arrays need not be the same length. Listing 3.11 uses a `TableFrame` to create a table of square and cube roots. Users can copy data from this table into the system clipboard by highlighting a block of cells and then using the keyboard copy command.

Listing 3.11 `TableFrameApp` demonstrates how to create a table of numeric values.

```

1 package org.opensourcephysics.manual.ch03;
2 import org.opensourcephysics.frames.TableFrame;
3
4 public class TableFrameApp {
5     public static void main(String[] args) {
6         TableFrame tableFrame = new TableFrame("Root Table");
7         tableFrame.setRowNumberVisible(false);
8         tableFrame.setColumnNames(0, "x");
9         tableFrame.setColumnNames(1, "square root");
10        tableFrame.setColumnNames(2, "cube root");
11        for(int i = 0;i<10;i++) {
12            try {
13                tableFrame.appendRow(new double[] {i, Math.sqrt(i),
14                                Math.pow(i, 1.0/3)});
```

```

15     } catch (Exception ex) {}
16 }
17 tableFrame.setVisible(true);
18 tableFrame.setDefaultCloseOperation(
19     javax.swing.JFrame.EXIT_ON_CLOSE);
20 }
21 }
```

### 3.9 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch03` package.

#### **BallAndTableApp**

`BallAndTableApp` creates a visualization of a ball and a table using a `DisplayFrame`. See Section 3.2.

#### **Complex2DFrameApp**

`Complex2DFrameApp` demonstrates how to plot a 2D complex scalar field by displaying a two-dimensional Gaussian wave function with a momentum boost. See Section 3.5.

#### **ComplexPlotFrameApp**

`ComplexPlotFrameApp` demonstrates how to plot a complex function by displaying a one-dimensional Gaussian wave function with a momentum boost. See Section 3.5.

#### **Display3DFrameApp**

`Display3DFrameApp` demonstrates how to create a simple 3D view by displaying 25 particles in a 3D box. See Section 3.7.

#### **FFT2DFrameApp**

`FFT2DFrameApp` tests the `FFT2DFrame` class by taking the two-dimensional Fast Fourier Transform (FFT) of a harmonic function. See Section 3.6.

#### **FFTFrameApp**

`FFTFrameApp` tests the `FFTFrame` class by taking the one-dimensional Fast Fourier Transform (FFT) of a harmonic function. See Section 3.6.

#### **InteractionApp**

`InteractionApp` creates three bounded shapes and places them in a `DisplayFrame`. Double click on a shape to select it and then drag the hot spots. See Sec-

tion 3.2.

### **PlotFrameApp**

PlotFrameApp simulates a best fit to a Gaussian spectral line using a PlotFrame with two datasets. These datasets uses different point markers and drawing styles. See Section 3.3.

### **Scalar2DFrameApp**

Scalar2DFrameApp tests the Scalar2DFrame by plotting a scalar field  $f(x, y) = xy$ . See Section 3.4.

### **TableFrameApp**

TableFrameApp demonstrates how to create and use a TableFrame. See Section 3.8.

### **VectorFrameApp**

VectorFrameApp plots a  $1/r^2$  vector field using a Vector2DFrame. See Section 3.4.

### **Wave3DApp**

Wave3DApp displays a visualization of a transverse traveling wave. See Section sec:frame/display3dframe

## CHAPTER

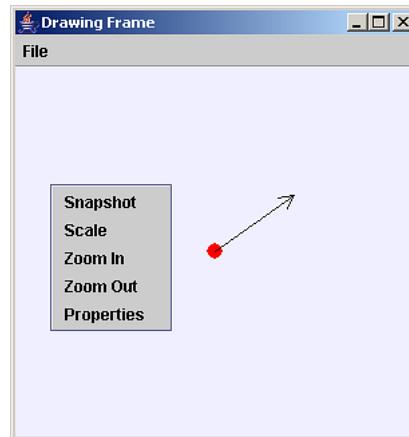
## 4

## Drawing

©2005 by Wolfgang Christian

The Open Source Physics 2D drawing framework is defined in the `org.opensourcephysics.display` package. Examples are located in the `org.opensourcephysics.manual.ch04` package unless otherwise stated.

#### 4.1 ■ OVERVIEW



**FIGURE 4.1** A drawing panel within a drawing frame. The drawing panel in the figure contains a drawable circle and a drawable arrow. The panel's popup menu is activated by right (control)-clicking within the panel.

One of the attractions of Java is that device and platform independent graphics is incorporated directly into the language. Lines, ovals, rectangles, images, and text can be drawn with just a few statements. The position of each shape is specified using an integer coordinate system that has its origin located at the upper left hand corner of the device — the computer display or printer paper — and whose positive  $y$  direction is defined to be down. Although we can use Java's graphics capabilities to produce visualizations and animations, creating even a simple graph in such a coordinate system can require a fair amount of programming. A scale

must be established, axes need to be drawn, and data needs to be transformed. An additional complication arises due to the fact that a graph must be able to redraw itself whenever an application is exposed or a window resized. But data visualization is a fairly common operation. We have developed a drawing framework that not only scales data, but can also be used for animations and other visualizations. Listing 4.1 uses this framework to create a rectangle and a circle in a plot frame.

Listing 4.1 A drawing with a drawable shape.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.BoundedShape;
3 import org.opensourcephysics.frames.PlotFrame;
4 import javax.swing.JFrame;
5
6 public class DrawingApp {
7     public static void main(String[] args) {
8         PlotFrame frame = new PlotFrame("x", "y", "Drawing Demo");
9         frame.setPreferredMinMax(-10, 10, -10, 10);
10        BoundedShape ishape = BoundedShape.createBoundedCircle(3, 4,
11            5);
12        ishape.setHeightDrag(true);
13        ishape.setXYDrag(false);
14        frame.addDrawable(ishape);
15        ishape = BoundedShape.createBoundedRectangle(0, 0, 9, 3);
16        frame.addDrawable(ishape);
17        frame.setVisible(true);
18        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19    }
}

```

Run DrawingApp and do the following:

1. Click-drag within the drawing and notice the coordinate display in the lower left hand corner.
2. Double-click on a shape to select it and drag the hot-spots to change the shape's properties.
3. Right-click within the drawing to display a pop-up menu and explore the menu items.
4. Resize the frame and observe changes in the axes.
5. Explore the menu items in the frame's menu bar.

The drawing framework is defined in the `org.opensourcephysics.display` package. As the above example shows, the display package defines a drawing API with a high level of abstraction. This chapter examines the details of how the OSP 2D drawing API is constructed.

## 4.2 ■ DRAWABLES

The drawing framework is based on the *Drawable* interface and the *DrawingPanel* class that are defined in the `org.opensourcephysics.display` package. The *Drawable* interface contains a single method, `draw`, that is invoked automatically from within the drawing panel's `paintComponent` method.

```
1 public interface Drawable {
2     public void draw(DrawingPanel drawingPanel, Graphics g);
3 }
```

Drawable objects, that is, objects that implement this interface, are instantiated and then added to a drawing panel where they will draw themselves in the order that they are added. Listing 4.2 shows a program that creates a drawing panel containing a circle and an arrow. Other drawable objects are listed in Table 4.1.

Listing 4.2 Drawable objects in a DrawingPanel.

```
1 import org.opensourcephysics.display.*;
2 import javax.swing.JFrame;
3
4 public class CircleAndArrowApp {
5     public static void main(String[] args) {
6         // create a drawing frame and a drawing panel
7         DrawingPanel panel = new DrawingPanel();
8         panel.setPreferredMinMax(-10,10,-10,10);
9         DrawingFrame frame = new DrawingFrame(panel);
10        panel.setSquareAspect(false);
11        Drawable circle = new Circle(0, 0);    // create circle
12        panel.addDrawable(circle);           // add to panel
13        Drawable arrow = new Arrow(0, 0, 4, 3); // create arrow
14        panel.addDrawable(arrow);           // add to panel
15        frame.show();                      // show frame
16        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17    }
18}
```

Drawables are easy to create using either Java *awt* or Java 2D drawing techniques.<sup>1</sup> For example, the drawable circle defined in the display package has a center at  $(x, y)$  in world coordinates and a radius in pixels (see Listing 4.3). It implements *Drawable* using a standard Java graphics context *Graphics*.

Listing 4.3 A drawable circle with fixed pixel radius.

```
1 import java.awt.*;
2 public class Circle implements Drawable {
3     int pixRadius = 6;
```

<sup>1</sup>The Java Abstract Windows Toolkit, *awt*, was introduced in Java 1.0, and implements primitive pixel based drawing methods.

```

4   Color color = Color.red;
5   double x = 0;
6   double y = 0;
7
8   public Circle(double _x, double _y) {
9       x = _x;
10      y = _y;
11  }
12
13  public void draw(DrawingPanel panel, Graphics g) {
14      int xpix = panel.xToPix(x) - pixRadius;
15      int ypix = panel.yToPix(y) - pixRadius;
16      g.setColor(color);
17      g.fillOval(xpix, ypix, 2*pixRadius, 2*pixRadius); // draws a circle
18  }
19

```

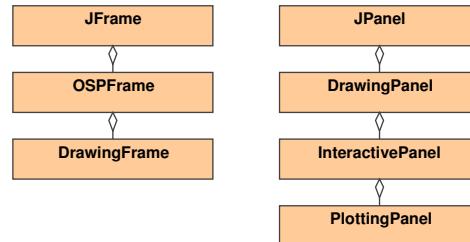
Better drawing paradigms than those in the Java 1.0 `awt` API are now common. This API only supports solid single pixel lines, limited fonts, and no rotation or scaling. The Java2D API introduced in Java 1.2 is not restricted to pixel coordinates nor is it restricted to solid single-pixel lines. Much of its flexibility is proved by affine transformations as described in Section 4.9. *Java Advanced Imaging* (JAI) is available from Sun as an add-on package that provides extended imaging processing capabilities beyond those found in Java 2D.

**TABLE 4.1** Examples of drawable objects defined in the display package.

<b>org.opensourcephysics.display</b>	
<code>Arrow</code>	An arrow with the location of its tail and <i>x-y</i> components defined in world units.
<code>Circle</code>	A circle with its center defined in world units and its radius defined in pixels.
<code>Dataset</code>	Data points with various rendering options such as color and marker styles.
<code>FunctionDrawer</code>	Draws a graph by evaluating a function at every pixel on the drawing panel's <i>x</i> -ordinate.
<code>Trail</code>	An array of connected data points.

The Open Source Physics library contains numerous other components for data visualization including a contour component, a wire mesh component, and a vector field component as described in Chapter 8. These components all implement the drawable interface so that it is possible, for example, to show the motion of a circle superimposed on potential energy contours by adding a contour plot and a circle to a drawing panel.

### 4.3 ■ DRAWING PANEL AND FRAME



**FIGURE 4.2** Drawing Frames and Drawing Panels inherit from Swing components.

The `DrawingPanel` class is a subclass of `JPanel`. Its primary purpose is to define a system of *world coordinates* that allow us to specify the location and size of objects in units other than pixel units. The `DrawingFrame` class is a subclass of `OSPFframe` which is a subclass of `JFrame`. A `DrawingFrame` is designed to display a single drawing panel in the center of its content pane. This frame also has a default menu including a print option.

The `InteractivePanel` class is a subclass of `DrawingPanel` that gathers mouse and keyboard actions and passes them to a listener. The `PlottingPanel` class is a subclass of `InteractivePanel` that adds axes. Because plotting data is an important application of drawing, the full capabilities of this class are presented in Chapter 6. We will use the `PlottingPanel` class in various examples in this chapter when it is convenient to display a scale.

Drawable objects do not “belong” to any panel and can be rendered in more than one panel. These objects can represent themselves in different panels because the `draw` method obtains a reference to the drawing panel and to the graphics context. Drawable objects can, for example, simultaneously render themselves in a zoom and an extended view.

Swing components paint themselves using the `paintComponent` method and the drawing panel’s `paintComponent` method begins by reading the panel’s pixel width and height. The panel then calculates the optimum scale using preferred values of  $x_{\min}$ ,  $x_{\max}$ ,  $y_{\min}$ , and  $y_{\max}$ . Finally, it calculates the world-to-pixel coordinate transformation. If an aspect ratio of unity is selected, the pixels per unit are equal along both axes, and the minimum and maximum values for each axis are calculated to be at least as large as the preferred values. Examples of common drawing panel methods are shown in Table 4.2.

When a drawing panel is repainted, it iterates through all drawable objects invoking each object’s `draw` method. Each drawable object has access to the panel’s coordinate transformation because the `draw` method is passed a reference to the drawing panel. It is, of course, possible to combine drawable objects to build compound objects such as graphs (see Chapter 6). The display package defines all the

**TABLE 4.2** Common drawing panel methods.

<b>org.opensourcephysics.display.DrawingPanel</b>	
getAspectRatio	Gets the ratio of $x$ pixels per unit to $y$ pixels per unit.
getPixelTransform	Gets the Java 2D affine transformation that converts from world coordinates to pixel coordinates.
pixToX	Converts horizontal pixel values to world coordinates.
pixToY	Converts vertical pixel values to world coordinates.
setPreferredMinMax	Sets the preferred coordinate scale.
xToPix	Converts $x$ world coordinates to horizontal pixel values.
yToPix	Converts $y$ world coordinates to vertical pixel values.

components necessary to produce a plot including axes and a `Dataset` class that stores and renders  $x$ - $y$  data points.

#### 4.4 ■ INSPECTORS

It is often convenient to change a drawing panel's properties while a program is running by right-clicking to activate the panel's popup menu. We can, for example, *zoom-in* and *zoom-out* for axes that are *not* autoscaled. If an axis is autoscaled, then that axis scale is set by the objects within the panel. The pop-up menu also allows a user to display the drawing panel's *inspector*.

Drawing panel properties can be examined using an inspector. The default drawing panel inspector is shown in Figure 4.3. If the inspector option has been enabled, it can be displayed by right (control)-clicking within the panel.

```

1 PlottingPanel plottingPanel = new PlottingPanel ("x", "y",
    "Title");
2 plottingPanel.enableInspector(true);
3 plottingPanel.showInspector();
```

The default inspector uses an *extensible markup language* (xml) based syntax and this syntax may not be appropriate for nontechnical users (see Chapter 12). It is, however, very useful for developers because it allows them to inspect and sometimes edit an object's properties while a program is running. It is straightforward to create a user friendly inspector for distribution to less technical audiences. We replace the default inspector with a custom Window component using the `setCustomInspector` method.

## 4.5 Message Boxes

63

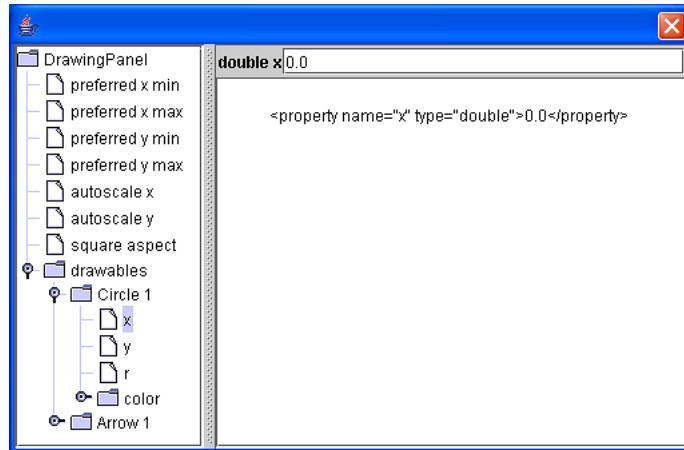


FIGURE 4.3 The default drawing panel inspector.

```

1 PlottingPanel plottingPanel = new PlottingPanel ("x", "y",
    "Title");
2 plottingPanel.setCustomInspector(window); // window is an
    instance of JWindow

```

## 4.5 ■ MESSAGE BOXES

Drawing panels borrow the `GlassPane` concept from the Java `JRootPane` class. The drawing panel's glass pane is a transparent component that sits on top of the panel. It is designed to display Swing components without interfering with the drawing panel. We use the glass pane to layout yellow message boxes that display coordinate values when the mouse is dragged and that display custom messages.

The default glass pane contains four message boxes located in the drawing panel's four corners. Messages can be placed into these boxes using the `setMessage` method.

```

1 double t = 0, en = 10;
2 PlottingPanel plottingPanel = new PlottingPanel("x","y","title");
3 plottingPanel.setMessage("time = "+t); // lower right is default
4 plottingPanel.setMessage("energy = "+en, DrawingPanel.TOP_LEFT);

```

Message boxes are located at the four corners and the default message box is located in the lower right corner.

Users can add their own custom components to the glass pane by obtaining a reference from the drawing panel. Listing 4.4 shows how this is done. Note that the glass pane uses a custom `OSPLayout` manager that supports corner and centered placement of components.

Listing 4.4 Messages are drawn using a glass pane that sits on top of the drawing panel.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.*;
3
4 public class MessageApp {
5     public static void main(String[] args) {
6         // create a drawing frame and a drawing panel
7         DrawingPanel panel = new DrawingPanel();
8         DrawingFrame frame = new DrawingFrame(panel);
9         panel.setMessage("time="+0);
10        panel.setMessage("energy="+0, 2); // top right
11        MessagePanel messagePanel = new MessagePanel();
12        panel.getGlassPanel().add(messagePanel, OSPLayout.CENTERED);
13        frame.setVisible(true);
14        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
15    }
16 }
17
18 class MessagePanel extends javax.swing.JPanel {
19     MessagePanel() {
20         setPreferredSize(new java.awt.Dimension(100, 100));
21         this.setBackground(java.awt.Color.RED);
22     }
23 }
```

## 4.6 ■ SCALE

Computers typically use an integer coordinate system based on pixels to access locations on a device. The origin  $(0, 0)$  is located in the upper left hand corner, the horizontal  $x$  coordinate increases toward the right, and the vertical  $y$  coordinate increases toward the bottom. We refer to these coordinates as *pixel coordinates*. Physical models use *world coordinates* in whatever units are appropriate. For example, the distance from Earth to Sun is defined to be one astronomical unit (AU) and this scale is often used when modeling the solar system. In order to make it easy for programs to create visualizations, the position of a physical object is almost always computed in world coordinates. In a computer simulation, a model is solved in whatever units are appropriate and the location of drawable objects representing the physics are then set using world coordinates.

A `DrawingPanel` contains a transformation that is used to scale and locate drawable objects before drawing their visual representation on the output device. The default scale along each axis is -10 to 10 but these limits can be changed a number of ways. The most direct approach is to specify the minimum and maximum values along the  $x$  (horizontal) and  $y$  (vertical) axes. The following code fragment sets the  $x$  axis to  $[0, 20]$  and the  $y$  axis to  $[-10, 10]$ .

## 4.6 Scale

## 65

```

1 // setPreferredMinMax signature is (xmin, xmax, ymin, ymax)
2 drawingPanel.setPreferredMinMax(0, 20, -10, 10);
3 drawingPanel.setSquareAspect(true); // default is true

```

These minimum and maximum values specify a preferred range. The true range can differ because the panel may not be square and the computed pixels per unit in the *x* direction may differ from the pixels per unit in the *y* direction. The algorithm adjusts the scale's minimum and maximum values to insure that pixels per unit are the same in both directions. This algorithm also guarantees that the true range contains the preferred range. Setting the square aspect property to false allows the panel to honor the preferred minimum and maximum values. A non-square aspect will, of course, distort the appearance of geometric shapes.

The number of screen pixels per world unit can also be set explicitly using the `setPixelsPerUnit` method. The following code fragment sets the panel's scale to be exactly 10 pixels per unit in both the *x* and *y* directions.

```

1 drawingPanel.setPixelsPerUnit(true, 10, 10);

```

The center of the display area is set to the average of the preferred minimum and maximum values and true minimum, and maximum values are then computed using the window size and the given pixels per unit values. In other words, setting the pixels per unit causes the size of the visible "world" to decrease and increase in direct proportion to the window size.

A third scaling option computes a drawing panel's maximum and minimum values using data supplied by the drawable objects themselves. For example, `Dataset` objects keep track of their minimum and maximum *x* and *y* values as data are appended and these values can be used to insure that all points are displayed. Drawable objects can provide these minimum and maximum values to a panel by implementing the `Measurable` interface as described in the next section. If autoscaling is enabled and if measurable objects are added to the panel, then the drawing panel computes a scale using data supplied by the drawable objects themselves.

```

1 drawingPanel.setAutoscaleX(true);
2 drawingPanel.setAutoscaleY(true);

```

Although it is sometimes convenient to allow objects to adjust a panel's scale, this method can lead to inappropriate values if the *x-y* range is small. We might, for example, wish to insure that the minimum *y* value be no greater than zero and the maximum *y* value be no smaller than ten even if every point has a *y* value of 5. This behavior can be obtained by setting floor and ceiling limits.

```

1 drawingPanel.setAutoscaleY(true);
2 drawingPanel.limitAutoscaleY(0, 10); // sets floor and ceiling

```

Limits allow the panel to autoscale the axis if the measurable object's minimum and maximum values exceed the given interval [0, 10], but insure that the axis always contains the interval.

## 4.7 ■ MEASURABLE

If the drawing panel's `autoscaleX` or `autoscaleY` properties are set to true, the panel reads the minimum and maximum values of its measurable objects at the beginning of the drawing cycle and sets the scale appropriately. The `Measurable` interface in the display package provides this functionality and is defined as follows:

```

1 public interface Measurable extends Drawable {
2     public double getXMin();
3     public double getXMax();
4     public double getYMin();
5     public double getYMax();
6     public boolean isMeasured(); // set to true if the measure is
7 }
```

Note that the `isMeasured` method should return false if the object, such as an empty dataset, does not have meaningful minimum and maximum values to report.

The measurable interface is very flexible. We can, for example, define a `MeasuredCircle` class that reports the center of the circle as its minimum and maximum values.

```

1 public class MeasuredCircle extends Circle implements Measurable
2 {
3     boolean enableMeasure = true;
4     // x and y are defined in the circle superclass.
5     public MeasuredCircle (double x, double y) { super (x, y); }
6
7     public boolean isMeasured () { return enableMeasure; }
8     public double getXMin () { return x; }
9     public double getXMax () { return x; }
10    public double getYMin () { return y; }
11    public double getYMax () { return y; }
}
```

If an ensemble of measured circles is created and these circles are added to a drawing panel, then the drawing panel will always rescale itself so that at least the center of every circle is visible.

Another simple but instructive example of `Measurable` uses `Dataset` to produce a strip chart recorder by overriding `getMin` value so that a fixed *x*-interval is obtained.

```

1 public Stripchart extends Dataset {
2     double range = 10;
3     public double getXMin() { return xmax-range; }
4 }
```

## 4.8 Interactive

67

Because `Dataset` already implements `Measurable`, additional methods need not be defined. A more robust strip chart would trim excess data so as to limit the size of the internal data arrays. See the `Stripchart` class in the `display` package for the code for such a class.

**4.8 ■ INTERACTIVE**

It is often desirable for a user to interact with a program using a mouse, which can be done using an interactive drawing panel, `InteractivePanel`, in combination with the `Interactive` interface and the `InteractiveMouseListener` interface. The following main method shows how easy it is to create a circle and a rectangle that can be positioned within an interactive panel by clicking and dragging using the mouse.

```

1 import org.opensourcephysics.display.*;
2
3 public class InteractiveShapeApp {
4     public static void main(String[] args) {
5         DrawingPanel panel = new PlottingPanel("x","y","Interactive
6             Demo");
7         DrawingFrame frame = new DrawingFrame(panel);
8         panel.setPreferredMinMax(-10,10,-10,10);
9         // creates two interactive drawables and adds them to the
10            panel
11         panel.addDrawable(new InteractiveCircle(0, 0));
12         InteractiveShape
13             ishape=InteractiveShape.createRectangle(3,4,2,2);
14         panel.addDrawable(ishape);
15         frame.setVisible(true);
16         frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
17     }
18 }
```

`InteractivePanel` is a subclass of `DrawingPanel` that collects mouse events. Because the `PlottingPanel` class is a subclass of `InteractivePanel`, it too supports the interactions. The interactive panel collects incoming events but only stores the last event. It notifies a mouse handler that an action has occurred. The handler's `handleMouseAction` method has the option of responding to this action. Because the interactive drawing panel's default handler is the panel itself, simple programs are not required to handle the mouse action. Interactive objects that are added to an `InteractivePanel` or a subclass are draggable without additional programming.

**Interactive Mouse Handler**

The `InteractiveMouseListener` interface enables objects to define their own interactive mouse actions.

```

1 public interface InteractiveMouseHandler {
2     public void handleMouseAction(InteractivePanel panel,
3         MouseEvent evt);
}

```

Interactive mouse handlers register their interest in events using an interactive panel's `setInteractiveMouseHandler` method. The object then receives all interactive mouse actions.

```

1 InteractivePanel interactivePanel = new InteractivePanel();
2 interactivePanel.setInteractiveMouseHandler(interactiveMouseHandler);

```

Mouse handlers determine the type of action using the interactive panel's `getMouseAction` method. A handler will often invoke the panel's default action method in order to drag objects. The following handler is typical. The method uses the panel's handler to drag the interactive object and then uses a switch statement to display the position of the mouse action.

```

1 public void handleMouseAction(InteractivePanel panel, MouseEvent
2     evt) {
3     panel.handleMouseAction(panel, evt); //drag the objects in the
4     // panel
5     // add custom actions
6     switch(panel.getMouseAction()) {
7         case InteractivePanel.MOUSE_PRESSED :
8             double x = panel.getMouseX();
9             double y = panel.getMouseY();
10            System.out.println("Mouse pressed at x="+x+" y="+y);
11            break;
12        case InteractivePanel.MOUSE_RELEASED :
13            double x = panel.getMouseX();
14            double y = panel.getMouseY();
15            System.out.println("Mouse released at x="+x+" y="+y);
16            break;
17    } // end of switch
18 }

```

### Interactive Interface

Interactive objects implement the `Interactive` interface to get and set an object's position, to enable dragging, and to determine the interactive "hot spots." The `findInteractive` method determines which hot spot contains the given coordinates. The interactive object compares the mouse coordinates to one or more locations and returns a reference to an object that can be used to drag, resize, or otherwise modify the selected object. If there is only one hot spot, it is likely that the object will return a reference to itself, `this`. If the coordinates are not within a hot spot, the method should return `null`.

## 4.8 Interactive

69

```

1 public interface Interactive extends Measurable {
2     public Interactive
3         findInteractive( DrawingPanel panel, int xpix, int ypix);
4     public void setEnabled(boolean enabled); // enables the
interaction
5     public boolean isEnabled();
6     public void setXY(double x, double y);
7     public void setX(double x);
8     public void setY(double y);
9     public double getX();
10    public double getY();
11 }
```

Listing 4.5 defines an interactive circle by extending the `MeasuredCircle` class which extends the `Circle` class introduced in Section 4.2.

Listing 4.5 Interactive circle class.

```

1 import java.awt.*;
2
3 public class InteractiveCircle extends MeasuredCircle
4     implements Interactive {
5     boolean enableInteraction = true;
6
7     public InteractiveCircle(double x, double y) {
8         super(x, y);
9     }
10
11    public void setEnabled(boolean _enableInteraction) {
12        enableInteraction = _enableInteraction;
13    }
14
15    public boolean isEnabled() {
16        return enableInteraction;
17    }
18
19    // Set and get methods for x and y are defined in superclass.
20    public void setXY(double _x, double _y) {
21        x = _x;
22        y = _y;
23    }
24
25    public Interactive findInteractive(DrawingPanel panel, int
26        _xpix,
27        int _ypix) {
28        if (!enableInteraction) {
29            return null;
30        }
31        int xpix = panel.xToPix(x); // convert the center x to pixels
32    }
33 }
```

```

31 int ypix = panel.yToPix(y); // convert the center y to pixels
32 if((Math.abs(xpix-xpix)<pixRadius)
33     &&(Math.abs(ypix-ypix)<pixRadius)) {
34     return this;
35 } else {
36     return null;
37 }
38 }
39 }
```

The `InteractiveHandlerApp` program on the CD demonstrates how to handle mouse actions from an interactive panel by creating an interactive circle. Click-dragging within the panel moves interactive objects and causes a message to appear.

In summary, a program must do the following to use interactive objects within an interactive panel:

1. Instantiate an interactive object and add it to an interactive panel.
2. Implement the interactive mouse handler in an appropriate class.
3. Instantiate the interactive mouse handler and register this handler with the interactive panel.

## 4.9 ■ AFFINE TRANSFORMATIONS

Affine transformations are powerful tools that can be used to quickly transform objects including points, shapes, and text. An affine transformation is a transformation in which parallel lines are still parallel after being transformed. Two examples of such transformations (written in matrix form) are rotation by an angle  $\theta$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad (4.1)$$

and translation along the  $x$  and  $y$  axes by  $a, b$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (4.2)$$

The most general affine transformation is a combination of scaling, translation, and rotation. It can be written as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (4.3)$$

## 4.9 Affine Transformations

71

The `AffineTransform` class in the `java.awt.geom` package defines affine transformations. Instances of this class are constructed as follows:

```

1  AffineTransform( double m00, double m10,
2                  double m01, double m11,
3                  double m02, double m12); .

```

Note that the `AffineTransform` class also defines convenience methods for constructing pure rotations, pure translations, and pure shears. The usual rules of matrix arithmetic apply when these transformations are combined.<sup>2</sup>

An affine transformation can be used to change a drawing panel's graphics context before drawing takes place. For example, we can define a draw method that renders a 80 pixel by 160 pixel rectangle at 30 degrees from a standard position by concatenating a rotation with the current graphics context's transformation.

```

1 public void draw(DrawingPanel panel, Graphics g) {
2     Graphics2D g2 = (Graphics2D) g;
3     AffineTransform oldAT=g2.getTransform();
4     AffineTransform at=g2.getTransform();
5     at.concatenate(AffineTransform.getRotateInstance(Math.PI/6.0,140,230));
6     // rectangle with center at (140, 230)
7     g2.drawRect(100,150,80,160);
8     g2.setTransform(oldAT);
9 }

```

This code fragment demonstrates a number of important features. First, because affine transformations uses the Java 2 API, the graphics context, `Graphics g`, must be cast to a Java 2D graphics context, `Graphics2D g2`. Second, although the graphics context that is passed to a drawable object is in pixel units, it may already have been transformed by the Java VM. Consequently, we save the original context before drawing and restore the original context after drawing. Third, because of the rules of matrix multiplication, we concatenate the rotation with the existing transformation. Note that the `getRotateInstance` method in the Java geometry package has two signatures. We have used a rotation instance that allows us to specify a rotation about the rectangle's center, (140, 230).

Although transforming the entire graphics context is sometimes useful, there are side effects. Scaling changes the location of every point except the origin. In other words, applying a scale instance not only changes the width and height of a rectangle, it also changes its location. In addition, scaling changes line thicknesses and text sizes. Text labels may, for example, become unreadable at high magnifications. It is usually better to instantiate a Java Shape using world coordinates and then use affine transformations to transform this shape into pixel coordinates. This technique is used in the next section to define OSP drawable shapes.

<sup>2</sup>See the online Java 2D documentation from Sun Microsystems and *Java 2D Graphics* by Jonathan Knudsen (O'Reilly 1999) for a more complete discussion of affine transformations.

### 4.10 ■ SHAPES

The Java 2D API defines a set of classes that can be used to create high-quality graphics using composition, image processing, anti-aliasing, and text-layout. We use a subset of the Java 2D API in the Open Source Physics display package. The `Shape` interface is one of the most useful.

The `java.awt.Shape` interface is a cornerstone of the Java 2D API, and we make it easy to incorporate objects that implement this interface into drawing panels. A shape represents a geometric object that has an outline and an interior. Concrete representations defined in the `java.awt.geom` package include `Rectangle2D`, `Ellipse2D`, `Line2D`, and `GeneralPath`. A typical shape is constructed by passing geometric information to a constructor. A Java 2D rectangle, for example, is constructed by passing upper left-hand corner  $(x, y)$  coordinates, width  $w$ , and height  $h$ .

```
1 // Shape is defined in the Java 2D API.
2 Shape rectangle = new Rectangle2D.Double(x,y,w,h);
```

Because a `DrawingPanel` object contains an affine world to pixel affine transformation, we can easily transform Java 2D shapes before they are rendered within a drawing panel. The `DrawableShape` class uses the world to pixel transformation to render shapes whose position and size have been specified using world coordinates. Note that the location of a Java 2D shape is usually the top-left corner of the object, but we can offset the shape's origin by half the width and half the height to position it using the geometric center. In the following code fragment we instantiate a Java 2D `Rectangle` with a width of three and a height of one with its top-left corner at  $(-1.5, -0.5)$ . The  $(x, y)$  position of the drawable shape's center is  $(-5, -4)$  in world coordinates.

```
1 // DrawableShape draws Java 2D shapes in a DrawingPanel
2 DrawableShape rect =
3     new DrawableShape(new
4         Rectangle2D.Double(-1.5,-0.5,3,1),-5,-4);
5 drawingPanel.addDrawable(rect);
```

Having to first create a Java 2D shape is awkward. In order to facilitate the creation of simple geometric shapes, the `DrawableShape` class and `InteractiveShape` class contain static (factory) methods to create common shapes, such as circles and rectangles, whose  $x$ - $y$  locations are set to the geometric center (see Table 4.3). A program that creates a rectangle with a width of four and a height of five centered at  $(-3, -4)$  is shown in Listing 4.6.

**Listing 4.6** `DrawableShapeApp` tests the `DrawableShape` class by creating and manipulating shapes.

```
1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.*;
3 import java.awt.Color;
```

## 4.10 Shapes

73

**TABLE 4.3** Static (factory) methods in the `DrawableShape` class and `InteractiveShape` class create simple geometric shapes.

<code>org.opensourcephysics.display.DrawableShape</code>	<code>org.opensourcephysics.display.InteractiveShape</code>
<code>createArrow</code>	Creates an arrow with the specified center, horizontal component, and vertical component.
<code>createCircle</code>	Creates a circle with the given center and radius.
<code>createEllipse</code>	Creates a ellipse with the given center, width, and height.
<code>createRectangle</code>	Creates a rectangle with the given center, width, and height.
<code>createSquare</code>	Creates a circle with the given center, and width.

```

4
5 public class DrawableShapeApp {
6     public static void main(String[] args) {
7         PlottingPanel panel = new PlottingPanel("x", "y", null);
8         panel.setSquareAspect(true);
9         DrawingFrame frame = new DrawingFrame(panel);
10        DrawableShape rectangle = DrawableShape.createRectangle(-3,
11                      -4, 4,
12                      5);
13        rectangle.setTheta(Math.PI/4);
14        Color fillColor = new Color(255, 128, 128, 128);
15        Color edgeColor = new Color(255, 0, 0, 255);
16        rectangle.setMarkerColor(fillColor, edgeColor);
17        panel.addDrawable(rectangle);
18        DrawableShape circle = DrawableShape.createCircle(3, 4, 6);
19        panel.addDrawable(circle);
20        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
21        frame.setVisible(true);
22    }
}

```

The position of a shape can be changed after it is created using accessor methods such as `setX`, `setY`, and `setXY`. These methods move the drawable shape's origin. The `setTheta` sets the orientation.

Another approach is to transform the shape using an affine transformation. The following code fragment illustrates the difference between transforming transforming a `DrawableShape` and transforming an internal Java 2D Shape.

```

1 DrawableShape rectangleOne =
2           DrawableShape.createRectangle(0,0,2,3);
3 DrawableShape rectangleTwo =
4           DrawableShape.createRectangle(0,0,2,3);

```

```

3 rectangleOne.setXY(6,8);
4 rectangleTwo.transform(AffineTransform.getTranslateInstance(6,8));
5 System.out.println(rectangleOne.getX()); // prints 6
6 System.out.println(rectangleTwo.getX()); // prints 0
7 rectangleOne.setTheta(Math.PI/4); // rotates about (6,8)
8 rectangleTwo.setTheta(Math.PI/4); // rotates about (0,0)

```

Both rectangles instantiated with center at (0,0) and have a width of 2 and height of 3. These rectangles are then translated so that they are drawn with centers at (6,8). The last transformation will, however, produce dramatically different results because the rotation is performed about the drawable shape's origin. The first drawable shape has its origin shifted whereas the second rectangle has shifted the center of the internal Java 2D object. The `transform` method applies an affine transformation, such as a shear and a rotation, directly to the internal geometry, whereas set commands such as `setXY` are applied to an object that positions the shape.

Java shapes are not restricted to simple geometric objects. The `GeneralPath` class in the `java.awt.geom` package defines a shape using line segments and Bézier curves. The code fragment shown below uses this class to define a triangle using a sequence of `moveTo` and `lineTo` methods. The triangle is converted to a drawable and added to a drawing panel.

```

1 // GeneralPath is defined in java.awt.geom package and
   implements Shape
2 GeneralPath path = new GeneralPath();
3 path.moveTo(3,0);
4 path.lineTo(0,3);
5 path.lineTo(0,-3);
6 path.closePath();
7 DrawableShape triangle = new DrawableShape(path,0,0);
8 drawingPanel.addDrawable(triangle);

```

A shape's border is a path and this path can be accessed using the `getPathIterator` method in the `Shape` interface. A rectangle's perimeter, for example, consists of a series of four line segments and each iteration will return the next segment end point. This paradigm allows us to compute path integrals as shown in Listing 4.7. Note that because we have flattened the shape's path, only straight line segments are returned by the iterator.<sup>3</sup>

Listing 4.7 A path integral can be calculated using an iterator.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.InteractiveShape;
3 import java.awt.geom.PathIterator;
4
5 public class PathIntegralApp {

```

<sup>3</sup>If the `flatten` parameter is omitted when getting the path iterator, then a shape may return quadratic or cubic line segments and a more sophisticated integration algorithm must be employed.

## 4.11 Interactive Shapes

75

```

6  public static void main(String [] args) {
7      InteractiveShape circle = InteractiveShape.createCircle(-2,
8          2, 1);
9      // get an iterator with line flatness less than 0.001
10     PathIterator it = circle.getShape().getPathIterator(null,
11         0.001);
12     double [] coord = new double [6];
13     double
14         sum = 0, x1 = 0, y1 = 0, xstart = 0, ystart = 0;
15     while (!it.isDone()) {
16         switch(it.currentSegment(coord)) {
17             case PathIterator.SEG LINETO :
18                 sum += Math.sqrt((x1-coord[0])*(x1-coord[0])
19                     +(y1-coord[1])*(y1-coord[1]));
20                 x1 = coord[0];
21                 y1 = coord[1];
22                 break;
23             case PathIterator.SEG MOVETO :
24                 x1 = coord[0];
25                 y1 = coord[1];
26                 xstart = x1; // start of the path
27                 ystart = y1;
28                 break;
29             case PathIterator.SEG CLOSE :
30                 sum += Math.sqrt((x1-xstart)*(x1-xstart)
31                     +(y1-ystart)*(y1-ystart));
32                 xstart = x1;
33                 ystart = y1;
34                 break;
35             default :
36                 System.out.println("Segment Type not supported. Type="
37                     +it.currentSegment(coord));
38             }
39             it.next();
40         }
41     }

```

## 4.11 ■ INTERACTIVE SHAPES

The display package defines two types of interactive shapes, `InteractiveShape` and `BoundedShape`. An `InteractiveShape` is similar to a `DrawableShape` except that its x-y position can be adjusted by click-dragging within the an interactive panel. A `BoundedShape` can be dragged, rotated and resized.

Listing 4.8 The `InteractiveShapeApp` program tests the `InteractiveShape` class.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.*;
3 import javax.swing.JFrame;
4
5 public class InteractiveShapeApp {
6     public static void main(String[] args) {
7         PlottingPanel panel = new PlottingPanel("x", "y",
8             "Interactive Demo");
9         panel.setPreferredMinMax(1, 10, 1, 10);
10        DrawingFrame frame = new DrawingFrame(panel);
11        // create interactive shapes and add them to the panel
12        InteractiveShape ishape =
13            InteractiveShape.createRectangle(3, 4,
14                2, 2);
15        panel.addDrawable(ishape);
16        InteractiveShape arrow = InteractiveShape.createArrow(3, 4,
17            1, 5);
18        panel.addDrawable(arrow);
19        frame.setVisible(true);
20        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}

```

Because it can be computationally expensive (and visually confusing) to perform these mouse actions if a panel contains multiple bounded shapes, the user must first double-click a `BoundedShape` object to select it. Selecting a shape highlights it by showing a light-blue bounding box and one or more *hotspots*. Click-dragging the hotspot changes an object's properties. Note that only the *x-y* drag interaction is enabled when a `BoundedShape` is instantiated. Other interactions are explicitly enabled using accessor methods such as `setRotateDrag` and `setWidthDrag`.

Listing 4.9 `BoundedShapeApp` tests the `BoundedShape` class.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.*;
3
4 public class BoundedShapeApp {
5     public static void main(String[] args) {
6         PlottingPanel panel = new PlottingPanel("x", "y",
7             "Bounded Shape Demo");
8         DrawingFrame frame = new DrawingFrame(panel);
9         panel.setPreferredMinMax(-10, 10, -10, 10);
10        BoundedShape bShape = BoundedShape.createBoundedRectangle(3,
11            4, 5,
12                6);
13        bShape.setRotateDrag(true);
}
}

```

## 4.12 Text

77

```

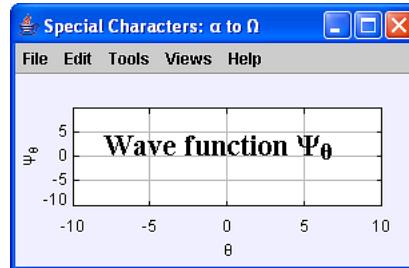
13 panel . addDrawable( bShape );
14 bShape = BoundedShape . createBoundedEllipse( -3, -4, 6, 6 );
15 bShape . setHeightDrag( true );
16 bShape . setWidthDrag( true );
17 panel . addDrawable( bShape );
18 frame . setVisible( true );
19 frame . setDefaultCloseOperation( javax . swing . JFrame . EXIT_ON_CLOSE );
20 }
21 }
```

Although shapes usually specify their *x-y* location using world coordinates, it is sometimes convenient to use pixels to specify their size. It would, for example, be awkward if text, images, or dataset point markers changed their appearance when the coordinate scale changed. The `InteractiveShape` class and `BoundedShape` class implement both pixel and world size units. The following code fragment creates a rectangle a  $40 \times 80$  pixel rectangle at location (2,3) in world coordinates.

```

1 InteractiveShape
2     ishape=InteractiveShape . createRectangle( 2,3,40,80 );
3 ishape . setPixelSized( true );
```

## 4.12 ■ TEXT



**FIGURE 4.4** The `DrawableTextLine` class supports special characters, superscripts, and superscripts using a *TeX*-like markup language.

Java 1.4 renders text through software by forming characters pixel by pixel. We can do better if the characters are always the same, in the same font, at the same size, and in the same color. We render the text as an image and then rotate and scale the image using affine transformations. The `DrawableTextLine` class implements this drawing technique. In addition, we have added processing that allows us to incorporate Greek characters and to change the size and location of text

to produce superscripts and subscripts using a syntax that mimics the *T<sub>E</sub>X* markup language. Subscripts begin with an underscore `_` and are enclosed in braces `{ }`. Superscripts begin with a caret `^` and are also enclosed in braces.

**Listing 4.10** OSP supports special characters using a syntax that mimics the *T<sub>E</sub>X* markup language.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.frames.*;
4
5 public class GreekCharApp {
6     public static void main(String[] args) {
7         PlotFrame frame = new PlotFrame("$\\theta",
8                                         "$\\Psi_{\\theta}",
9                                         "Special Characters: $\\alpha$ to
10                                        $\\Omega$");
11        String inputStr = "Wave function $\\Psi_{\\theta}";
12        DrawableTextLine textLine = new DrawableTextLine(inputStr,
13            -8, 0);
14        frame.addDrawable(textLine);
15        textLine.setFontSize(22);
16        textLine.setFontStyle(java.awt.Font.BOLD);
17        frame.setVisible(true);
18        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
19    }
20}
```

Greek characters are specified using a double backslash followed by the character name enclosed within matching \$ delimiters. We use two backslash characters because a single backslash is the Java escape sequence and has special meaning. The `DrawableTextLine` class, axis labels, and frame titles support the commonly used Greek characters shown in Table 4.4. Additional *unicode* character mappings are defined in the `GUIUtil` class in the `display` package. As in *T<sub>E</sub>X*, if the Greek character name begins with a capital letter the capital Greek symbol is used.

### 4.13 ■ MEASURED IMAGES AND SNAPSHOTS

Images can be added to a drawing panel using the `MeasuredImage` class. These images can either be computed by the program or loaded as a resource from a disk drive or URL using the `ResourceLoader`.

Image pixels are data just as array elements are data. In fact, it is possible to access image pixels using the Java `BufferedImage` class in the `java.awt.image` package. The `MeasuredImage` class in the `display` package accepts a `BufferedImage` and assigns a scale that converts image pixels to and from a drawing panel's world coordinates. Listing 4.11 shows how the `MeasuredImage` class

## 4.13 Measured Images and Snapshots

79

**TABLE 4.4** Greek characters are defined in the `GUIUtil` class in the `display` package.

	Lowercase	Uppercase	
$\alpha$	<code>\alpha</code>	$\Gamma$	<code>\Gamma</code>
$\beta$	<code>\beta</code>	$\Delta$	<code>\Delta</code>
$\gamma$	<code>\gamma</code>	$\Theta$	<code>\Theta</code>
$\delta$	<code>\delta</code>	$\Pi$	<code>\Pi</code>
$\epsilon$	<code>\epsilon</code>	$\Sigma$	<code>\Sigma</code>
$\zeta$	<code>\zeta</code>	$\Phi$	<code>\Phi</code>
$\eta$	<code>\eta</code>	$\Psi$	<code>\Psi</code>
$\theta$	<code>\theta</code>	$\Omega$	<code>\Omega</code>
$\iota$	<code>\iota</code>	$\Xi$	<code>\Xi</code>
$\kappa$	<code>\kappa</code>		
$\lambda$	<code>\lambda</code>		
	$\omega$	$\omega$	<code>\omega</code>

is used to display an image that is computed within the program. Because `MeasuredImage` implements the `Measured` interface, the drawing panel sets its coordinate scale to match that given in the `MeasuredImage` constructor if the panel's `autoscale` option is enabled.

Listing 4.11 Measured image test program.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.*;
3 import java.awt.*;
4 import java.awt.image.BufferedImage;
5
6 public class MeasuredImageApp {
7     static final int SIZE = 20;
8
9     public static void main(String[] args) {
10        PlottingPanel panel = new PlottingPanel("x", "y",
11                                         "Measured Image");
12        DrawingFrame frame = new DrawingFrame(panel);
13        panel.setPreferredMinMax(-2, 2, -2, 2);
14        BufferedImage image = new BufferedImage(SIZE, SIZE,
15                                         BufferedImage.TYPE_INT_ARGB);
16        Graphics g = image.getGraphics();
17        g.setColor(Color.RED);
18        g.fillRect(0, 0, SIZE, SIZE);
19        g.dispose();
20        int color = 0xFF00FF00; // opaque and green in ARGB color
21        space
22        for(int i = 0; i < SIZE; i++) {
23            image.setRGB(i, 0, color);
24            image.setRGB(i, i, color);
25            image.setRGB(i, SIZE-1, color);

```

```

25     image.setRGB(i, SIZE-i-1, color);
26 }
27 MeasuredImage mi = new MeasuredImage(image, -1, 1, -1, 1);
28 panel.addDrawable(mi);
29 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30 frame.setVisible(true);
31 frame.setSize(500, 400);
32 }
33 }
```

The drawing panel uses a measured image to capture a *snapshot* of its visual representation. The `snapshot` method invokes the `getRenderedImage` method and places a copy of this image into a new frame. Listing 4.12 shows how this method is implemented. This `snapshot` method can be invoked from within a program or by using the drawing panel's snapshot menu item.

Listing 4.12 The drawing panel's snapshot method.

```

1 public void snapshot() {
2     DrawingPanel panel = new DrawingPanel();
3     DrawingFrame frame = new DrawingFrame(panel);
4     frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
5     frame.setKeepHidden(false);
6     panel.setSquareAspect(false);
7     int w = (isVisible())
8         ? getWidth()
9         : getPreferredSize().width;
10    int h = (isVisible())
11        ? getHeight()
12        : getPreferredSize().height;
13    if((w==0) || (h==0)) {
14        return;
15    }
16    BufferedImage snapimage = new BufferedImage(w, h,
17        BufferedImage.TYPE_INT_ARGB);
18    render(snapimage);
19    MeasuredImage mi =
20        new MeasuredImage(snapimage, pixToX(0), pixToX(w),
21            pixToY(h), pixToY(0));
22    panel.addDrawable(mi);
23    panel.setPreferredMinMax(pixToX(0), pixToX(w), pixToY(h),
24        pixToY(0));
25    panel.setPreferredSize(new Dimension(w, h));
26    frame.setTitle("Snapshot");
27    frame.pack();
28    frame.setVisible(true);
29 }
```

Java *resources* are ancillary files that are used within a program. They may be loaded from the disk or the internet and can be almost anything including

## 4.14 Effective Java

81

text files such as html pages or binary files such as images or sound. Because Java uses different loading mechanisms depending on whether the program is running as an application or as an applet, we have created the `ResourceLoader` class in the tools package. Listing 4.13 uses the `ResourceLoader` to load an image containing a Mercator projection of Earth. Note that the `MeasuredImage` is assigned a scale that determines its size in the drawing panel.

Listing 4.13 Image resource loader test program.

```

1 package org.opensourcephysics.manual.ch04;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.tools.ResourceLoader;
4 import java.awt.image.BufferedImage;
5 import javax.swing.JFrame;
6
7 public class LoadImageApp {
8     public static void main(String[] args) {
9         PlottingPanel panel = new PlottingPanel("x", "y", "Load
10            Image");
11        DrawingFrame frame = new DrawingFrame(panel);
12        String s = "org/opensourcephysics/manual/ch04/earthmap.gif";
13        BufferedImage earth = ResourceLoader.getBufferedImage(s);
14        MeasuredImage mi = new MeasuredImage(earth, -180, 180, -90,
15            90);
16        panel.addDrawable(mi);
17        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18        frame.setVisible(true);
19        frame.setSize(500, 400);
20    }
21 }
```

If the program is an application, the `ResourceLoader` searches for the resource in the specified directory; if the program is an applet, the program searches for the resource starting at the applet's codebase. If the resource cannot be found in these locations, the loader searches in the jar file from which the program was loaded. The `ResourceLoader` caches resources in memory so that a requested file is only loaded once.

## 4.14 ■ EFFECTIVE JAVA

The book *Effective Java* by Joshua Bloch provides a useful guide to how the Java language is best used in practice. Three of his fifty-seven rules help us to decide when to use inheritance and when to use interfaces.

- Prefer interfaces to abstract classes.
- Favor composition over inheritance.

- Design and document for inheritance or else prohibit it.

It is generally considered safe to use inheritance within a package where all the classes are under the control of the same programmers. Carefully designed and documented class hierarchies, such as the user interface library distributed by Sun, can easily be subclassed. We have done so in designing the drawing panel, the interactive panel, and the plotting panel in the display package. The `DrawingPanel` adds world coordinates, coordinate transformations, and the ability to manage drawable objects to the basic panel class, the `InteractivePanel` adds a framework for repositioning objects using the mouse, and the `PlottingPanel` adds axes, titles, and logarithmic scales.

Inheritance is a powerful way to achieve code reuse, but it breaks encapsulation. A subclass all too often depends on the implementation details of its superclass if for no other reason than it invokes the superclass constructor. Because of the hierarchy shown on Figure 4.2, changing the world to pixel coordinate transformation in the drawing panel might break logarithmic axes in the plotting panel.

In addition to allowing and encouraging encapsulation, interfaces are easier to use because any class can be retrofitted to implement an interface. For example, after we solve for the trajectory of a baseball using a differential equation, we can create a visualization by implementing the `Drawable` interface anywhere within the baseball class hierarchy. Adding the baseball to a drawing panel (and redrawing the panel containing the baseball as the solution is generated) produces an animation as shown in Chapter 7. The `Drawable` interface makes it possible for the baseball object to display itself in a drawing panel. Similarly, the `Measurable` interface makes it possible for an object to rescale a drawing panel, and the `Interactive` interface makes it possible to reposition an object in a drawing panel. In subsequent chapters we introduce additional interfaces including the `Control` interface to store and retrieve values from a graphical user interface and the `ODE` interface to solve differential equations.

It is reasonable to declare variables using an interface rather than a class. For example, rather than defining a variable to be an interactive circle, we can define a variable that references an `Interactive` object if only the methods in the interface are required.

```
1 // ball at location x = 20, y = 0
2 Interactive baseball = new InteractiveCircle(20,0);
```

The one disadvantage to interfaces is that it is easier to evolve a class hierarchy or an abstract class than it is to evolve an interface. If we add a new method to the particle class named `launch` that gives particles an initial velocity, all existing subclasses will be able to perform this method. If we were to add the `launch` method to the `Drawable` interface, we would have to add this method to all classes that implement this interface. This example is contrived, but the problem is real. If other programmers use our particle class, they will probably not notice if the new `launch` method exists. But if `launch` is added to the `Drawable` interface, any code that uses this interface will cease to work.

The safest approach to object design is to use a technique known as *composition*. A new class is defined that implements an interface, such as `Drawable`. This class contains private references to other objects that implement the necessary methods. Methods in the new class implement interfaces by invoking methods in the private variables. Defining a method in an object that invokes a method in another object is known as *forwarding*. Although this technique may not always be possible or efficient, it decouples a class from all implementation details.<sup>4</sup>

## 4.15 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch04` package.

### AffineTestApp

`AffineTestApp` demonstrates how to transform 2D shapes using the Java 2D `AffineTransform` class. See Section 4.9.

### BoundedShapeApp

`BoundedShapeApp` demonstrates how to use the `BoundedShape` class by creating a rectangle and an ellipse. See Section 4.9.

### CircleAndArrowApp

`CircleAndArrowApp` tests the `DrawingPanel` class by creating two drawable objects. This program is described in Section 4.3.

### BoundedShapeApp

`BoundedShapeApp` tests the `BoundedShape` class by creating a rectangle and an ellipse. Click on a bounded shape to select it and then drag a hot spot. See Section 4.11.

### DrawableShapeApp

`DrawableShapeApp` demonstrates how to create, translate, and rotate a `DrawableShape`. This program is described in Section 4.10.

### DrawingApp

`DrawingApp` demonstrates the OSP drawing framework. See Section 4.1.

### GreekCharApp

`GreekCharApp` demonstrates Greek characters and other drawing features such as superscripts and subscripts. See Section 4.12.

<sup>4</sup>See *Effective Java* by Joshua Bloch, Addison Wesley (2001).

**InteractiveHandlerApp**

InteractiveHandlerApp demonstrates how to handle mouse actions from an interactive drawing panel. Pressing the mouse within the panel causes a message to appear. This program is described in Section 4.8.

**InteractiveShapeApp**

InteractiveShapeApp tests the InteractiveShape class by creating a circle and a square. Users can drag these shapes within the panel. This program is described in Section 4.8.

**LoadImageApp**

LoadImageApp loads an image as a resource and adds it to a drawing panel using the MeasuredImage class. See Section 4.13

**MeasuredImageApp**

MeasuredImageApp creates an image, assigns a measure in world units, and places the image in a plotting panel. This program is described in Section 4.13.

**MessageApp**

MessageApp tests message boxes and adds a component to the glass pane. See Section 4.5.

**PathIntegralApp**

PathIntegralApp demonstrates how to calculate a path integral using an iterator. This program is described in Section 4.10.

**StripChartApp**

StripChartApp demonstrates how to use the Measurable interface to create a strip chart. This program is described in Section 4.7.

## CHAPTER

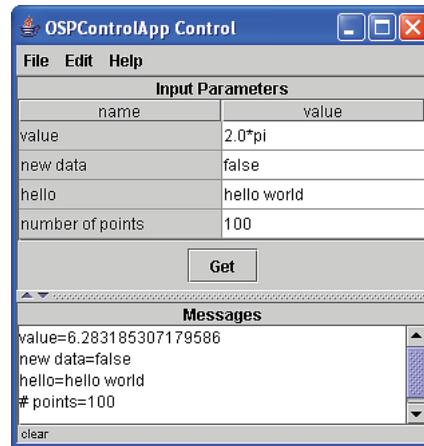
## 5

## Controls and Threads

©2005 by Wolfgang Christian

We define a framework based on the `Control` interface for creating graphical user interfaces. Custom user interfaces are then constructed using *Easy Java Simulations* components. Sample programs are located in the `org.opensourcephysics.-manual.ch05` package unless otherwise stated.

### 5.1 ■ OVERVIEW



**FIGURE 5.1** An `OSPControl` stores variables using a table.

Physics models require parameters (such as mass) that are held constant during a calculation and initial values of state variables (such as position and velocity) that evolve during a calculation. These parameters and state variables are usually stored in a model using `int` and `double` identifiers. However, users cannot access these variables directly. Users interact with a graphical user interface that collects data (usually from a keyboard) and passes this data to the model in response to an action such as a button click. This user interface object is an example of a *control*.

A control is an object that stores data and invokes methods in another object. Controls often have a graphical user interface such as shown in Figure 5.1,

whereas controlled objects are often complex models whose job it is carry out a computation or simulation. In principle, any object can have a control. In practice, the most important control is the control that starts, stops, and initializes a program.

The `OSPControlApp` program shown in Listing 5.1 creates an `OSPControl` object. Because the control must store a reference to the model in order for the Get button to invoke methods in the model, we pass a reference to `this` to the control's constructor. The program's `reset` method stores default values for four variables in the control. These variables can be edited and are read when the user clicks the Get button. Run the program and edit the input values. Note the behavior of the control if incorrect data are typed into the table and the button is pressed.

**Listing 5.1** The `OSPControlApp` program creates the user interface shown in Figure 5.1. It reads and displays values in response to button clicks.

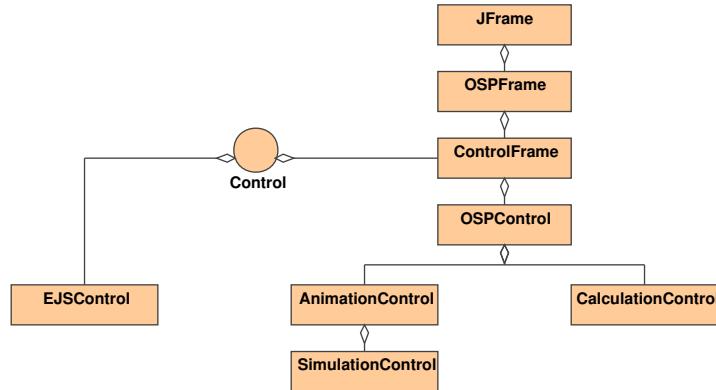
```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.controls.*;
3
4 public class OSPControlApp {
5     OSPControl control = new OSPControl(this); // creates the
6         control
7
8     public OSPControlApp() {
9         control.addButton("getValues", "Get",
10             "Gets values from the control.");
11         reset(); // stores default values
12     }
13
14     public void getValues() {
15         double val = control.getDouble("value");
16         control.println("value="+val);
17         boolean newData = control.getBoolean("new data");
18         control.println("new data="+newData);
19         String str = control.getString("hello");
20         control.println("hello="+str);
21         int n = control.getInt("number of points");
22         control.println("# points="+n);
23     }
24
25     public void reset() {
26         control.setValue("value", "2.0*pi");
27         control.setValue("new data", false);
28         control.setValue("hello", "hello world");
29         control.setValue("number of points", 100);
30     }
31
32     public static void main(String[] args) {
33         new OSPControlApp();
34     }

```

34 { }

## 5.2 ■ CONTROL INTERFACE



**FIGURE 5.2** The Open Source Physics library contains a number of pre-defined controls and one general purpose control framework known as *EJS*.

The Open Source Physics library contains a number of pre-defined controls and one general purpose control framework known as *EJS* as shown in Figure 5.2. These controls implement the *Control* interface. The purpose of this interface is to enable the two-way communication needed to store and retrieve values from an object using *set* and *get* methods.

The *Control* interface may seem complex but the concept is very simple. There is a *setValue* method with signatures to store primitive data types and generic objects. There are five accessor methods to retrieve primitive data types and objects from the control. There are four methods to display and clear messages in a console-like text area. And there are four utility methods for specialized applications.

Listing 5.2 The *Control* interface in the `org.opensourcephysics.controls` package.

```

1  public interface Control {
2      // stores values in the control
3      public void setValue(String name, Object val);
4      public void setValue(String name, double val);
5      public void setValue(String name, int val);
6      public void setValue(String name, boolean val);
  
```

```

7   // retrieves values from the control
8   public int getInt(String name);
9   public double getDouble(String name);
10  public String getString(String name);
11  public boolean getBoolean(String name);
12  public boolean getObject(String name);
13
14
15  // shows messages in the control's console
16  public void println(String s);
17  public void println();
18  public void print(String s);
19  public void clearMessages();
20
21  // utility methods
22  public void clearValues();           // clears name-value pairs
23  public void calculationDone(String message);
24  public void setLockValues(boolean lock); // locks interface
25  public java.util.Collection getPropertyNames(); // gets names
26 }

```

The superclass for the standard OSP controls is the `OSPControl` class. A program uses the control's `setValue` method to store data by passing the name of the variable and an initial value. The control then displays these name-value pairs using a table that can be edited. The user enters new values into the table and presses the Enter (Return) key. Note that the value field's background becomes yellow until the data has been entered. The new value can then be read by invoking an accessor method such as `getDouble` or `getInteger`.

OSP controls can store primitive data types, objects, and arrays. For example, if we want the user interface to provide access to the mass  $m$ , the initial position  $x_0$  of a particle, and the initial velocity of a particle  $\mathbf{v} = (2, 3)$ , we execute the following statements:

```

1 control.setValue("x0", 2.0);           // stores
                                         primitive data type
2 control.setValue("y0", "sqrt(2.0)");    // stores
                                         arithmetic expression
3 control.setValue("mass", new Double(3)); // stores object;
                                         restricts input
4 control.setValue("v", new double[]{2.0,3.0}); // stores array

```

The control now holds a copy of these variables and displays them in a user interface. The user edits the values in the control, but these changes are not reflected in the model until they are read from the control. Note that the control is able to store mathematical expressions as strings and to convert them to numbers. The user can type almost anything if the value contains a `String` or a primitive data type, but if the value is wrapped in an object, such as `Double` or `Integer`, then the user cannot enter any other data type. The user cannot, for example, enter an

## 5.2 Control Interface

89

arithmetic expression for the mass in the control.

In addition to arrays, compound objects can be stored in a control if the object has an XML loader. (Many objects in the OSP library do. See Chapter 12.) The control table displays the name of the object on a pale green background. Double-clicking on the cell shows an inspector window that displays the object's properties.

```
1 control.setValue("Circle",new InteractiveCircle());
```

Objects must be cast to the correct type when they are read from the control.

```
InteractiveCircle circle= (InteractiveCircle)  
control.getObject("Circle");
```

Reading values from a control is often initiated by an action such as a button click. Buttons are added to standard OSP controls by passing a method name, a button title, and a description (hint) to the  `addButton` method.

```
control.addButton("calculate", "Calculate", "Does a  
calculation.");
```

The button's action method invokes the `calculate` method which does something like the following:

```
1 public void calculate () {  
2     double x = control.getDouble("x0");  
3     double y = control.getDouble("y0");  
4     double m = control.getDouble("mass");  
5     double [] v = (double []) control.getDouble("v");  
6     speed = Math.sqrt(v[0]*v[0]+v[1]*v[1]);  
7     // continue the calculation  
8 }
```

The `OSPControl` class is able to store and retrieve data in multiple formats. Storing a floating point number and retrieving an integer or a string is allowed. So is storing an arithmetic expression and retrieving an integer or a double. Character strings such as `pi` or `2*sqrt(3)` are evaluated to produce numbers when the `getDouble` method is invoked. If the user mistypes the expression, the conversion will fail. This failure is indicated in the control by turning the input field's background pale red.

As the calculation progresses, the model can modify the control's copy of the data by invoking the `setValue` method. The control will reflect this change, but constantly updating the control may demand considerable processing power. If multiple values are being accessed using set or get methods, performance can be improved by locking and unlocking the control, which is done as follows:

```
1 control.setLockValues(true); // locks values in  
   user interface  
2 control.setValue("x0", 2.0); // stores  
   primitive data type
```

```

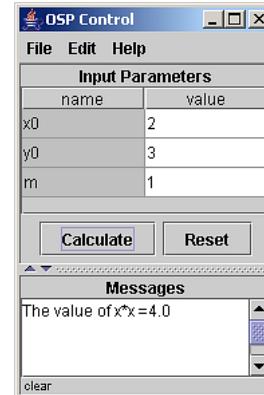
3 control.setValue("y0", "sqrt(2.0)");           // stores
      arithmetic expression
4 control.setValue("mass", new Double(3));        // stores object
5 control.setValue("v", new double[]{2.0,3.0});    // stores array
6 control.setLockValues(false);                   // updates user
      interface

```

A locked control stores and returns its current values but is not required to update its graphical user interface until it is unlocked.

Because user actions can occur at any time, synchronization errors may result when the control sends data to a model as explained in Section 5.4. It is the programmer's responsibility to insure that actions can safely modify the model's data. The best solution is to read values after all computation has stopped, but these design considerations are entirely up to the programmer.

### 5.3 ■ CALCULATIONS



**FIGURE 5.3** A simple control for use with a Calculation. The control stores three variables and shows the result of a calculation.

The CalculationControl shown in Figure 5.3 is designed to perform a short computation. This control inherits the OSPControl user interface and adds two buttons that invoke methods in a Calculation. Because we want this control to interact with many different programs, we define an interface in Listing 5.3 with a one-to-one correspondence between the control's buttons and the calculations's methods.

Listing 5.3 The Calculation interface.

```

1 public interface Calculation {

```

## 5.3 Calculations

91

```

2 // button actions
3 public void calculate ();
4 public void resetCalculation ();
5
6 // register the control with the model
7 public void setControl (Control control);
8 }
```

Models that use a `CalculationControl` implement the `Calculation` interface. The calculate and reset buttons shown in Figure 5.3 invoke the corresponding methods in the `Calculation` object.

A `CalculationControl` must contain a reference to its `Calculation` because the control expects to invoke certain methods. On the other hand, the `Calculation` must contain a reference to its `Control` because it will set and read variables in the control. This two-way communication is established when the control and calculation are created in the application's main method.

```

1 public static void main (String [] args) {
2     Calculation calculation = new CalculationApp ();
3     Control control = new CalculationControl (calculation);
4     calculation.setControl (control);
5 }
```

Note the use of Java interfaces. The application's `main` method starts by creating an object named `calculation` that implements the `Calculation` interface. This object (model) contains the physics. Next, the application creates a `Control` and passes the calculation to the control. Finally, the `setControl` method establishes communication from the calculation to the control so that the calculation can access the control's variables. Because the process of creating a control and a calculation occurs again and again, we have defined the following static method in the `CalculationControl` class to automate the process of establishing control-model communication.

```

1 public static void main (String [] args) {
2     CalculationControl.createApp (new CalculationApp ());
3 }
```

The entire control-model communication mechanism can be seen in action by running the short test program shown in Listing 5.4.

Listing 5.4 A calculation test program.

```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.controls.*;
3
4 public class CalculationApp implements Calculation {
5     private Control control; // the control
6     private double x0; // a parameter
7 }
```

```

8  public void setControl(Control control) {
9    this.control = control;
10   resetCalculation();
11 }
12
13 public void calculate() {
14   x0 = control.getDouble("x0"); // read a parameter value
15   // do some calculations here.
16   control.println("The value of x*x =" +(x0*x0));
17 }
18
19 public void resetCalculation() {
20   control.clearMessages();
21   control.setValue("x0", 2); // set a parameter value
22   control.setValue("y0", 3); // set a parameter value
23   control.setValue("m", 1); // set a parameter value
24 }
25
26 public static void main(String[] args) {
27   CalculationControl.createApp(new CalculationApp());
28 }
29 }
```

The Calculation interface has an important limitation – it is unable to respond to the click of a button while the calculation is being performed. This lack of response is not a problem if the computation is short. However, if the computation takes a long time, the user might assume that the program has crashed or is an infinite loop. The best way of overcoming this problem is to use a button to spawn a Thread as described in the next section.

## 5.4 ■ THREADS

Traditional programs start by executing a line of code, then another line, then another. There may be times when the program jumps from one block of code to another, but the program always moves from one statement to the next and never enters a state where statements are executed independently. Java is not like that. Java is almost always trying to do more than one thing at a time and these calculations are independent. In particular, the graphical user interface should be independent of the calculation. Otherwise, the interface will freeze while a calculation is being performed. This problem can be overcome by using a powerful feature of modern computer languages known as a *thread*.

A Java Thread is an object that executes statements within a program. Every application begins by executing statements within the main method from the *main thread*. This thread can be paused using the `sleep` method as shown in Listing 5.5. The main thread *dies* when the main method exits. The main thread cannot be restarted, but other threads can take over and continue to execute statements

## 5.4 Threads

93

within the program. The most important of these is the *event dispatch thread* which invokes methods in response to keyboard and mouse events.

Listing 5.5 The main thread executes statements in the main method.

```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.frames.*;
4
5 public class MainThreadApp {
6     public static void main(String[] args) {
7         DisplayFrame frame = new DisplayFrame("x", "y", "Main Thread
8             App");
9         InteractiveShape rect = InteractiveShape.createRectangle(4,
10             3, 8,
11             10);
12         double
13             theta = 0, dtheta = 0.1;
14         frame.addDrawable(rect);
15         frame.setVisible(true);
16         frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
17         while(true) {
18             rect.setTheta(theta);
19             theta += dtheta; // increment theta
20             frame.repaint(); // repaint with new theta
21             try { // sleep for 100 milliseconds
22                 Thread.sleep(100);
23             } catch(InterruptedException ex) {}
24         }
25     }
26 }
```

Having multiple threads is similar to having multiple applications insofar as the operating system causes one thread to run and then another. But threads are different from multiple applications; threads are independent calculations (not simultaneous calculations) that access the same data. A typical application has one thread to handle user actions and another thread to run the computation. For example, a graph may be plotting data that is currently being computed, while the user is using the mouse to drag an object within the graph. Because threads share the same data, we need to be careful to make sure that they work in such a way that neither thread corrupts data that is being used by the other.

A thread invokes a method within an object that implements the `Runnable` interface. This interface consists of a single method, the `run` method, and the thread executes the code within this method. It is usually an error for a program to invoke the `run` method directly. You should instead create a thread.

Listing 5.6 The `Runnable` interface.

```

1 public interface Runnable {
```

```

2  public void run();
3 }
```

Listing 5.7 defines a `Runnable` object called `RotationApp` that contains a rectangular shape, a drawing panel, and a thread. The thread invokes the `run` method automatically soon after it starts. The object's `run` method is invoked by the thread and enters an infinite loop that increments the rectangle's orientation. The `run` method terminates when the window is closed and the application dies. Note that the program's main method creates two `RotationApp` objects and that these objects rotate independently.

Listing 5.7 A program with two animation threads. Each animation consists of a rectangle rotating within a drawing panel.

```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.display.*;
3
4 public class RotationApp implements Runnable {
5     DrawingPanel panel = new PlottingPanel("x", "y", "Rotating
6         Shape");
7     DrawingFrame frame = new DrawingFrame(panel);
8     InteractiveShape ishape = InteractiveShape.createRectangle(2,
9         1, 2,
10            1);
11    double
12        theta = 0, dtheta = 0.1;
13    Thread thread = new Thread(this);
14
15    public RotationApp(double dtheta) {
16        this.dtheta = dtheta;
17        panel.setPreferredMinMax(-5, 5, -5, 5);
18        panel.addDrawable(ishape);
19        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
20        frame.setVisible(true);
21        thread.start();
22    }
23
24    public void run() {
25        while(true) {
26            theta += dtheta;
27            ishape.setTheta(theta);
28            panel.repaint();
29            try {
30                Thread.sleep(100); // wait 100 millisecond
31            } catch(InterruptedException ie) {}
32        }
33    }
34    public static void main(String[] args) {
```

## 5.4 Threads

95

```

34     new RotationApp(0.1); // create an animation
35     new RotationApp(0.2); // create an animation
36   }
37 }
```

A thread should not monopolize the computer's processor without giving other threads a chance to run. This is done by invoking the `sleep` method from time-to-time within the `run` method. The argument passed to `sleep` is the number of milliseconds that the thread should wait before continuing the calculation. A good value for computational speed is 10 milliseconds. This interval gives the operating system (event dispatch thread) the opportunity to process keyboard and mouse and events. Because there is a tradeoff between flicker free animation and the time available to compute the physics, a good value for simulations is 100 milliseconds or 1/10 of a second.<sup>1</sup> Note that Java requires that the `sleep` method be enclosed in a `try` block to properly handle interrupt exceptions.

It is important to know how to end the `run` method, thereby stopping the computation and killing the thread. One way to do so is to have the `run` method loop until some condition is satisfied. Listing 5.8 contains a `run` method that tests to see if it is being executed by the proper thread or if the thread identifier, `thread`, has been set to null. Note that the thread identifier has been declared to be `volatile`. The `Volatile` keyword guarantees that any thread that reads a variable will see the most recently written value.<sup>2</sup>

Listing 5.8 A `Runnable` object with its own thread.

```

1 package org.opensourcephysics.manual.ch05;
2 public class RunComputation implements Runnable {
3   private volatile Thread thread;
4
5   public synchronized void startRunning() {
6     if(thread != null) {
7       return; // thread is already running
8     }
9     thread = new Thread(this);
10    thread.start(); // start the thread
11  }
12
13  public synchronized void stopRunning() {
```

<sup>1</sup>Ideally the animation thread's sleep time should be no more than the reciprocal of the monitor's video refresh rate. This is usually not attainable for all but the simplest simulations because a typical refresh rate is 72 Hz.

<sup>2</sup>Although the Java language guarantees that reading and writing a 32 bit variable is atomic, it does not guarantee that a value written by one thread will immediately be seen by another unless that variable is declared to be `volatile`. A thread is allowed to copy a variable from main memory to cache. Other threads will also copy the variable from main memory and will not see changes made to the first copy unless the variable is declared to be `volatile`. See *The Java Language Specification*, 2nd ed., by Gosling, Steele, and Bracha. Unless you are running on a multiprocessor Java VM, you are unlikely to observe this effect.

```

14     Thread tempThread = thread; // temporary reference
15     thread = null; // signal the thread to die
16     // return if thread already stopped; cannot join with
17     // current thread
18     if (tempThread==null || tempThread==Thread.currentThread()) {
19         return;
20     }
21     try {
22         tempThread.interrupt(); // get out of sleep state
23         tempThread.join(
24             1000); // wait up to one second for the
25         // thread to die
26     } catch(InterruptedException e) {}
27 }
28
29 public void run() {
30     while(thread==Thread.currentThread()) {
31         // add code for computation here
32         System.out.println("still running");
33         try {
34             Thread.sleep(100);
35         } catch(InterruptedException ie) {}
36     }
}

```

Any of the following actions causes the `while` loop in Listing 5.8 to terminate and the thread to die:

1. The thread identifier has been set to null. This is the correct way to signal the run method to stop, as is done in the `stopAnimation` method in our example.
2. A new thread has been created but the run method is executing the previous thread. Because only one calculation should be running at a time, we exit the loop. The old calculation thread will die and the new thread will soon start and take over the run method. This condition should not occur in our code because the thread identifier is checked in the `startAnimation` method.
3. The run method was invoked directly rather than by a calculation thread. This is usually an error and the run method should exit.

Note that because the `currentThread()` method is static, we invoke it directly without creating an instance variable.

There is an important point about stopping the thread that needs to be mentioned. Setting the calculation thread to null does not immediately stop the calculation. In fact, the thread is probably not even executing when the `stopAnimation` method is invoked because the Java virtual machine is busy process-

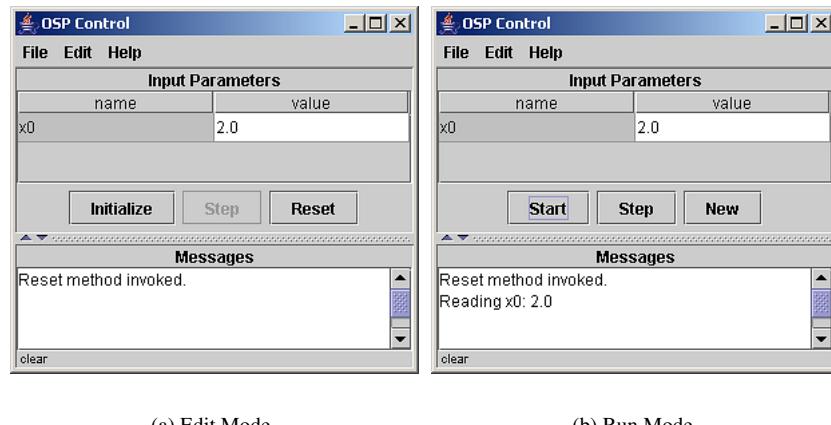
## 5.5 Animations

97

ing a user action. The program should, however, wait for the thread to die before returning from the `stopAnimation` method. This is accomplished using the `join` method. Notice also that the `stopAnimation` method invokes the `interrupt` method in order to awake the thread if it happens to be in a sleep state.

Finally, we mention that there are two library classes, `java.util.Timer` and `javax.swing.Timer`, that can save you from programming your own thread.<sup>3</sup> However, you still need to be careful when using timers because invoking a timer's `stop` method does not instantly halt the timer's action method.

## 5.5 ■ ANIMATIONS



**FIGURE 5.4** A simple control for use with an Animation.

The `AnimationControl` class shown in Figure 5.4 is designed to start and stop threads. It implements the control interface and is similar to the `CalculationControl` discussed in Section 5.3. This control has three buttons that invoke methods in a class that implements the `Animation` interface. The labels on the buttons change depending on the control's mode so that all five of the action methods shown in Listing 5.9 can be invoked. The sixth method, `setControl`, is called when the program is instantiated from within the `main` method so that the model knows which control is being used to store and retrieve parameter values. The model's `setControl` method usually invokes other methods within the model, such as `resetAnimation`, in order to initialize the model

<sup>3</sup>There are also thread libraries available on the Internet. See, for example, Doug Lea's `util.concurrent` package and the book *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed., by the same author.

and the user interface. If this initialization is not done, the control's parameter input table will be empty when it first appears onscreen.

Listing 5.9 The Animation interface.

```

1 public interface Animation extends Runnable {
2     public void startAnimation();
3     public void stopAnimation();
4     public void initializeAnimation();
5     public void resetAnimation();
6     public void stepAnimation();
7     // register the control with the model
8     public void setControl(Control control);
9 }
```

Starting an animation is a two-step process. When an animation control is created, it is ready to load new parameter values or edit existing parameter values. Clicking the control's Initialize button puts the control into run mode and invokes the animation's `initializeAnimation` method. The model uses this method to read parameter values from the control and send them to other objects as needed. Run mode is designed to start and stop threads and to single step the animation. The label on the Start button changes to Stop and the single Step button is disabled when an animation thread is running. In order to avoid synchronization problems, we do not allow users to edit parameter values while the program is running. Users can stop the animation, edit parameters, and press Start to continue. It is up to the programmer to read parameter values from the control when needed. It might, in fact, not be desirable to completely re-initialize the model when continuing an animation.

Custom buttons are disabled within an animation control when the animation thread is running. This paradigm may seem constraining at first, but it avoids synchronization problems. It insures that input parameters are clearly stated and that these parameters are not changed arbitrarily while the program is running.

In order to make it easy to create an animation, we have defined the `AbstractAnimation` class. This class implements the `Animation` interface and has one abstract method named `doStep`. The class contains an animation thread and this thread invokes the `doStep` method at regular intervals. The `doStep` method is also invoked from within the `stepAnimation` method. Examine the `AbstractAnimation` and note that we have insured that only one animation thread is running by stopping the current thread and waiting for a *join* to occur whenever the `stopAnimation` method is invoked.

Concrete animations define the `doStep` method to evolve the model. The test program shown in Listing 5.10 creates a “toy” animation by implementing this method.

Listing 5.10 An `AnimationControl` test program.

```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.controls.*;
```

## 5.5 Animations

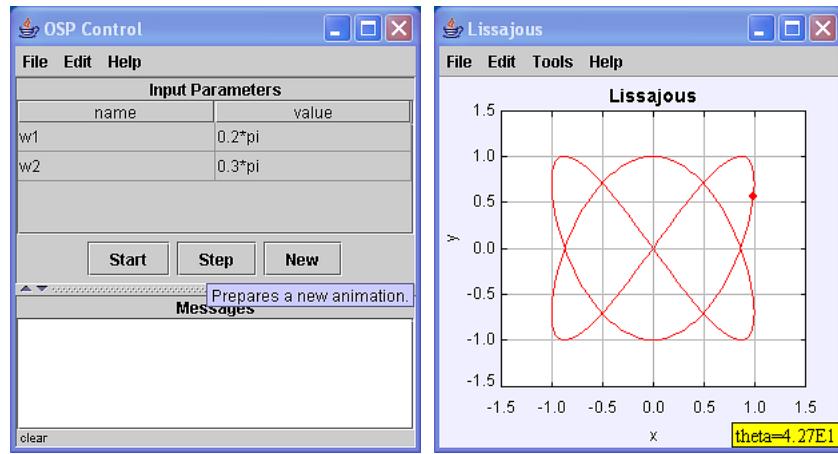
99

```

3  public class AnimationApp extends AbstractAnimation {
4      public void initializeAnimation() {
5          double x0 = control.getDouble("x0"); // read a parameter
6          value
7          control.println("initializeAnimation method invoked.");
8          control.println("reading x0: "+x0);
9      }
10
11     public void stopAnimation() {
12         control.println("stopAnimation method invoked");
13         super.stopAnimation();
14     }
15
16     public void stepAnimation() {
17         control.println("step method invoked");
18         super.stepAnimation();
19     }
20
21     public void startAnimation() {
22         control.println("startAnimation method invoked");
23         super.startAnimation();
24     }
25
26     public void resetAnimation() {
27         control.println("resetAnimation method invoked");
28         super.resetAnimation();
29         control.setValue("x0", 2.0); // initialize a parameter value
30     }
31
32     public void doStep() {
33         control.println("a step");
34     }
35
36     public static void main(String[] args) {
37         AnimationControl.createApp(new AnimationApp());
38     }
39 }
```

The `AbstractAnimation` class is very defensive. It always stops the animation before the program is reset, and it checks to determine if the animation is running before creating another thread. We could omit these checks if the control is designed to insure that initialization, starting, stopping, stepping, and resetting always occur in the proper sequence. For example, the `AnimationControl` is constructed so that it is not possible to invoke `initializeAnimation` or `resetAnimation` while in run mode. However, it is easy to create custom controls that do not enforce this paradigm as shown in Section 5.7.

## 5.6 ■ SIMULATIONS



(a) Simulation control.

(b) Lissajous figure.

**FIGURE 5.5** A graphical user interface controls the Lissajous simulation.

The animation framework provides the mechanism for stopping and starting a thread that performs a repetitive computation. The model is responsible for determining every aspect of the computation. The `AbstractSimulation` and `SimulationControl` classes in the `controls` package are based on the `AbstractAnimation` and `AnimationControl` classes, respectively, but include code that performs routine programming chores such as repainting windows after every computation.

A `Simulation` is a time dependent model of a physical system. The simulation begins with a set of initial conditions that determine the dynamical behavior of the model and generates data in the form of tables and plots as the model evolves. A `SimulationControl` is designed to control this type of a model. Frames are automatically repainted and data are automatically cleared when the Initialize and Reset buttons are clicked in a `SimulationControl`. Listing 5.11 demonstrates how the simulation framework is used by showing a two-dimensional trajectory. The program plots  $x(t)$  and  $y(t)$  using sinusoidal functions with angular frequencies  $\omega_1$  and  $\omega_2$ , respectively. The resulting patterns are known as Lissajous figures and are shown in Figure 5.5.

Listing 5.11 `LissajousApp` demonstrates how to create a simulation by drawing Lissajous figures.

## 5.6 Simulations

101

```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.*;
4 import org.opensourcephysics.frames.PlotFrame;
5
6 public class LissajousApp extends AbstractSimulation {
7     PlotFrame frame = new PlotFrame("x", "y", "Lissajous");
8     double
9         time = 0, dt = 0.1;
10    double w1, w2;
11    Circle circle = new Circle(0, 0, 3);
12
13    public void reset() {
14        time = 0.0;
15        frame.setPreferredMinMax(-1.5, 1.5, -1.5, 1.5);
16        frame.setConnected(true);
17        frame.setMarkerShape(0, Dataset.NOMARKER);
18        frame.addDrawable(circle);
19        control.setValue("omega 1", "0.2*pi");
20        control.setValue("omega 2", "0.3*pi");
21        initialize();
22    }
23
24    public void initialize() {
25        w1 = control.getDouble("omega 1");
26        w2 = control.getDouble("omega 2");
27        time = 0.0;
28        frame.append(0, 0, 0);
29        circle.setXY(0, 0);
30        frame.setMessage("theta="+decimalFormat.format(time), 1);
31    }
32
33    protected void doStep() {
34        time += dt;
35        double
36            x = Math.sin(w1*time), y = Math.sin(w2*time);
37        frame.append(0, x, y);
38        circle.setXY(x, y);
39        frame.setMessage("theta="+decimalFormat.format(time), 1);
40    }
41
42    public static void main(String[] args) {
43        SimulationControl.createApp(new LissajousApp());
44    }
45}

```

The LissajousApp program creates a `SimulationControl` in its main method and the program uses the control's `setValue` and `getValue` accessor methods to save and retrieve parameters from this control. Note how the an-

gular velocity is expressed as arithmetic expressions and how these strings are automatically converted to numbers by the `getDouble` method.

The `AbstractSimulation` automatically performs a number of common programming chores. It makes all `DrawingFrames` visible when the Initialize button is clicked. It repaints frames whose `animated` property is true after every call to `doStep`.

```
1 frame.setAnimated(true); // true by default within frames
  package
```

Simulations must explicitly invoke a drawing method such as `repaint` or `render` if a frame is not animated as described in Chapter 7.

Some frames such as `PlotFrame` store data; these data are usually cleared when a simulation is reset or initialized. The `SimulationControl` clears a frame's data when the initialize button is pressed if the `autoclear` property is true.

```
1 frame.setAutoClear(true); // true by default within frames
  package
```

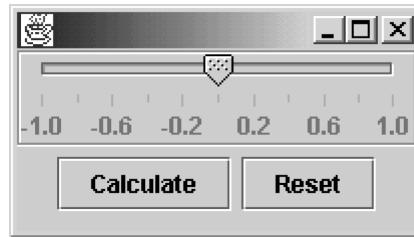
Although we usually clear data when initializing a simulation, we might set a plot frame's `autoclear` property to false if we wish to compare trajectories with different parameters. Data are always cleared when the Reset button is pressed because this action recreates the program's initial state.

Deciding what should be cleared from a frame depends on the situation. A projectile simulation might use a circle to show the particle's *x-y* position, and it would be inconvenient if this circle were automatically removed during initialization. The particle is always present. A molecular dynamics simulation, on the other hand, may create *N* circles and add them to a frame after reading the number of particles. Existing circles should be removed when this program is initialized. In general, frames will automatically clear data stored within the frame's drawable objects, but will not remove the drawable objects themselves. Drawable objects can, however, be removed from a frame using any of the following methods.

```
1 Circle circle = new Circle(); // creates a drawable
  object
2 frame.addDrawable(circle); // adds circle to frame
3 // examples of how to remove drawables
4 frame.clearDrawables(); // removes all
  drawable objects
5 frame.removeDrawable(circle); // removes a single
  object
6 frame.removeObjectsOfClass(Circle.class); // removes all
  instances of Circle
```

## 5.7 Ejs Controls

103

FIGURE 5.6 A custom *Ejs* control that can control any object.

## 5.7 ■ EJS CONTROLS

*Easy Java Simulations* (EJS), is a high-level modelling tool written by Francisco Esquembre at the University of Murcia, Spain, that builds Java applications and applets using a drag and drop graphical user interface. We have include some EJS components in the core Open Source Physics library in the `ejs` package in order to enable programmers to quickly build custom user interfaces. This section provides an overview of the `ejs` package components. A more complete description is given in the EJS technical manual available on the OSP website. A description of the entire Easy Java Simulations project, including the complete modeling program, is available at:

<http://fem.um.es/Ejs/>

The `ejs` package builds controls without any preconditions as to the type of object that will be controlled. That is, a controlled class need not implement any given interface. For example, the following code fragment creates the user interface shown in Figure 5.6 consisting of a slider and two buttons that invoke methods in `anyObject`.

Listing 5.12 An EJS control containing a slider and two buttons.

```

1 GroupControl control = new GroupControl(anyObject);
2 control.add("Frame",
3   "name=controlFrame;exit=true; size=200,90;layout=vbox");
4 control.add("Slider", "parent=controlFrame;minimum=-1; maximum=1;
5   ticks=11;ticksFormat=0.0; variable=x0,
6   dragaction=sliderMoved");
7 control.add("Panel", "name=controlPanel; parent=controlFrame;
8   size=300,300; position=south; layout=flow");
9 control.add("Button",
10  "parent=controlPanel;text=Calculate;action=calculate()");
11 control.add("Button",
12  "parent=controlPanel;text=Reset;action=resetCalculation()");
13 control.update();

```

The code starts by instantiating a group control named `control` and passing it a reference to the object that will be controlled, `anyObject`. This controlled object will be referred to as the *target*. The target receives actions, such as button clicks or slider moves, from the control. The second statement begins the process of creating the control's user interface by instantiating a `ControlFrame`. The next statement adds a slider whose action invokes the `sliderMoved` method in the target. The remaining statements repeat this process by adding another panel and two buttons to the frame. The actions of the slider and the buttons are public methods within the target.

EJS supports a wide variety of user interface components including labels, radio buttons, check boxes, and one-line text fields. These components are created in a script-like style by invoking the group control's `add` method with the following signature:

```
1 GroupControl add(String type, String property_list);
```

The `add` method's first parameter specifies the type of object that is being created. The second parameter is a semicolon delimited list of properties. Properties depend on the type of object being created but typically include the following:

- **name** The name of the instance. This name can later be used to access the object.
- **parent** The name of the object's container. Typical containers are panels and windows.
- **action** The name of a method in the model that will be invoked when the user clicks, slides, or otherwise activates the control.

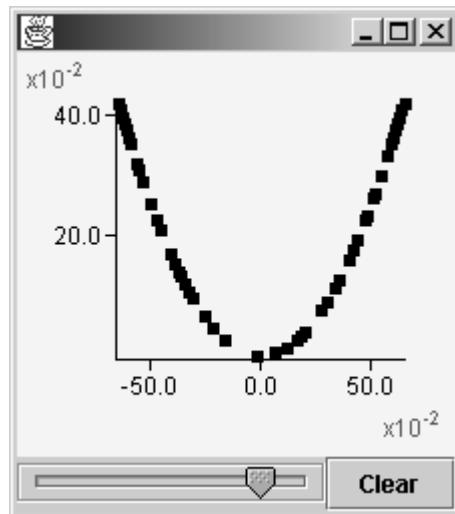
Listing 5.13 shows a “toy” program that creates the user interface shown in Figure 5.7 by combining two EJS controls with a plotting panel. Moving the slider varies the control's  $x$  parameter and plots the function  $y = x^2$ . Pressing the button clears the data.

Listing 5.13 A demonstration of how to use an EJS control with a drawing panel.

```
1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.ejs.control.GroupControl;
4
5 public class EJSSliderApp {
6     DrawingPanel plottingPanel = new PlottingPanel("x", "y",
7             "EJS Controls Test");
8     GroupControl control;
9     Dataset dataset = new Dataset();
10
11    public EJSSliderApp() {
12        control = new GroupControl(this);
13        control.add(
```

## 5.7 Ejs Controls

105



**FIGURE 5.7** A user interface containing a plotting panel and two *EJS* controls.

```

14     "Frame",
15     "name=plottingFrame;title=EJS Example;exit=true;
16     size=300,300");
17 control.addObject(plottingPanel, "Panel",
18     "name=plottingPanel; parent=plottingFrame;
19     position=center");
20 control.add(
21     "Panel",
22     "name=controlPanel; parent=plottingFrame;
23     position=south; layout=hbox");
24 control.add(
25     "Slider",
26     "parent=controlPanel; minimum=-1; maximum=1;variable=x;
27     dragaction=sliderMoved()");
28 control.add("Button",
29     "parent=controlPanel; text=Clear;
30     action=clearPlot()");
31 plottingPanel.addDrawable(dataset);
32 control.update();
33 }
34
35 public void clearPlot() {
36     dataset.clear();
37     plottingPanel.repaint();
38 }
39
40 public void sliderMoved() {
41 }
```

```

36     double x = control.getDouble("x");
37     dataset.append(x, x*x);
38     plottingPanel.repaint();
39 }
40
41 static public void main(String[] args) {
42     new EJSSliderApp();
43 }
44 }
```

The `addObject` method is used to add an existing object to an EJS control, and the `add` method is used to instantiate and add an object to an EJS control. The control's `update` method guarantees that control elements display current values.

Although `GroupControl` does not implement the OSP Control interface, `EjsControl` and `EjsControlFrame` do. `EjsControlFrame` extends `EjsControl`. It contains a `JFrame` and implements the `RootPaneContainer` interface in order to function as a container for user interface elements.

Unlike our predefined controls, controls that inherit from `EjsControlFrame` can have an arbitrary number of user interface components and actions. In Listing 5.14 we build a control with a slider to replace a `CalculationControl` with a single parameter. Figure 5.6 shows a screen shot of the resulting user interface.

**Listing 5.14** An *Ejs* control can replace a calculation control's parameter input table.

```

1 public class CustomCalculationFrame extends EjsControlFrame {
2     public CustomCalculationFrame(Object target) {
3         super(target,
4             "name=controlFrame;exit=true; size=200,90;layout=vbox");
5         // creates the user interface for the control
6         add("Slider", "parent=controlFrame;minimum=-1; maximum=1;
7             ticks=11; ticksFormat=0.0; variable=x0");
8         add("Panel", "name=controlPanel; parent=controlFrame;
9             size=300,300; position=south; layout=flow");
10        add("Button",
11            "parent=controlPanel;text=Calculate;action=calculate");
12        add("Button",
13            "parent=controlPanel;text=Reset;action=resetCalculation");
14        update();
15    }
16 }
```

The new custom calculation control is a direct replacement for the control in the calculation test program shown in Listing 5.4. The `main` method must, of course, instantiate the custom control.

```

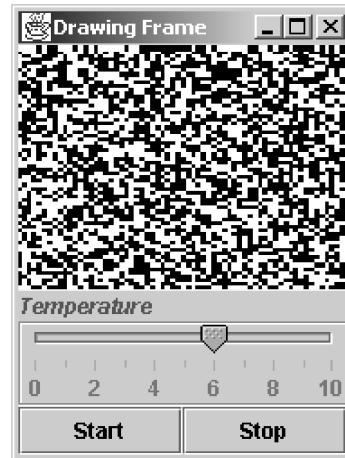
1 public static void main (String[] args) {
2     Calculation model = new CalculationTestApp();
3     Control control = new CustomCalculationControl(model);
```

## 5.7 Ejs Controls

107

```

4 model.setControl(control);
5 }
```



**FIGURE 5.8** A custom *EJS* control can replace an *AnimationControl*.

Simulations often benefit from the visual simplicity of a custom control that starts and stops a thread. Consider the Ising model shown in Figure 5.8. Our implementation of this model contains an animation thread and a temperature parameter. In Listing 5.15 we create a custom control with two buttons and a slider to replace the original *AnimationControl*.

**Listing 5.15** A custom control for the Ising model.

```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.ejs.control.EjsControlFrame;
3
4 public class IsingControl extends EjsControlFrame {
5     public IsingControl(IsingApp model) {
6         super(model,
7             "name=controlFrame;title=Control
8                 Frame;size=400,400;layout=border;exit=true");
9         addObject(
10             model.drawingPanel, "Panel",
11             "name=drawingPanel; parent=controlFrame; layout=border;
12                 position=center");
13         add("Panel",
14             "name=controlPanel; parent=controlFrame; layout=border;
15                 position=south");
16         add("Panel",
```

```

14         "name=sliderPanel; position=north; parent=controlPanel;
15             layout=border");
16     add("Slider",
17         "position=center; parent=sliderPanel; variable=T;
18             minimum=0; maximum=5; ticks=11; ticksFormat=0;
19             action=sliderMoved()");
20     add("Label",
21         "position=north; parent=sliderPanel; text=Temperature;
22             font=italic,12; foreground=blue;
23             horizontalAlignment=center");
24     add("Panel",
25         "name=buttonPanel; position=south; parent=controlPanel;
26             layout=flow");
27     add("Button",
28         "parent=buttonPanel; text=Run; action=startAnimation()");
29     add("Button",
30         "parent=buttonPanel; text=Stop; action=stopAnimation()");
31 }
32 }
```

EJS controls can either instantiate standard Swing user interface components using the `add` method or they can accept already existing objects using the `add-Object` method. In our example, the model already instantiated a `DrawingPanel`, and we want to add this panel to the control. EJS sets the existing panel's properties and adds it to the parent because the `DrawingPanel` is a subclass of `JPanel`.

When the `IsingControl` slider's action method adjusts the temperature, it is the model's responsibility to insure that the temperature can safely be changed by the slider even if the animation thread is running. We do so by temporarily stopping the animation thread. The Ising model is shown in Listing 5.16. The physics-related code has been removed from the listing for clarity. The Ising control is a separate class, but it is short and could have been defined as an inner class within this model.

A final word about simplicity of design. `CalculationControl` and `AnimationControl` are designed for input/output without worrying about details such as layout managers and appearance. This approach is ideal for testing code and for teaching students how to program. *EJS* provides the opportunity to create very attractive user interfaces without changing the computational model. We recommend that users develop their models first and then — if a better interface is desired — build an *EJS* control to fit the model.

Listing 5.16 An outline of the Ising program.

```

1 package org.opensourcephysics.manual.ch05;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.DrawingPanel;
4 import org.opensourcephysics.display2d.BinaryLattice;
5
6 public class IsingApp extends AbstractAnimation {
```

## 5.8 Programs

109

```

7   int size = 32;
8   double temperature = 2;
9   DrawingPanel drawingPanel = new DrawingPanel();
10  BinaryLattice lattice = new BinaryLattice(size, size);
11
12  public IsingApp() {
13      drawingPanel.addDrawable(lattice);
14  }
15
16  public void sliderMoved() {
17      boolean isRunning = animationThread!=null;
18      stopAnimation();
19      temperature = control.getDouble("T");
20      // physics has been removed for clarity
21      if(isRunning) {
22          startAnimation();
23      }
24  }
25
26  protected void doStep() {
27      // physics has been removed for clarity
28      lattice.randomize();
29      drawingPanel.repaint();
30  }
31
32  public static void main(String[] args) {
33      IsingApp model = new IsingApp();
34      Control control = new IsingControl(model);
35      model.setControl(control);
36  }
37 }
```

**5.8 ■ PROGRAMS**

The following examples are in the `org.opensourcephysics.manual.ch05` package.

**AnimationApp**

`AnimationApp` test the `Animation` interface. Edit this example to build new animations.

**AnimationFrameApp**

`AnimationFrameApp` creates an animation by extending the `AbstractAnimation` class. Edit this example to build new animations.

**CalculationApp**

CalculationApp tests the Calculation interface. Edit this example to build new animations.

**CustomButtonApp**

CustomButtonApp extends AbstractCalculation and creates a CalculationControl with a custom button.

**EJSSliderApp**

EJSSliderApp creates an EJS control containing a button and a slider.

**IsingApp**

IsingApp builds a custom graphical user interface for the Ising model using EJS.

**LissajousApp**

LissajousApp demonstrates how to create a Simulation by animating Lissajous figures using a PlotFrame.

**MainThreadApp**

MainThreadApp pauses the main thread using the sleep method to produce a simple animation.

**OSPControlApp**

OSPControlApp creates an OSPControl, adds a button to the control, and reads parameters from the control.

**RotationApp**

RotationApp creates two rotating rectangle animations. This program is described in Section 5.4.

**SetValueApp**

SetValueApp tests the setValue method and the setAdjustableValue method in SimulationControl.

**ShapeAnimationApp**

ShapeAnimationApp creates a rotating rectangle and implements an AbstractAnimation.

**SimulationApp**

SimulationApp creates a rotating square by extending an AbstractSimulation and implementing the doStep method. Because the frame is set to be animated,

## 5.8 Programs

**111**

the render method is invoked automatically after every animation step.

## CHAPTER

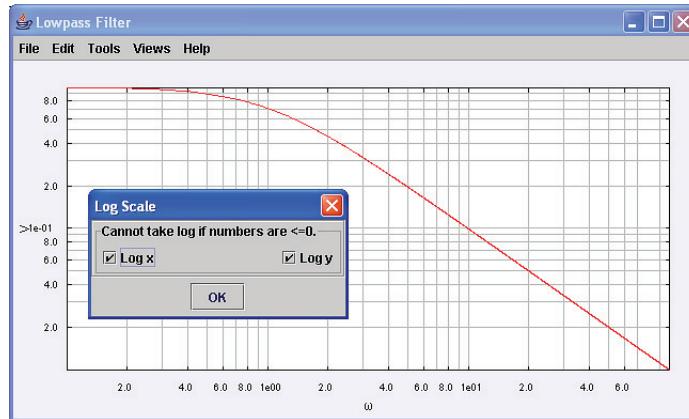
## 6

## Plotting

©2005 by Wolfgang Christian

The display package defines components that can be used to create a wide variety of x-y scatter plots including polar, semi-log, and log-log plots.

### 6.1 ■ OVERVIEW



**FIGURE 6.1** A lowpass RC filter frequency response plotted using a `PlotFrame` in the `frames` package. The log-scale dialog is available in the `Views` menu.

The easiest way to create an x-y scatter plot is to use the `PlotFrame` (see Figure 6.1) in the `frames` package. This frame contains all the necessary components to plot multiple datasets, change scale, display log-log and semilog plots, and show data in tabular form. It is, however, impossible to meet every data visualization need without duplicating the functionality of specialized programs such as *GNUPlot* and the `Export` item under the file menu is available if this high level plotting program is needed. This chapter describes how the `PlotFrame` and `PlottingPanel` classes are used to create custom plots.

## 6.2 ■ DATASET

A plot is created by storing data points in one or more *datasets*. A `Dataset` object is a `Drawable` object that stores and renders  $(x, y)$  data points. In order to construct a simple graph, we instantiate a dataset, add it a drawing panel, and append points to the dataset. Typical code is shown in Listing 6.1.

Listing 6.1 A drawing panel containing a dataset and two axes components.

```

1 package org.opensourcephysics.manual.ch06;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.display.axes.XAxis;
4 import org.opensourcephysics.display.axes.YAxis;
5 import javax.swing.JFrame;
6
7 public class DatasetApp {
8     public static void main(String[] args) {
9         // create a drawing frame and a drawing panel
10        DrawingPanel panel = new DrawingPanel();
11        DrawingFrame frame = new DrawingFrame(panel);
12        panel.setSquareAspect(false);
13        panel.setAutoscaleX(true);
14        panel.limitAutoscaleY(-5, 15);
15        // create data and append it to a dataset
16        double[] x = {-2, -1, 0, 1, 2};
17        double[] y = {-2, 2, 6, 10, 14};
18        Dataset dataset = new Dataset();
19        dataset.setConnected(true);
20        dataset.setSorted(true); // sort points as needed
21        dataset.append(x, y);
22        dataset.append(5, 11); // a single point
23        dataset.append(3.5, 2.5); // this point will be sorted
24        // add the drawable objects to the drawing panel
25        panel.addDrawable(dataset);
26        XAxis xaxis = new XAxis("x");
27        panel.addDrawable(xaxis);
28        YAxis yaxis = new YAxis("y");
29        panel.addDrawable(yaxis);
30        panel.repaint();
31        frame.setVisible(true);
32        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33    }
34}

```

Single points and arrays of points can be added to a dataset using the `append` method. However, views (such as drawing panels with datasets) are not refreshed until the view is repainted. This allows a program to append multiple points to a graph without unnecessary drawing.

Data points with error estimates can also be appended to datasets using the

following signature:

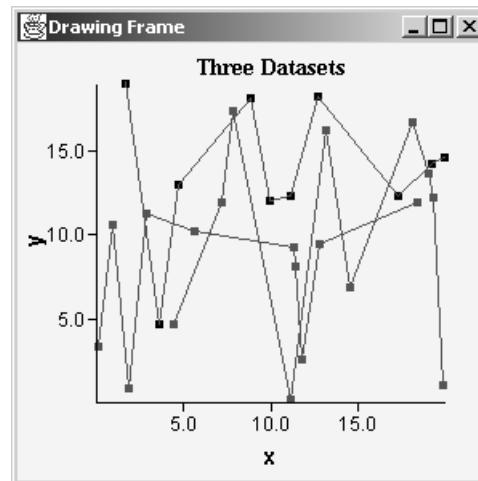
```
dataset.append(x, y, err_x, err_y); //parameters can be double  
or double[]
```

Error estimates are optional and can be included with only a few select points to limit the number of error bars being drawn.

Table 6.1 lists common Dataset methods. Consult the source-code for a complete listing of dataset methods and more extensive documentation.

Because a dataset has many scaling and marker options, drawing large numbers of points can be slow. The `setMaximumPoints` method limits the number of points that will be stored. The default maximum number is 16K. Points at the beginning of the dataset are dropped if this number is exceeded. There are other objects that can draw many more points because they create simpler representations for each point. The `Trail` object in the display package displays a path of connected points without markers. The `DataRaster` in the `display2d` package creates an image by coloring one pixel per point. (See Section 7.6.)

### 6.3 ■ DATASET MANAGER



**FIGURE 6.2** A plotting panel showing three datasets. (See Listing 6.2.)

A dataset manager is a composite object that performs many of the housekeeping chores associated with multiple datasets. Datasets are accessed using an integer index and the required dataset is created the first time the index is used. It is possible to set these (and other) options globally as well as for individual

## 6.3 Dataset Manager

115

**TABLE 6.1** Common dataset methods.

<b>org.opensourcephysics.display.Dataset methods</b>	
append	Adds data to the data set as $x, y$ coordinate pairs. The data can be either doubles or arrays of doubles. Note that data points having the non-numeric values of Double.NaN or infinity are discarded.
clear	Removes all data.
isMeasured	Gets the valid-measure flag. This flag is true if the data set contains at least one valid data point.
getXMax	Gets the maximum x value in the dataset. Note that this value is guaranteed to be correct only if the isMeasured flag is true.
getXMin	Similar to getXMax.
getYMax	Similar to getXMax.
getYMin	Similar to getXMax.
toString	Creates a string representation of the data with every point on a new line. X and y values are separated by a tab.
setConnected	Connects the data points with straight lines. Default is false.
setLineColor	Sets the color of the lines connecting the points. Ignored if points are not connected. Default is black.
setMarkerColor	Sets the color of the data markers. Ignored if the data marker is set to NO_MARKER. Default is black.
setMarkerShape	Sets the shape of the data markers. Valid shapes are NO_MARKER, CIRCLE, SQUARE, AREA, PIXEL, BAR, and POST . The default is SQUARE.
setMaximumPoints	Sets the allowed number of points. The default is 16K.
setSorted	Sort the dataset by increasing x value. Default is false.

datasets. For example, the marker type, color, and connectedness of data points can be set using the following code fragment.

```

1 datasetManager = new DatasetManager();
2 datasetManager.setConnected(true);      // connects points in all
                                         datasets
3 datasetManager.setLineColor(3, Color.blue);        // dataset 3
                                         line color

```

```
4 datasetManager.setMarkerShape(3, Dataset.CIRCLE); // dataset 3
marker
```

Listing 6.2 uses a dataset manager to create three sets of random data. The first dataset has an index of 0. The manager assigns different colors and markers to each dataset.

**LISTING 6.2** A dataset manager containing three datasets.

```
1 package org.opensourcephysics.manual.ch06;
2 import org.opensourcephysics.display.*;
3 import javax.swing.JFrame;
4
5 public class DatasetManagerApp {
6     public static void main(String[] args) {
7         int numberOfDatasets = 3;
8         int pointsPerDataset = 10;
9         DatasetManager datasetManager = new DatasetManager();
10        datasetManager.setConnected(true);
11        datasetManager.setSorted(true);
12        for(int i = 0;i<numberOfDatasets;i++) {
13            for(int j = 0;j<pointsPerDataset;j++) {
14                datasetManager.append(i, Math.random()*20,
15                                      Math.random()*20);
16            }
17        }
18        PlottingPanel panel = new PlottingPanel("x", "y",
19                                         "Three Datasets");
20        panel.setAutoscaleX(true);
21        panel.setAutoscaleY(true);
22        panel.addDrawable(datasetManager);
23        DrawingFrame frame = new DrawingFrame(panel);
24        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25        frame.setVisible(true);
26    }
}
```

Many of the methods in the `Dataset` class have `DatasetManager` counterparts. Methods without an index set the default and perform the method on all existing datasets. Table 6.2 shows some common methods implemented in the manager.

## 6.4 ■ MARKERS

The `PlotFrame` class creates datasets with point *markers* having different drawing properties. The marker shape, size, and color as well as a boolean that deter-

## 6.4 Markers

117

**TABLE 6.2** Typical dataset manager methods.**`org.opensourcephysics.display.DatasetManager` methods**

<code>append</code>	Appends data to the i-th dataset.
<code>clear</code>	Removes data from all datasets or from the i-th dataset.
<code>isMeasured</code>	Returns true if at least one dataset in the manager has a valid point.
<code>getXMin</code>	Gets the minimum x value in all datasets.
<code>getXMax</code>	Gets the maximum x value in all datasets.
<code>getYMin</code>	Gets the minimum y value in all datasets.
<code>getYMax</code>	Gets the maximum y value in all datasets.

mines if markers should be connected with line segments can be set using dataset accessor methods. Listing 6.3 uses two datasets with different drawing options to simulate a Gaussian spectral line with random noise. This example use as a `PlotFrame` object rather than components from the `display` package to emphasize that the frame's API is based on methods that are forwarded to low-level components.

Listing 6.3 Program `GaussianPlotApp` displays two datasets.

```

1 package org.opensourcephysics.manual.ch06;
2 import org.opensourcephysics.display.Dataset;
3 import org.opensourcephysics.frames.PlotFrame;
4 import javax.swing.JFrame;
5
6 public class GaussianPlotApp {
7     public static void main(String[] args) {
8         PlotFrame frame = new PlotFrame("$\Delta f", "intensity",
9                                         "Gaussian Spectral Line");
10        frame.setConnected(0, true);
11        frame.setMarkerShape(0, Dataset.NO_MARKER);
12        for(double f = -10; f < 10; f += 0.2) {
13            double y = Math.exp(-f*f/4);
14            frame.append(0, f, y); // datum
15            frame.append(1, f, y+0.1*Math.random()); // datum + noise
16        }
17        frame.setVisible(true);
18        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19    }
20}

```

Marker shapes are changed by passing a named integer constant, such as `CIRCLE` or `SQUARE`, to the `setMarkerShape` method. The marker size (in pixels) can also be adjusted as shown in the following code fragment.

```

1 // set style for dataset 1
2 frame.setMarkerShape(1, Dataset.CIRCLE);
3 frame.setMarkerSize(1,2); //half width of marker is 2 pixels

```

**TABLE 6.3** The `Dataset` class uses named integer constants to tag common marker shape.

<b>Dataset marker shapes</b>	
AREA	Fills the area between the connected points and the x axis with a solid color.
BAR	A rectangle with its bottom edge on the x axis and its top edge centered on the data point.
CIRCLE	A circle centered on the data point.
NO_MARKER	Disables marker drawing. Line segments connecting data point centers will be drawn if the dataset is connected.
PIXEL	A marker of exactly one pixel.
POST	A square centered on the data point and a line connecting the data point to the x axis.
SQUARE	A square centered on the data point.

Marker shapes defined in the `Dataset` class are described in Table 6.3.

Markers can be used to create a variety of graphs as shown in Figure 6.3. Color is set using the `setMarkerColor` method and this method has a number of signatures to produce different effects. Calling this method with a single color produces a shape with a single color. It is usually more attractive to use multiple colors for the marker's interior, edge, and error bars. The color's *alpha* value can be used to define a fill color that is slightly transparent so that overlapping data points can be seen. For example, the following code creates a dataset with a pale-red semi-transparent interior, red edge, and black error bars.

```

1 Dataset dataset = new Dataset ();
2 dataset.setConnected (false); // do not connect the points with
   lines
3 dataset.setMarkerShape (Dataset.SQUARE);
4 dataset.setMarkerColor (new
   Color(255,128,128), Color.red, Color.black);

```

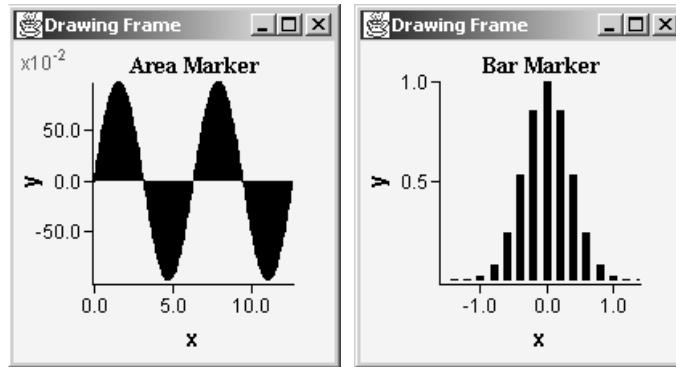
The BAR marker renders each data point as a rectangle with a base centered at  $x$  and a height of  $y$ . This marker is useful for histogram-like plots. The POST marker is similar except that a small square is drawn centered on the data point together with a line connecting the marker to the x-axis. The line is drawn using the marker's edge color.

The AREA marker fills the area between the x-axis and the connected data points with a solid color.

The PIXEL marker draws a single pixel at the data point. It is useful for Poincaré sections. (The `DataRaster` class described in Section 7.6 is a specialized drawing component that is designed to render very large datasets quickly by coloring a single pixel for each datum.)

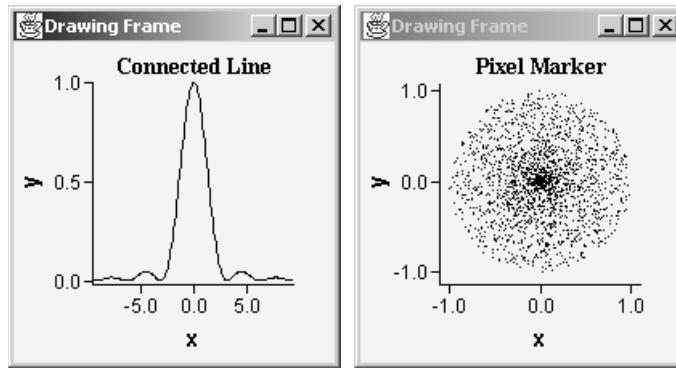
## 6.4 Markers

119



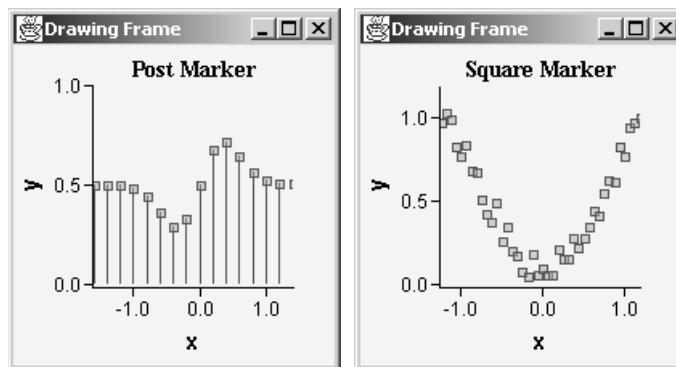
(a) Area marker.

(b) Bar marker.



(c) Connected line.

(d) Pixel marker.



(e) Post marker.

(f) Square marker.

**FIGURE 6.3** Dataset markers can be used to create a variety of graphs.

Custom point markers can be created using the Shape interface in the Java 2D API. The GeneralPath class in Sun's geometry package defines a shape using a sequence of `moveTo` and `lineTo` methods to create an outline. The following code fragment creates a custom butterfly marker for a dataset and then sets the marker's interior color and edge color to red and black, respectively.

```

1 // import java.awt.geom.*;
2 GeneralPath marker = new GeneralPath();
3 marker.moveTo(3, 3);
4 marker.lineTo(3, -3);
5 marker.lineTo(-3, 3);
6 marker.lineTo(-3, -3);
7 marker.closePath();
8 frame.setCustomMarker(1, marker);
9 frame.setMarkerColor(1, Color.RED, Color.BLACK);
```

## 6.5 ■ AXES

Axes are special drawable components that implement the `DrawableAxes` interface and are drawn within a plotting panel before all other objects. (See Table 6.4.) An axes object always exists although it can be hidden by setting its visible property to false.

**TABLE 6.4** Concrete implementations of the `DrawableAxes` interface.

**org.opensourcephysics.display.axes package classes**

<code>CartesianType1</code>	Cartesian axes that are drawn outside the plotting region.
<code>CartesianType2</code>	Alternate cartesian axes that are drawn outside the plotting region.
<code>CartesianType3</code>	Cartesian axes that are drawn within the plotting region. Each axis is dragable.
<code>CustomAxes</code>	A basic class with an interior and a title. Drawable objects can be added to this class to produce custom axes.
<code>PolarType1</code>	Polar coordinates that remain centered in the drawing panel. The panel's minimum and maximum values are set equal to the largest preferred value.
<code>PolarType2</code>	Polar coordinates that respect the panel's preferred values and can therefore be off-center.

The plotting panel's default axes component, `AxesType1`, is based on open source code released by the University of California. `AxesType2` is similar but

## 6.7 Dataset Tables

121

is based on open source code released by Leigh Bookshaw. `AxesType3` is also based on Bookshaw's code but each axis is drawn within the plotting region and is draggable. All axes have similar APIs. When an axes component is constructed, a plotting panel is passed to the constructor. This constructor sets the panel's gutters and registers the new axes with the panel. The following code fragment is taken from the `PolarPlottingApp` program in the manual's chapter 2 package.

```

1 PlottingPanel plottingPanel=new PlottingPanel(null, null, null);
2 PolarAxes axes= new PolarAxes(plottingPanel);
3 axes.setRadialGrid(1);
4 axes.setThetaGrid(Math.PI/8);
5 plottingPanel.setTitle("PolarPlot");
6 plottingPanel.repaint();

```

Because axes may correspond to variables other than x and y, we have implemented mechanisms that allow us to disable the coordinate readout or to change the labels in the coordinate readout. The `setShowCoordinates` method allows us to disable the panel's coordinate display in the left-hand corner yellow message box when the mouse is pressed. The `CoordinateStringBuilder` class is responsible for assembling the string when the mouse is pressed. The labels that are prepended to the coordinate values can be set using the `setCoordinateLabels` method.

```

1 plottingPanel.setShowCoordinates(true); // default is true
2 plottingPanel.getCoordinateStringBuilder().setCoordinateLabels("t", "amp");

```

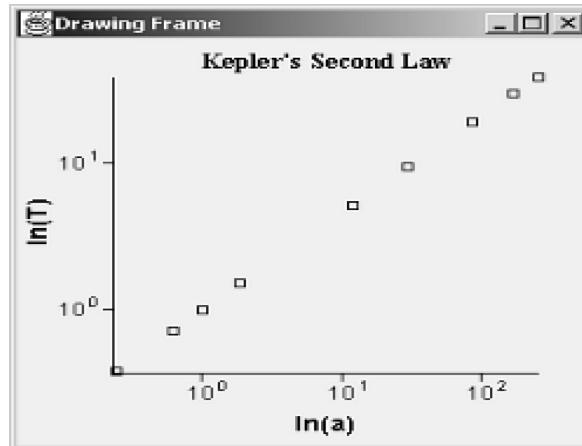
## 6.6 ■ LOG SCALE

The `PlottingPanel` contains the necessary axes and titles to produce linear, log-log, and semilog graphs. For example, Listing 6.4 shows that Kepler's second law is plausible by producing a log-log plot of the semi-major axis of the planets versus the orbital period. The data arrays, `a` and `T`, contain the semimajor axis of the planets in astronomical units and the orbital period in years, respectively. Note that the plot automatically adjusts itself to fit the data because the `autoscale` option is set to true for both x and the y axis.

Setting the log scale option causes the drawing panel to transform the data as it is being plotted and causes the axis to change how labels are rendered. If the drawing panel has minimum and maximum values of 1 and 3, respectively, then the labels will be drawn as  $10^1$ ,  $10^2$ , and  $10^3$ .

## 6.7 ■ DATASET TABLES

Because `Dataset` and `DatasetManager` implement the `TableModel` interface, they can display themselves in a data table. (Section 3.8 describes how to cre-



**FIGURE 6.4** Kepler's second law demonstration. A plot of the semi-major axis of the planets in AU versus the log of the orbital period in years produces a straight line with a slope of 1.5.

ate tables without datasets.) Dataset tables are created by instantiating a `DataTable` and a `DataTableFrame` and then adding a dataset manager to the table. The paradigm is similar to that for adding datasets to a drawing panel and is shown in Listing 6.5. Note that the code invokes `refreshTable`. This method is equivalent to calling a drawing panel's `repaint` method insofar as it forces the data table to repaint itself.

Listing 6.5 A dataset manager can be used to create a table as shown in Figure 6.5.

```

1 package org.opensourcephysics.manual.ch06;
2 import org.opensourcephysics.display.DataTable;
3 import org.opensourcephysics.display.DataTableFrame;
4 import org.opensourcephysics.display.DatasetManager;
5 import javax.swing.JFrame;
6
7 public class DataTableApp {
8     public static void main(String[] args) {
9         DatasetManager datasets = new DatasetManager();
10        datasets.setXPointsLinked(true); // x points only for 0th
11        dataset
12        double[] xpoints = {1.45, 3.99, 5, 7};
13        double[] ypoints = {2, 4, 6, 8};
14        datasets.append(0, xpoints, ypoints);
15        double[] ypoints2 = {10.0, 12, 14, 45.2};
16        datasets.append(1, xpoints, ypoints2); // same x values
17        datasets.setXYColumnNames(0, "x1", "y1");
18        datasets.setXYColumnNames(1, "x2", "y2");

```

## 6.7 Dataset Tables

123

**LISTING 6.4** Kepler's second law plot.

```

1 package org.opensourcephysics.manual.ch06;
2 import org.opensourcephysics.display.*;
3
4 public class KeplerPlotApp {
5     public static void main(String[] args) {
6         PlottingPanel plotPanel = new PlottingPanel("ln(a)", "ln(T)",
7                                         "Kepler's Second Law");
8         // PlottingPanel plotPanel=
9         // PlottingPanel.createType2("ln(a)", "ln(T)", "Kepler's
10        // Second Law");
11         plotPanel.setLogScale(true, true);
12         DrawingFrame drawingFrame = new DrawingFrame(plotPanel);
13         Dataset dataset = new Dataset();
14         dataset.setConnected(false);
15         double[] period = {
16             0.241, 0.615, 1.0, 1.88, 11.86, 29.50, 84.0, 165, 248
17         };
18         double[] a = {
19             0.387, 0.723, 1.0, 1.523, 5.202, 9.539, 19.18, 30.06, 39.44
20         };
21         dataset.append(a, period);
22         plotPanel.addDrawable(dataset);
23         drawingFrame.setDefaultCloseOperation(
24             javax.swing.JFrame.EXIT_ON_CLOSE);
25         drawingFrame.setVisible(true);
26         // create a data table
27         DataTable dataTable = new DataTable();
28         DataTableFrame tableFrame = new DataTableFrame("Orbital
29             Data",
30             dataTable);
31         dataset.setXYColumnNames("T (years)", "a (AU)");
32         dataTable.addRowNumberVisible(true);
33         dataTable.refreshTable();
34         tableFrame.setVisible(true);
35     }
36 }
```

```

18     DataTable dataTable = new DataTable();
19     DataTableFrame frame = new DataTableFrame("Data Table",
20         dataTable);
21     dataTable.add(datasets);
22     datasets.setStride(1); // the default
23     dataTable.setRowNumberVisible(true);
24     dataTable.refreshTable();
```

row	x1	y1	y2
0	1.45	2	10
1	3.99	4	12
2	5	6	14
3	7	8	45.2
**FIGURE 6.5** Data within a dataset manager displayed in a table.

```

24 frame . setDefaultCloseOperation ( JFrame . EXIT_ON_CLOSE ) ;
25 frame . setVisible ( true ) ;
26 }
27 }
```

Each dataset produces two columns, one for the x value and another for the y values. These columns are labeled using the `setXYColumnNames` method.

```
datasets . setXYColumnNames ( 0 , " x1 " , " y1 " );
```

It is often the case that datasets have common x values as in the example above. It is therefore convenient to suppress x columns except those for the first dataset. This is done by setting the `points-linked` property.

```
datasetManager . setXPointsLinked ( true ); // X only for 0-th dataset
```

It is also possible to show and hide individual columns and to skip rows as shown in Table 6.5. Consult the data table source-code documentation for information about these data table methods.

We now show a longer example that uses multiple datasets and a table by solving the following differential equation:

$$dy/dt = -y. \quad (6.1)$$

The analytic solution to this equation is well known and we want to compare this solution to a numerical approximation that was generated using the what is known as Euler's algorithm. (See Chapter 9.) We will plot the analytic and numerical solutions and we will show these values in a table with a column whose entries are their difference. Figure 6.6 shows the resulting table and graph. Managing these data is easy using a dataset manager as shown in Listing 6.6. Note that all datasets are displayed within the table whereas only datasets 1 and 2 are shown in the plot.

The first dataset keeps track of the step number and the time in the first and second column, respectively. The remaining datasets use time for the first column, but this column is suppressed so only the time values from the first dataset are displayed in the table.

## 6.7 Dataset Tables

125

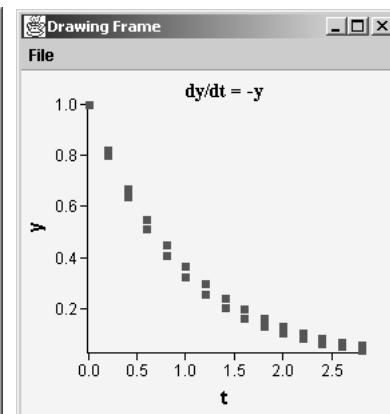
**TABLE 6.5** Common data table methods.**org.opensourcephysics.display.DataTable methods**


---

<code>setRefreshDelay</code>	Sets a short delay before table is redrawn on the screen. A delay is useful if table data changes frequently during an animation. A 500 ms delay minimizes unnecessary rendering and may reduce screen flicker.
<code>setStride</code>	Sets the stride in the data table. The stride allows a table to skip rows so that very long data sets can be displayed. For example, a stride of ten shows every tenth row. The default stride is one.
<code>setXColumnVisible</code>	Sets the visibility of the x column of the dataset in a table view.
<code>setYColumnVisible</code>	Sets the visibility of the Y column of the dataset in a table view.
<code>setXYColumnNames</code>	Sets the column names when rendering the dataset in a JTable.

---

n	t_n	Exacty	Euler y	Error
0	0	1	1	0
1	0.2	0.819	0.8	0.019
2	0.4	0.67	0.64	0.03
3	0.6	0.549	0.512	0.037
4	0.8	0.449	0.41	0.04
5	1	0.368	0.328	0.04
6	1.2	0.301	0.262	0.039
7	1.4	0.247	0.21	0.037
8	1.6	0.202	0.168	0.034
9	1.8	0.165	0.134	0.031
10	2	0.135	0.107	0.028
11	2.2	0.111	0.086	0.025
12	2.4	0.091	0.069	0.022
13	2.6	0.074	0.055	0.019
14	2.8	0.061	0.044	0.017



(a) Data Table.

(b) Graph.

**FIGURE 6.6** A comparison of the exact and Euler's method solution to a differential equation.

Listing 6.6 A comparison of the exact and Euler solution to the exponential decay equation.

```

1 package org.opensourcephysics.manual.ch06;
2 import org.opensourcephysics.display.*;

```

```

3  import javax.swing.JFrame;
4
5  public class ExponentialEulerApp {
6      DatasetManager datasetManager;
7      DataTable dataTable;
8      DataTableFrame tableFrame;
9      PlottingPanel plottingPanel;
10     DrawingFrame drawingFrame;
11     double t0 = 0.0;                      // initial value of t
12     double y0 = 1.0;                      // initial value of y
13     double tmax = 3.0;                    // t value that stops the
14         calculation
15     double dt = 0.10;                     // step size
16
17     public ExponentialEulerApp() { // ExponentialEulerApp
18         constructor
19         // set up the dataset manager
20         datasetManager = new DatasetManager();
21         datasetManager.setXPointsLinked(true); // X only for 0-th
22             dataset
23         datasetManager.setXYColumnNames(0, "n", "t_n");
24         datasetManager.setXYColumnNames(1, "t", "Exact y");
25         datasetManager.setXYColumnNames(2, "t", "Euler y");
26         datasetManager.setXYColumnNames(3, "t", "Error");
27         // set up table
28         dataTable = new DataTable();
29         tableFrame = new DataTableFrame(dataTable);
30         dataTable.add(datasetManager); // add datasetManager to table
31         tableFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32         tableFrame.setVisible(true);
33         // set up graph
34         plottingPanel = new PlottingPanel("t", "y", "dy/dt = -y");
35         drawingFrame = new DrawingFrame(plottingPanel);
36         plottingPanel.addDrawable(datasetManager.getDataset(1)); // exact
37         plottingPanel.addDrawable(datasetManager.getDataset(2)); // Euler
38         drawingFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39         drawingFrame.setVisible(true);
40     }
41
42     public void calculate() {
43         double t = t0; // current value of x
44         double y = y0; // current value of the Euler solution
45         int counter = 0;
46         while(t<=tmax) {
47             datasetManager.append(0, counter, t); // store counter and
48                 x
49             double exact = Math.exp(-t);
50         }
51     }
52 }
```

## 6.8 Programs

**127**

```

46     datasetManager.append(1, t, exact); // store t and exact
47             x
48     datasetManager.append(2, t, y); // store Euler
49             solution
50     datasetManager.append(3, t, exact-y); // store Euler
51             solution
52     y = y-y*dt; // increase y
53     t = t+dt; // increase t
54     counter++; // number of steps
55 }
56
57 public static void main(String[] args) {
58     ExponentialEulerApp app = new ExponentialEulerApp();
59     app.calculate();
60 }
61 }
```

## 6.8 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch06` package.

### DatasetApp

`DatasetApp` tests the `Dataset` class. This program is described in Section 6.2. Other programs that demonstrate how to use datasets are available: `MarkerAreaApp`, `MarkerBarApp`, `MarkerCircleApp`, `MarkerConnectedApp`, `MarkerPixelApp`, and `MarkerPostApp`.

### DatasetManagerApp

`DatasetManagerApp` demonstrates how to create three datasets using a dataset manager and how to append data points to these datasets. This program is described in Section 6.3.

### DataTableApp

`DataTableApp` creates and displays a data table containing 3 datasets. This program is described in Section 6.7.

### ExponentialEulerApp

`ExponentialEulerApp` uses a dataset manager and a data table to show the numeric and analytic solution of a simple differential equation,  $dy/dt = -1$ . The numeric solution is obtained using Euler's method in Section 6.7.

**GaussianPlotApp**

GaussianPlotApp simulates a best fit to a Gaussian spectral line using a PlotFrame with two datasets as described in Section 6.4. These datasets use different drawing styles.

**KeplerPlotApp**

KeplerPlotApp demonstrates how to create log-log plot using data from planetary orbits. The program demonstrates the validity of Kepler's second law by plotting the log of the planet's semi-major axis vs. the log of the orbital period. This program is described in Section 6.6.

**LowpassFilterApp**

LowpassFilterApp plots the frequency response for a lowpass RC filter. The output is shown in Figure 6.1.

**MarkerApps**

MarkerAreaApp, MarkerBarApp, MarkerCircleApp, MarkerConnectedApp, MarkerPixelApp, and MarkerPostApp demonstrate how to use a dataset with different marker styles as described in Section 6.4

**PolarPlottingApp**

PolarPlottingApp and Polar2PlottingApp demonstrate how to create a plotting panel with polar coordinate axes as described in Section 6.5

## CHAPTER

## 7

## Animation, Images, and Buffering

©2005 by Wolfgang Christian

Tools and techniques for producing animations are presented. Because animations use image buffers, techniques for creating and storing images are described.

### 7.1 ■ ANIMATION

A drawing panel paints drawable objects one at a time in the panel's `paintComponent` method as described in Chapter 4. This drawing is usually done by invoking the component's `repaint` method. Invoking `repaint` does not, however, repaint the screen directly but places a request into the program's *event queue*. The Java VM will invoke the `paintComponent` method at a later time from the *event dispatch thread* and may, in fact, collapse multiple repaint requests into a single operation. Because there is no way of knowing when painting takes place, painting may not occur at a constant rate and frames may be dropped.<sup>1</sup>

The `RepaintApp` program shown in Listing 7.1 attempts to produce a fast animation by drawing a rotating spiral. (The `Spiral` class is a simple implementation of `Drawable` and is not shown but is available on the CD.) The animation invokes the `sleep` method after invoking the `repaint` method to give the event dispatch thread access to system resources. The sleep time has been set to one millisecond in an attempt to produce the fastest possible frame rate.

The animation produced by `RepaintApp` is unsatisfying because the frame rate is not uniform and the speed of the rotation depends on the computer and the operating system. In order to produce a smooth animation it is necessary to control the frame rate. A smooth animation paints (renders) images onto a screen faster than the human eye's response type. Film, for example, displays frames twenty times per second and video rates are even higher. Although high frame rates are possible when image data are prerecorded and when hardware is optimized to process this data, high frame rates may not be possible if both the data and the data's visualization are computed in real time. Because lowering the frame rate to ten frames per second allows more time to compute interesting physics we usually use this setting. Execute `RepaintApp` with a sleep time of 100 milliseconds and note the improvement in the animation. It is usually better

<sup>1</sup> The situation is similar for other events such as a button clicks or a keyboard entry. An event is placed into the dispatch queue and this queue is processed sequentially by the event dispatch thread. Because Swing graphical user interface components are not thread safe, they should only be accessed from this thread.

**LISTING 7.1** The `repaint` method places a request into the *event queue*. The *event dispatch thread* processes these requests in the order received but processing may be delayed if there are other pending events.

```

1 package org.opensourcephysics.manual.ch07;
2 import org.opensourcephysics.display.*;
3 import javax.swing.JFrame;
4
5 public class RepaintApp {
6     static int sleepTime = 1;
7
8     public static void main(String[] args) {
9         DrawingPanel panel = new DrawingPanel();
10        DrawingFrame frame = new DrawingFrame("Direct Drawing",
11            panel);
12        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13        frame.setVisible(true);
14        Spiral spiral = new Spiral();
15        panel.addDrawable(spiral);
16        while(true) {
17            spiral.theta += 0.1;
18            try {
19                Thread.sleep(sleepTime); // allow the event queue to
20                // redraw
21            } catch(InterruptedException ex) {}
22            panel.repaint();
23        }
24    }
25}
```

to have a rate that produces smooth motion at all times than to run as fast as possible for several frames and then slow to a crawl because another thread is playing catch up.

The `AbstractAnimation` class in the controls package is designed to produce animations with a constant frame rate. This class implements the `Animation` interface and the `Runnable` interface so that it can work with both the `AnimationControl` class and the `Thread` class. The `startAnimation` method in the `AbstractAnimation` class spawns a `Thread` named `animationThread` and this thread is passed a reference to the `Runnable` instance. The thread invokes the animation's `run` method which is shown in Listing 7.2. The `doStep` method is abstract and must be implemented in a concrete subclass.

The `run` method in `AbstractAnimation` reads the system time before and after invoking the `doStep` method and the difference in these times is used to set the sleep time. Although the `System.currentTimeMillis` method only has a time resolution of 10 milliseconds, this coarse resolution is adequate for our

**LISTING 7.2** The run method in the `AbstractAnimation` class determines the frame rate.

```

1 // AbstractAnimation method
2 public void run(){
3     long sleepTime = delayTime;
4     while (animationThread==Thread.currentThread()){
5         long currentTime = System.currentTimeMillis();
6         doStep();
7         // adjusts sleep time for a more uniform animation rate
8         sleepTime =
9             Math.max(1,delayTime-(System.currentTimeMillis()-currentTime));
10        try{
11            Thread.sleep(sleepTime);
12        }
13        catch (InterruptedException ie){}
14    }
15 }
```

simulations<sup>2</sup>.

The `DirectAnimationApp` class shown in Listing 7.3 extends `AbstractAnimation` and implements the `doStep` method. This method sets the spiral's starting angle and invokes the panel's repaint method.

Painting objects directly onto a panel works fine in the case of static graphical user interfaces or if the object being drawn is simple. But this approach is not well-suited for animation if the object is complex or if the state of the object changes while it is being drawn. Data may be corrupted because another thread modifies values while the object is being painted. Although it is possible to avoid data corruption by making copies of data and by using synchronization techniques, it is simpler to use a single Thread. This single-thread design pattern is used in the next section to synchronize both the physics computation and the data visualization.

Listing 7.3 `DirectAnimationApp` demonstrates how to create an animation by extending `AbstractAnimation`.

```

1 package org.opensourcephysics.manual.ch07;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.*;
4
5 public class DirectAnimationApp extends AbstractAnimation {
6     DrawingPanel panel = new DrawingPanel();
7     DrawingFrame frame = new DrawingFrame("Direct Drawing", panel);
8     Spiral spiral = new Spiral();
```

<sup>2</sup>The resolution can be improved using the `System.nanoTime` method introduced in Java version 1.5 (JDK 5.0).

```

9   double dtheta = 0.1;
10  int counter = 0;
11  long startTime = 0;
12
13  public DirectAnimationApp() {
14      panel.setPreferredMinMax(-5, 5, -5, 5);
15      panel.addDrawable(spiral);
16      frame.setVisible(true);
17  }
18
19  public void startAnimation() {
20      delayTime = control.getInt("delay time (ms)");
21      super.startAnimation();
22      counter = 0;
23      startTime = System.currentTimeMillis();
24  }
25
26  public void stopAnimation() {
27      float rate = counter
28          /(float) (System.currentTimeMillis()-startTime);
29      super.stopAnimation();
30      panel.repaint();
31      control.println("frames pr second="+1000*rate);
32      control.println("ms per frame="+(1.0/rate));
33  }
34
35  protected void doStep() {
36      spiral.theta += dtheta;
37      panel.repaint();
38      counter++;
39  }
40
41  public void resetAnimation() {
42      control.setValue("delay time (ms)", 100);
43  }
44
45  public static void main(String[] args) {
46      AnimationControl.createApp(new DirectAnimationApp());
47  }
48 }
```

## 7.2 ■ BUFFERED PANELS

In order to create smooth animation at a constant frame rate without on-screen flashing or tearing it is common to draw onto a hidden image known as a *buffer* and then copy the buffer to the screen all at once. Swing components already have a built in buffer and this buffer is used automatically in graphical user interfaces

## 7.2 Buffered Panels

133

without having to worry about the buffer's details. Swing components are not, however, thread safe and it is necessary to create our own buffers if we wish to render frames from an animation thread.

Setting a drawing panel's buffered option to true creates two image buffers. We refer to one of these images as the offscreen buffer and second as the working buffer. If a panel is buffered, the panel's `paintComponent` method does not recreate the drawing. Its only job is to copy the offscreen buffer onto the screen. The actual drawing is done by painting drawable objects onto the working buffer using the `render` method. After rendering is complete the working buffer and the display buffer are switched and the new image is ready to be copied to the screen. Data integrity is insured because the physics computation and the rendering are done sequentially in the animation thread.

Listing 7.4 enables buffering by setting the drawing panel's buffered option in the program's constructor. The buffer's content is drawn by invoking the panel's `render` method in the animation's `doStep` method.

Listing 7.4 A buffered animation.

```

1 package org.opensourcephysics.manual.ch07;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.*;
4
5 public class BufferedAnimationApp extends AbstractAnimation {
6     DrawingPanel panel = new DrawingPanel();
7     DrawingFrame frame = new DrawingFrame("Buffered Drawing",
8         panel);
9     Spiral spiral = new Spiral();
10    double dtheta = 0.1;
11    int counter = 0;
12    long startTime = 0;
13
14    public BufferedAnimationApp() {
15        panel.setPreferredSize(new Dimension(500, 500));
16        panel.addDrawable(spiral);
17        panel.setBuffered(true); // creates offscreen buffer
18        frame.setVisible(true);
19    }
20
21    public void startAnimation() {
22        panel.setIgnoreRepaint(true);
23        delayTime = control.getInt("delay time (ms)");
24        super.startAnimation();
25        counter = 0;
26        startTime = System.currentTimeMillis();
27    }
28
29    public void stopAnimation() {
        float rate = counter

```

```

30             / ( float ) ( System . currentTimeMillis () - startTime );
31         super . stopAnimation ();
32         panel . setIgnoreRepaint ( false );
33         panel . repaint ();
34         control . println ( "frames pr second=" + 1000 * rate );
35         control . println ( "ms per frame=" + ( 1.0 / rate ) );
36     }
37
38     protected void doStep () {
39         spiral . theta += dtheta;
40         panel . render ();
41         counter++;
42     }
43
44     public void resetAnimation () {
45         control . setValue ( "delay time (ms)" , 100 );
46     }
47
48     public static void main ( String [] args ) {
49         AnimationControl . createApp ( new BufferedAnimationApp () );
50     }
51 }
```

The `BufferedAnimationApp` class has another difference from the `DirectAnimationApp` class. Because we know that the drawing will be updated every 1/10 second from within the animation thread, we disable paint messages received from the operating system. This is done within the `startAnimation` method by setting the ignore repaint flag to true.

```
1 panel . setIgnoreRepaint ( true ); // invoked within startAnimation
```

The `startAnimation` method in `BufferedAnimationApp` must, of course, call the superclass `startAnimation` method to start the animation thread. The order of the methods is reversed in the `stopAnimation` method. The superclass `stopAnimation` method is first invoked to stop the thread and then the `ignoreRepaint` flag is set to false.

The drawing panel's `render` method returns the current offscreen buffer, but this image may be invalid if the frame containing the image is hidden or iconified. In this case, the `render` method returns the image that was rendered when the frame was last visible. Because the properties of the internal image buffer depend on the panel's current on-screen state, it is safer to pass an image to the `render` method if an image is needed for further processing.

```
1 BufferedImage img = new BufferedImage ( width , height ,
2                                         BufferedImage . TYPE_INT_RGB );
2 panel . render ( img ); // draws into the image
```

Saving images is a convenient way to record the time development of a model or to compare results obtained with different input parameters. See Chapter 13 for

## 7.3 Buffer Strategy

135

examples of how to record a video of an animation.

The following guidelines summarize how to use a drawing panel's offscreen image buffers to produce an animation.

1. Invoke the drawing panel's `setBuffered(boolean buffered)` method to enable buffering.
2. Disable repainting from the operating system when an animation thread is running and enable repainting when the animation is stopped.
3. Invoke the panel's `render` method whenever the panel's data are changed. Calling `render` is the preferred method for animation because the rendering is done in the calling method's thread.

Buffering has one disadvantage. It increases an application's consumption of display memory. An  $800 \times 600$  window with a color depth of 32 bits per pixel allocates 1.9 MByte of memory. This is usually not a problem on modern operating systems but you should consider using smaller fixed-size windows if the application requires numerous windows.

### 7.3 ■ BUFFER STRATEGY

You may notice an occasional ripple on your computer monitor when an animation is not synchronized to the monitor's refresh rate. This ripple is known as tearing and can be eliminated by copying the offscreen buffer to the video memory card while the electronics are being reset between refresh cycles. The `BufferStrategy` class introduced in Java version 1.4 takes care of these details.

Sun's `BufferStrategy` class creates and manages offscreen buffers and copies these buffers to the monitor without tearing. A `BufferStrategy` instance is easy to create as shown in the following code fragment.

```

1 JFrame frame = new JFrame();
2 frame.createBufferStrategy(2); // creates two buffers
3 BufferStrategy strategy = frame.getBufferStrategy();
```

In order to use a `BufferStrategy`, we obtain a graphics context from the strategy, draw the frame's contents, and tell the strategy to show the next buffer.

```

1 Graphics g = strategy.getDrawGraphics();
2 // insert code to draw here
3 strategy.show();
```

Sun claims that the `BufferStrategy` class optimizes the creation of image buffers because it creates these buffers in hardware-accelerated video memory and we have therefore implemented a `BufferStrategy` drawing method in the `OSPFFrame` class. Invoking the `bufferStrategyShow` method in `OSPFFrame` draws the frame's contents. The `BufferStrategyApp` class demonstrates this

animation technique but it not shown here because it is similar to our previous examples. The only change that is needed is that the `doStep` method invoke `bufferStrategyShow` in the drawing frame rather than the `render` method.

```

1 protected void doStep () {
2   spiral.theta += dtheta;
3   frame.bufferStrategyShow ();
4   counter++;
5 }
```

Although the `BufferStrategy` class is easy to use and gives smooth animation, we have found that it does not give the fastest frame rate. This may be because the code suspends execution until the image buffer is copied between video refreshes, or it may be due to a poor optimization strategy. The `BufferStrategy` class also fails using the Java 1.4 VM on dual monitor systems if the frame straddles both monitors or if the animation frame is dragged from one monitor to another. The `BufferStrategy` class is, however, the animation mechanism recommended by Sun and we have therefore implemented this mechanism in the OSP library. The `BufferStrategyApp` on the CD shows how this implementation is used.

## 7.4 ■ SIMULATION ARCHITECTURE

Typical physics simulations have a common architecture that is independent of the system being modeled. A simulation gathers input and starts an animation thread that periodically updates a model's state and renders views of the model's data. The `SimulationControl` class and `AbstractSimulation` class are designed to support this architecture with a minimum amount of programming.

Because simulations usually change data, the `AbstractSimulation` class automatically updates its drawing frames after invoking the `doStep` method if the frame's `animated` property is true. The simulation framework also assumes that data accumulated in graphs and other views should be cleared when the program is initialized or reset. The `SimulationControl` automatically clears data from a frame if the frame's `autoclear` property is set. Both the `animated` and `autoclear` properties are disabled in the `OSPFrame` class but are enabled for subclasses defined in the `frames` package. These properties can be set by invoking the following methods:

```

1 // frame is an instance of OSPFrame
2 frame.setAnimated(true);
3 frame.setAutoClear(true);
```

An `AbstractAnimation` creates an animation thread in its `startAnimation` method and stops this thread in the `stopAnimation` method. Concrete subclasses of `AbstractAnimation` that override these methods should start and

## 7.4 Simulation Architecture

137

stop this thread by calling their superclass implementations. Concrete realizations of `AbstractAnimation` should also invoke superclass implementations of `initializeAnimation` and `resetAnimation` when overriding these methods. The `AbstractSimulation` class provides convenience methods named `start`, `stop`, `initialize`, and `reset` that are invoked when the control's buttons are pressed. Because these methods are guaranteed to be empty in the superclass, concrete subclasses need not invoke these overridden methods in the superclass. The `start` and `stop` methods are invoked before a thread is started and after a thread is stopped, respectively.

One additional feature that we have found useful is the ability to execute a block of code in conjunction with the `doStep` method. The `startRunning` and `stopRunning` methods are invoked before an animation thread starts and after it stops, respectively. They are also invoked when an animation is single stepped using the `SimulationControl` step button but are not invoked during the animation.

Listing 7.5 demonstrates the OSP simulation framework by again implementing a rotating spiral. Note the statements that have been removed by comparing this program to Listing 7.4.

Listing 7.5 A simulation automatically updates its views after invoking the `doStep` method.

```

1 package org.opensourcephysics.manual.ch07;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.frames.*;
4
5 public class RenderApp extends AbstractSimulation {
6     DisplayFrame frame = new DisplayFrame("Direct Drawing");
7     Spiral spiral = new Spiral();
8     double dtheta = 0.1;
9     int counter = 0;
10    long startTime = 0;
11
12    public RenderApp() {
13        frame.setPreferredMinMax(-5, 5, -5, 5);
14        frame.addDrawable(spiral);
15        frame.setVisible(true);
16    }
17
18    public void startRunning() {
19        delayTime = control.getInt("delay time (ms)");
20        counter = 0;
21        startTime = System.currentTimeMillis();
22    }
23
24    public void stopRunning() {
25        float rate = counter
26            /(float) (System.currentTimeMillis()-startTime);

```

```

27   control.println("frames per second="+1000* rate);
28   control.println("ms per frame="+(1.0/ rate));
29 }
30
31 public void reset() {
32   control.setAdjustableValue("delay time (ms)", 100);
33 }
34
35 protected void doStep() {
36   spiral.theta += dtheta;
37   counter++;
38 }
39
40 public static void main(String[] args) {
41   SimulationControl.createApp(new RenderApp());
42 }
43 }
```

## 7.5 ■ DRAWABLE BUFFER

It is inefficient to redraw a panel at every time step if the appearance of most objects within the drawing do not change during an animation. A better approach is to draw a static background image, copy this image into the drawing panel, and then draw only those objects that change during the animation. A **DrawableBuffer** is a drawable object that creates and manages a background image that is the same size as the drawing panel that contains the buffer.

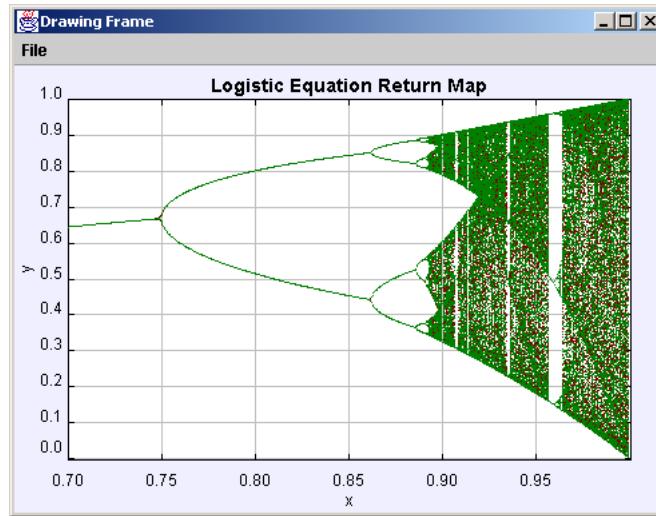
A **DrawableBuffer** acts in many respects like a buffered drawing panel. Drawable objects are added to the buffer and the buffer stores references to these objects for later rendering. Because the visual representation of the **DrawableBuffer** is an image, it is the programmer's responsibility to redraw the image if the objects change their state. This can be done by invoking the buffer's `updateImage(DrawingPanel drawingPanel)` method. You can also set the buffer's invalid image flag and a new image will be rendered when the panel containing the **DrawableBuffer** is repainted. This is done using the buffer's `invalidateImage` method.

Because drawing panels assumes that drawable buffers contains only static objects buffers, there is a limitation when they are used with interactive objects. A drawing panel can only locate the buffer and not the objects in the buffer. One way around this limitation is to add an invisible proxy component to the interactive drawing panel. This proxy intercepts mouse events and forwards them to objects within the buffer.

A second limitation is that a drawable buffer should be added to only one drawing panel because the off screen image uses the panel's dimension to set the image size. (A drawable buffer can be added to more than one drawing panel if these panels have the same dimension. In this case, the same offscreen image will

## 7.6 Data Raster

139



**FIGURE 7.1** A data raster showing the bifurcation diagram of the logistic map.

be copied to multiple panels.)

It is usually not necessary to use both a `DrawableBuffer` and a buffered drawing panel. (This would be “triple buffering.”) A `DrawableBuffer` is most effective when there are a small number of animated objects that can be drawn directly on a static background. See Section 8.5 for an example of how a `DrawableBuffer` is used to draw a particle on a background contour plot.

## 7.6 ■ DATA RASTER

Drawable components can implement their own buffering. The `DataRaster` component, for example, is designed to display tens of thousands of data points by coloring one image pixel for every point. Because it would be inefficient to redraw every point when a new point is added, a `DataRaster` creates an image of the drawing panel and colors only the pixel where the datum is located. When the drawing panel is repainted, this image is copied to the screen. If the size or the scale of the drawing panel are changed, then the image is recreated from the raw data.

Listing 7.6 uses a data raster to display the logistic equation return map. The logistic equation was first proposed by Robert May as a simple model of population dynamics. This equation can be written as a one-dimensional difference equation that transforms the population in one generation,  $x$ , into a succeeding generation,  $x_{n+1}$ . Because the population is scaled so that the maximum value is one, the domain of  $x$  falls on the interval  $[0, 1]$ .

$$x_{n+1} = 4rx_n(1 - x_n) \quad (7.1)$$

The behavior of the logistic equation depends on the value of the growth parameter,  $r$ . If the growth parameter is less than a critical value  $r \approx 0.75$ , then  $x$  approaches a stable fixed value. Above this value for  $r$ , the behavior of  $x$  begins to change as shown in Figure 7.1. First the population begins to oscillate between two values. If  $r$  increases further, then  $x$  oscillates between four values. Then eight values. This doubling ends when  $r > 0.8924864\dots$  after which almost any  $x$  value is possible.

The bifurcation diagram (7.1) shows the behavior of the logistic equation by plotting successive values of  $x$  after the initial transient behavior is discarded. Listing 7.6 calculates and discards 400 iterations. It then plots 200 iterations of  $x$  in red followed by another 200 iterations in green. This process is repeated 6000 times as the growth parameter is incremented thereby producing a data raster that contains 1.2 million data points.

Listing 7.6 The logistic equation demonstrates how very large datasets can be plotted using a data raster.

```

1 package org.opensourcephysics.manual.ch07;
2 import org.opensourcephysics.display.DrawingFrame;
3 import org.opensourcephysics.display.PlottingPanel;
4 import org.opensourcephysics.display2d.DataRaster;
5 import java.awt.Color;
6 import javax.swing.JFrame;
7
8 public class LogisticApp {
9     public static void main(String[] args) {
10         PlottingPanel panel = new PlottingPanel("x", "y",
11                                         "Logistic Equation Return Map");
12         DrawingFrame frame = new DrawingFrame(panel);
13         DataRaster dataRaster = new DataRaster(panel, 0.7, 1, 0, 1);
14         panel.addDrawable(dataRaster);
15         int nplot = 400;
16         dataRaster.setColor(0,
17             new Color(128, 0, 0, 128)); // first dataset is dark red
18         dataRaster.setColor(1,
19             new Color(0, 128, 0, 128)); // second dataset is dark
20         for(double r = 0.7;r<1.0;r += 0.0005) {
21             double x = 0.5; // starting value for x
22             for(int i = 1;i<=nplot;i++) { // nplot values not plotted
23                 x = 4*r*x*(1-x);
24             }
25             for(int i = 1;i<=nplot/2;i++) {
26                 x = 4*r*x*(1-x);
27                 dataRaster.append(0, r,
```

## 7.7 Programs

141

```

28         x); // show x-values for current
29         value of r
30     }
31     for(int i = 1; i <= nplot/2; i++) {
32         x = 4*r*x*(1-x);
33         dataRaster.append(1, r, x); // note different data
34     }
35     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36     frame.setVisible(true);
37 }
38 }
```

## 7.7 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch07` package.

**BufferedAnimationApp**

`BufferedAnimationApp` tests the frame rate using drawing panel buffering.

**BufferStrategyApp**

`BufferStrategyApp` tests the frame rate using a Java `BufferStrategy` for drawing.

**DataRasterApp**

`DataRasterApp` tests the `DataRaster` class by creating 100,000 random data points. Half the points are shown in red and half in green.

**DirectAnimationApp**

`DirectAnimationApp` tests the frame rate and the animation quality by invoking the `repaint` method. The drawing is rendered from within the event dispatch thread when the component is painted.

**LogisticApp**

`LogisticApp` tests the `DataRaster` class by creating a return map for the logistic function. This program is described in Section 7.6.

**RenderApp**

`RenderApp` tests the frame rate using the `render` method. An `AbstractSimulation` renders the drawing within the animation thread by after invoking the `doStep` method.

### **RepaintApp**

RepaintApp demonstrates erratic drawing using repaint requests.

## CHAPTER

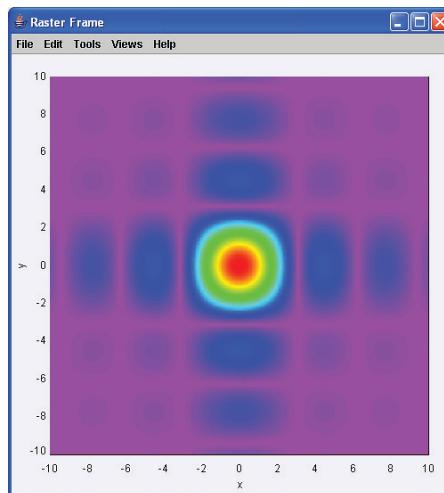
## 8

## Visualization of Two Dimensional Fields

©2005 by Wolfgang Christian

Visualization tools for two-dimensional scalar fields, vector fields, and complex fields are defined in the `org.opensourcephysics.display2d` package. Sample programs are located in the manual's `ch08` package unless otherwise stated.

### 8.1 ■ OVERVIEW



**FIGURE 8.1** Diffraction pattern from a 2D rectangular aperture illuminated by a monochromatic light source. The color changes from blue to red as the energy density increases.

Imagine a sheet of metal that is heated at an interior point and cooled along its edges. In principle, the temperature of the sheet can be measured at any point,  $T(x, y)$ . Calculating this temperature is a typical computational physics problem. A quantity, such as temperature, pressure, or light intensity, that is defined by a single number (scalar) at every point in space is known as a *scalar field*. Other types of fields are possible. Consider the force on a small test charge in the vicinity of other charges,  $\mathbf{F}(x, y)$ . Because the force is a vector, this situation defines a

*vector field.* We now describe a framework to display two-dimensional scalar and vector fields.

One way to store and display information in two-dimensional simulations is to divide space into a discrete grid and to store data at every grid point. A scalar field sampled at every grid point can be stored in a 2D array. This array is passed to an object in the `display2d` package to produce the visualization.

Figure 8.1 shows the two-dimensional diffraction pattern from a rectangular aperture. This pattern has an analytic solution

$$f(x, y) = I_0 \left( \frac{\sin x}{x} \right)^2 \left( \frac{\sin y}{y} \right)^2 \quad (8.1)$$

where we have chosen dimensionless units for  $x$  and  $y$ . Listing 8.1 plots this solution using a `Scalar2DFrame`. All that is required is that an array of values and a world coordinate scale be passed to the `Scalar2DFrame` using the `setAll` method.

A scalar field has a scale given by the locations of its corners in world units. This scale is set when data are passed to the frame. If the scale is not specified, the previously set scale is used.

```
1 frame.setAll(data, -10, 10, -10, 10); // sets data and scale
2 frame.setAll(data); // sets only data
```

Because it is convenient to be able to convert between array indices and the corresponding world coordinates, OSP scalar and vector field visualizations define array index to world coordinate conversion methods. For example, the following code fragment computes the  $x$  value of the tenth column and the array index closest to  $x = 0.5$ .

```
1 double x = frame.indexToX(10); // world coordinates for index 10
2 int i = frame.xToIndex(0.5); // index closest to x=0.5
3 // similar methods for y
```

Because these conversions use the current scale, it is important that the scale be set before these methods are invoked. This is why the `setAll` method is invoked using the uninitialized array with minimum and maximum values before `indexToX` and `indexToY` methods are invoked. The `setAll` method is invoked a second time without a scale after the diffraction pattern has been computed.

The `Scalar2DFrame` class plots arrays of floating point numbers. The frame's Views menu allows us to display this data as a grid plot, a contour plot, an interpolated plot, or a surface plot. The `org.opensourcephysics.display2d` package provides visualizations for other types of fields as shown in Table 8.1. For example, a 2D-vector field or a complex 2D-scalar field require at least two numbers (components).

Table 8.2 shows the rendering time in milliseconds for some OSP 2D visualizations. The times shown were obtained using the Sun Microsystems Java 1.4 VM running on Windows 2000 and Windows XP using a 400 MHz Pentium 2 and

## 8.1 Overview

## 145

**LISTING 8.1** DiffractionApp plots the analytic solution to the diffraction pattern from a 2D rectangular aperture.

```

1 package org.opensourcephysics.manual.ch08;
2 import org.opensourcephysics.frames.*;
3
4 public class DiffractionApp {
5     public static void main(String[] args) {
6         Scalar2DFrame frame = new Scalar2DFrame("x", "y", "Raster
7             Frame");
8         int nx = 512, ny = 512;
9         double [][] data = new double [nx][ny];
10        frame.setAll(data, -10, 10, -10,
11                      10); // sets the scale for this data
12        for(int i = 0;i<nx;i++) {
13            double x = frame.indexToX(i); // gets the x value for
14                the index
15            double xAmp = (x==0) ? 1 : Math.sin(x)/x;
16            for(int j = 0;j<ny;j++) {
17                double y = frame.indexToY(j); // gets the y value for
18                    the index
19                double yAmp = (y==0) ? 1 : Math.sin(y)/y;
20                double amp = xAmp*yAmp;
21                // sqrt of intensity gives better visibility
22                data[i][j] = Math.sqrt(amp*amp);
23            }
24        }
25    }
26 }
```

a dual 2.6 GHz Pentium 4 processor, respectively. These times are for rendering only. The overhead needed to generate the data is not included.

Using the timing results in Table 8.2 as a guide, we see that it may not be possible to produce a ten frame per second animation on a low end machine using some visualizations if the grid density is high or if multiple views are being drawn. Notice that the rendering times for lattice and raster components are insensitive to grid size because data are copied quickly and directly from an array into an image.

There are several ways to improve performance after a program is running correctly. The easiest technique is to void unnecessary screen repainting by performing several computation steps before the panel's `redraw` or `render` methods are invoked. Methods, such as `indexToX`, that are invoked frequently but perform straightforward and routine computations can be optimized for the given applica-

**TABLE 8.1** Two-dimensional visualization components.

<b>org.opensourcephyics.display2d package</b>	
BinaryLattice	An array of ones and zeros shown as a two-color grid.
CellLattice	An array of 256 possible values shown as multi-color grid rectangles.
ByteRaster	An array of 256 possible values shown using one pixel per value.
GridPlot	An array of numbers shown as multi-colored rectangles.
ComplexGridPlot	An array of complex numbers shown as a multi-color grid. Color and intensity are used to represent phase and magnitude, respectively.
ComplexInterpolatedPlot	A variant of a complex grid plot whose complex numbers have been interpolated at every pixel. This interpolation smooths the image.
ComplexSurfacePlot	A 3d surface whose height is proportional to the magnitude of a complex number and whose color represents phase.
ContourPlot	A contour plot.
GrayscalePlot	An array of numbers shown as a gray-scale grid.
InterpolatedPlot	A variant of a grid plot whose numbers have been interpolated at every pixel. This interpolation smooths the image.
SiteLattice	An array of 256 possible values shown as circles at the intersections of grid lines.
SurfacePlot	A 3d surface whose height is proportional to a scalar.
VectorPlot	A vector field plot.

tion and the optimized code can be in-lined to remove the method call. Java 2D components such as the `WritableRaster` can be used directly. If a significant portion of a visualization is static, you should consider creating a background image as described in Section 8.5. Special components , such as `DataRaster` described in Section 7.6, have been programmed so that very little computation is required when data are added. And finally, a program may render a low-resolution visualization when the program is running and high-resolution visualization when the animation stops.

## 8.2 Images and Rasters

147

**TABLE 8.2** Rendering times in milliseconds ( $\pm 10$  ms) for visualizations in the 2d-display package. The first number is the time recorded on a 400 MHz Pentium II. The second number is the time recorded on a 2.6 GHz dual-processor Pentium 4. The drawing panel size is  $300 \times 300$  pixels.

component	$16 \times 16$	$32 \times 32$	$64 \times 64$
BinaryLattice	20 10	30 10	40 10
CellLattice	20 0	30 0	30 0
ByteRaster	10 0	10 0	10 0
GridPlot	20 0	20 10	30 10
ComplexGridPlot	20 0	20 10	30 10
ComplexInterpolatedPlot	120 50	120 50	120 50
ComplexSurfacePlot	100 30	200 70	600 200
ContourPlot	140 70	480 200	2000 600
InterpolatedPlot	120 50	120 50	120 50
SurfacePlot	100 30	200 70	600 200
VectorPlot	50 15	110 60	410 200

## 8.2 ■ IMAGES AND RASTERS

An image in which pixels are color coded can be used to generate a visually appealing image of a scalar field. The computation is carried out using an  $n \times m$  array in which every array element corresponds to an image pixel. The Java `java.awt.image` package contains a `WritableRaster` class that enables a program to access image pixels.<sup>1</sup> In fact, a Java raster is used in the `DataRaster` class described in Section 7.6. A number of the visualizations in the `display2d` package convert their data into an off-screen image that is later copied to an output device. `BinaryLattice`, `GridPlot`, and `GrayscalePlot` use this technique. After the image pixels are colored, the image is copied to a computer display or printer. This rendering time for images is fast because image-based operations are usually performed on a video card and not by the main CPU.

The `ByteRaster` class in the `display2d` package simplifies the process of converting an array of bytes onto an image. A `byte` variable is a signed integer that takes on 256 values ranging from `Byte.MIN_VALUE=-128` to `Byte.MAX_VALUE=127` so that it can be stored in an eight bit memory location.<sup>2</sup> Incrementing the maximum value wraps around to produce the minimum value. Because casting an `int` to a `byte` folds the value into the range  $[-128, 127]$ , we sometimes perform a computation using integers and cast the results to bytes to produce a color palette that repeats every 256 values.

Listing 8.2 shows the `ByteRaster` class being used to display the ever popular Mandelbrot set. The Mandelbrot set consists of points,  $c$ , in the complex plane that obey the following rule:

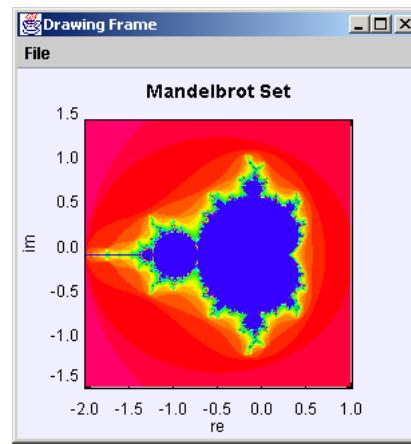
<sup>1</sup>The term raster was originally applied to the pattern of horizontal lines that appeared on a television or computer screen.

<sup>2</sup>A byte is always a signed number in Java. Java does not support an unsigned 8-bit data type.

1. Start with the complex number,  $z = 0 + i0$ .
2. Generate a new complex number,  $z'$ , by multiplying  $z$  by itself and adding the result to  $c$ .

$$z' = z^2 + c \quad (8.2)$$

3. Repeat steps [1] and [2]. If the complex number  $z$  goes toward infinity, then the starting point,  $c$ , is not a member of the Mandelbrot set. All numbers that remain bounded are members of the set.



**FIGURE 8.2** The Mandelbrot set.

It can be shown that if the magnitude of  $z$  is greater than 2, then  $z$  will approach infinity. The `MandelBrotApp` code assumes that the number  $c$  is in the Mandelbrot set if  $|z| < 2$  after 256 iterations. In order to show how rapidly a number fails the test, we color the pixel corresponding to the number of iterations. Note how the byte raster is assigned a scale using the `setAll` method. This method passes the data and sets the dimensions of the byte raster in world units. The plotting panel adjusts its gutters and scale to insure that its minimum and maximum match the byte raster's dimension.

The `RasterFrame` class in the `frames` package is a convenient wrapper for an image. The `DiffractionRasterApp` program shows this frame being used to display the diffraction of light from a two-dimensional aperture. Because computer displays have a limited intensity range and because the human eye does not respond linearly to intensity, the program plots the square root of the light intensity rather than the intensity.

## 8.2 Images and Rasters

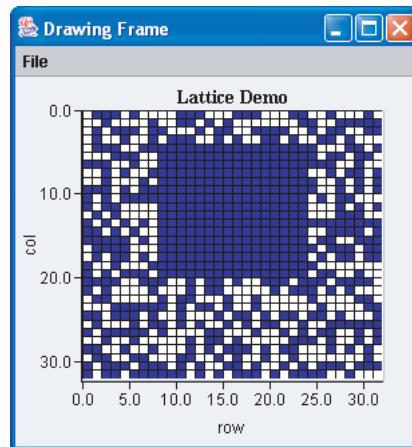
149

**LISTING 8.2** The Mandelbrot program.

```

1  package org.opensourcephysics.manual.ch08;
2  import org.opensourcephysics.display.*;
3  import org.opensourcephysics.display2d.ByteRaster;
4
5  public class MandelbrotApp implements Runnable {
6      static final int SIZE = 300; // the image size in pixels
7      PlottingPanel panel = new PlottingPanel("re", "im",
8                                         "Mandelbrot Set");
9      DrawingFrame frame = new DrawingFrame(panel);
10     ByteRaster byteRaster = new ByteRaster(SIZE, SIZE);
11     byte[][] data = new byte[SIZE][SIZE];
12     double
13         reMin = -2, reMax = 1, imMin = -1.5, imMax = 1.5;
14
15     MandelbrotApp() {
16         frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
17         panel.addDrawable(byteRaster);
18         frame.setVisible(true);
19         new Thread(this).start(); // creates and starts the thread
20     }
21
22     int iteration(double re, double im) {
23         // algorithm: take z, square it, add the number (re,im)
24         int count = 0;
25         double
26             a = 0, b = 0, temp = 0;
27         while(count<255) {
28             temp = a;
29             a = a*a-b*b+re;
30             b = 2*temp*b+im;
31             if(a*a+b*b>4.0) {
32                 break;
33             }
34             count++;
35         }
36         return(10*count); // expand the scale by ten
37     }
38
39     public void run() {
40         for(int iy = 0;iy<SIZE;iy++) {
41             double im = imMin+(imMax-imMin)*iy/(double) SIZE;
42             for(int ix = 0;ix<SIZE;ix++) {
43                 double re = reMin+(reMax-reMin)*ix/(double) SIZE;
44                 data[ix][iy] = (byte) (-128+iteration(re, im));
45             }
46             try { // yield after every row to give other threads a
47                 Thread.yield();
48             } catch(Exception e) {}
49         }
50         byteRaster.setAll(data, reMin, reMax, imMin, imMax);
51         panel.repaint();
52     }
53
54     public static void main(String[] args) {
55         new MandelbrotApp();
56     }
57 }
```

### 8.3 ■ LATTICES



**FIGURE 8.3** The  $32 \times 32$  binary lattice generated by Listing ??.

Arrays of ordinal data such as integers or bytes occur frequently in computer simulations such as the Ising model, random walks, percolation, and cellular automata such as the *Game of Life*. Open Source Physics lattice components, such as `BinaryLattice`, `CellLattice`, and `SiteLattice`, are designed to display this type of data. The `LatticeFrame` class in the frames package is a composite object that can be switched between site and cell lattice visualizations using the View menu. `LatticeFrameApp` is available on the CD and demonstrates how this frame is used.

Unlike the raster class introduced in Section 8.2, the image generated by a lattice can be resized. In other words, a  $32 \times 32$  lattice that is added to a  $320 \times 320$  drawing panel draws every element using a  $32 \times 32$  square of pixels. Lattice and raster APIs are otherwise very similar as shown in Table 8.3. The program shown in Listing 8.3 displays the  $32 \times 32$  lattice shown in Figure 8.3.

Listing 8.3 A binary lattice of random ones and zeros.

```

1 package org.opensourcephysics.manual.ch08;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.display2d.CellLattice;
4
5 public class CellLatticeApp {
6     static final int SIZE = 32;
7
8     public static void main(String[] args) {
9         PlottingPanel panel = new PlottingPanel("x", "y", "Byte
          Lattice");

```

## 8.3 Lattices

151

**TABLE 8.3** Methods common to the raster and lattice classes.**org.opensourcephysics.display2d.ByteLattice interface**

<code>getValue</code>	Gets the value of a cell or a site.
<code>setBlock</code>	Sets a block of cells or sites.
<code>setValue</code>	Sets the color palette using an array of colors.
<code>setCol</code>	Sets values within a single column.
<code>setColorPalette</code>	Sets the color palette using an array of colors.
<code>setGridLineColor</code>	Sets the grid color if the show grid option has been enabled.
<code>setIndexedColor</code>	Sets the color in a palette associated with a single value.
<code>setMinMax</code>	Sets world coordinates for the upper left and lower right corners.
<code>setRow</code>	Sets values within a single row.
<code>setShowGridLines</code>	Shows a grid if the cell size is larger than 2 pixels.
<code>showLegend</code>	Shows a legend of colors and values.

```

10 DrawingFrame frame = new DrawingFrame(panel);
11 CellLattice lattice = new CellLattice(SIZE, SIZE);
12 byte[][] data = new byte[SIZE][SIZE];
13 for(int iy = 0; iy<SIZE; iy++) {
14     for(int ix = 0; ix<SIZE; ix++) {
15         data[ix][iy] = (byte) (256*Math.random());
16     }
17 }
18 lattice.setAll(data, -1, 1, 1, -1);
19 lattice.setBlock(4, 8,
20     new byte[12][9]); // sets a block of cells to new values
21 panel.addDrawable(lattice);
22 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23 frame.setVisible(true);
24 lattice.showLegend();
25 }
26 }
```

We use `byte` arrays because they do not require much memory and provide a convenient range of 256 colors. The `ByteLattice` interface in the `2d-display` package defines a common interface that is implemented in a number of classes: `CellLattice`, `SiteLattice`, and `ByteRaster`. Classes that display floating point data do not implement this interface but implement similar methods. (See Section 8.4.)

The default byte lattice color palette shades values from blue toward red as the data value increase from  $-128$  to  $127$ . Zero is green. In order to customize the default palette, lattice classes implement the `setIndexedColor` method to set the color associated with a single value. For example, we can set the color associated

with zero to black as follows:

```
1 lattice.setIndexedColor(0, Color.black);
```

The entire color palette can be set by passing an array of colors to the `setColorPalette` and the `setGridLineColor` method.

```
1 lattice.setColorPalette(new Color[]
  {Color.blue, Color.green, Color.red, Color.white});
2 lattice.setGridLineColor(Color.black);
```

Values with undefined palette colors are rendered as black and extra palette colors are ignored. A `BinaryLattice` stores two palette values. Zero corresponds to the first palette color and one corresponds to the second palette color.

Lattice and raster classes use the first index as the *x* index and the second index as the *y* index. The [0][0] data value is drawn at the lattice x-minimum and y-minimum location in a drawing panel. Because the default x and y scales increase to the right and up, respectively, the [0][0] data value usually corresponds to the lower left hand corner of the visualization. The position of the [0][0] element can be changed by setting the lattice minimum and maximum values using either the `setAll` or the `setMinMax` methods. Because lattice classes implement the measurable interface, setting  $y_{min} > y_{max}$  not only sets the lattice minimum and maximum values but also flips the image if the axis is autoscaled.

```
1 lattice.setAll(data, -1, 1, 1, -1); // sets the data and the scale
2 lattice.setMinMax(-1, 1, 1, -1); // sets the scale
```

The `BinaryLattice` class is implemented differently from a `ByteLattice`. This class minimizes storage by creating a buffer using a Java *packed raster*. Because every *bit* of this raster corresponds to a single cell, every *byte* stores 8 cells. Although the code uses shift operators to change cell values, users need not decipher these details because lattice cells are accessed using high-level abstractions for setting and getting values.

```
1 int ix=3, iy=4;
2 int val=lattice.getValue(ix, iy); // gets a
  cell value
3 lattice.setValue(ix, iy, 1); // sets a
  cell value to one
4 lattice.setBlock(ix, iy, new int[8][4]); // zeros a block
  of cells
```

`BinaryLattice` and `ByteLattice` classes can superimpose grid lines to highlight cell boundaries. However, grid lines are not drawn if the cell size is less than 4 pixels and rasters cannot draw grid lines because their cell size is always one pixel. The visibility and color of grid lines are set using the `setShowGrid` and `setGridColor` methods.

```
1 lattice.setShowGridLines(true);
```

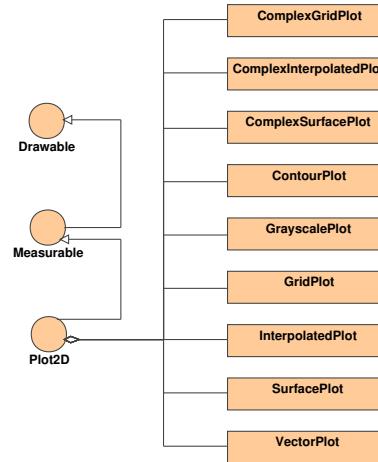
## 8.4 Plot2D Interface

153

```
2| lattice.setGridLineColor(Color.black);
```

Test programs for the lattice classes are: `BinaryLatticeApp`, `CellLatticeApp`, and `SiteLatticeApp`. These programs are not listed but are available on the CD.

## 8.4 ■ PLOT2D INTERFACE



**FIGURE 8.4** Concrete implementations of the `Plot2D` interface.

Various classes in the `display2d` package accept arrays of floating point numbers to create visualizations such as grid plots, contour plots, and 3D-surface plots. These classes implement the `Plot2D` interface as diagramed in Figure 8.4 and therefore have a similar API. To provide an overview of the API, we now show a simple example of a concrete implementation. The `GridPlot` class defines a visualization that shows the structure of a scalar field by drawing multi-colored rectangles with color representing the value of the field. Listing 8.4 uses this component to plot a gaussian scalar field,

$$U(x, y) = e^{-(x^2+y^2)}. \quad (8.3)$$

Because OSP visualizations implement the `drawable` interface, the example begins by instantiating a drawing panel and adding it to the panel. The data's world coordinates are stored when the `setAll` method is invoked with a data array followed by minima and maxima. This allows us to later invoke the `indexToX` and `indexToY` methods.

**LISTING 8.4** A scalar field plotting program.

```

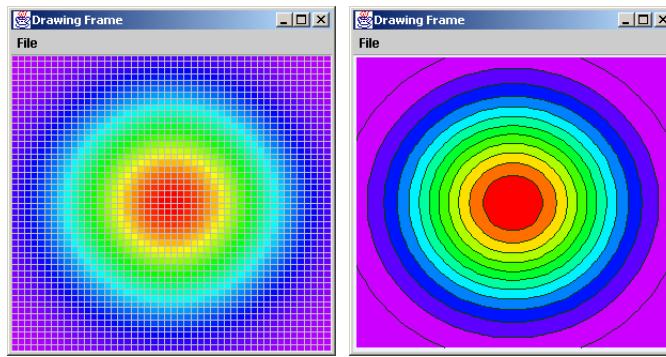
1 package org.opensourcephysics.manual.ch08;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.display2d.*;
4
5 public class GridPlotApp {
6     static final int SIZE = 32;
7
8     public static void main(String[] args) {
9         DrawingPanel plottingPanel = new PlottingPanel("x", "y",
10                                         "Grid Plot");
11         DrawingFrame frame = new DrawingFrame(plottingPanel);
12         double[][] data = new double[SIZE][SIZE];
13         GridPlot plot = new GridPlot();
14         plot.setAll(data, -1.5, 1.5, -1.5, 1.5); // sets the data
15         // and scale
16         for(int i = 0;i<SIZE;i++) { // calculate field
17             double x = plot.indexToX(i); // x coordinate at
18             this index
19             for(int j = 0;j<SIZE;j++) {
20                 double y = plot.indexToY(j); // y coordinate at
21                 this index
22                 data[i][j] = Math.exp(-(x*x+y*y)); // field value
23             }
24         }
25         plot.setAll(data); // sets the data
26         plottingPanel.addDrawable(plot);
27         frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
28         frame.setVisible(true);
29     }
30 }
```

Visualizations that implement the `plot2d` interface are designed to display arrays of floating point numbers. Each floating point array is stored in an object referred to as a *data model* that contains scale and other information. This data model can be created a number of ways. (Section 8.8 describes data models.) If a data model has already been instantiated, it can be passed to the visualization in the constructor. If a visualization is instantiated without a data model, then the tool will automatically create an appropriate model when the `setAll` method is invoked. This automatic creation of the data model requires that the program allocate the data array. The data array can be used over and over in the computation because the visualization copies the array's values into the data model. Figure 8.5 shows screen shots of four scalar field visualizations that implement the `Plot2D` interface.

Many visualizations contain a *color mapper* to convert data to colors. A gray-

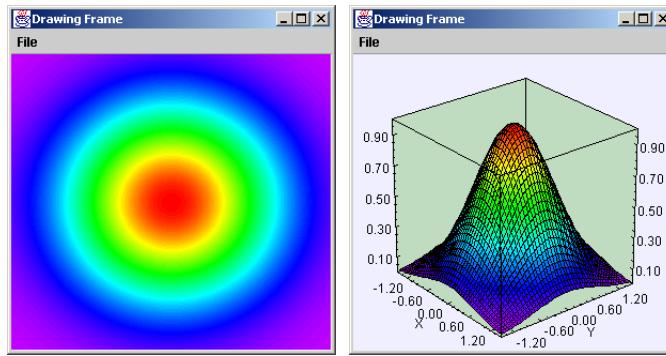
## 8.4 Plot2D Interface

155



(a) Grid plot.

(b) Contour plot.



(c) Interpolated plot.

(d) Surface plot.

**FIGURE 8.5** Visualizations of the scalar field  $U(x, y) = e^{-(x^2+y^2)}$ .

scale color mapper, for example, interprets a number as a gray level, while a dual-shade color mapper interprets data by shading colors from blue to red. The smallest and largest data values that can be translated to color are the mapper's *floor* and *ceiling* values. This range is referred to as the *z-range*. Values below the floor or above the ceiling are mapped to special colors. Setting the autoscale property to true will adjust the z-range to match the data. If this property is false, the user must supply a valid range. These display options are set using the `setAutoscaleZ` and `setFloorCeilColor` methods as follows:

```

1 double floor=-1, ceil=1;
2 Color floorColor=Color.black, ceilColor=Color.white;
3 // plot is a visualization that implements Plot2D
4 plot.setPaletteType (ColorMapper.DUALSHADE);

```

**TABLE 8.4** The Plot2D interface defines methods common to 2d-visualizations that use the GridData storage model.

<b>org.opensourcephysics.display2d.Plot2D interface</b>	
indexToX	Gets the x world coordinate for the given index. Similar method for y.
xToIndex	Gets closest index from the given x world coordinate. Similar method for y.
setAll	Sets the data and scales the data model. The grid is resized as needed.
setAutoscaleZ	Sets autoscaling parameters for the amplitude.
setColorPalette	Sets the color palette using an array of colors.
setFloorCeilColor	Sets the floor and ceiling colors.
setGridData	Sets the data storage object.
setGridLineColor	Sets the grid line or contour line color.
setIndexes	Specifies the ordering of data, such as amplitude and phase, when using multi-component data.
setPaletteType	Sets a predefined color palette.
setShowGridLines	Draws the grid lines or contour lines when true.
setVisible	Sets the visibility of the plot.
showLegend	Shows a legend of colors and values.
update	Updates the object's state. Update should be invoked after the grid's data or scale is changed.

```

5 plot.setAutoscaleZ(false, floor, ceil);
6 plot.setFloorCeilColor(floorColor, ceilColor);
7 plot.showLegend();

```

## 8.5 ■ SCALAR FIELDS

### Grid Plot

The GridPlot shown in Figure 8.5(a) draws a rectangle centered at the measurement point using a color that is determined by the value of the field. Listing 8.4 shows how a grid plot is created and used. This plot is similar to a lattice except that it uses floating point numbers rather than bytes to store data. The visualization copies the array into a data model that contains additional information such as a scale when the `setAll` method is invoked.

The grid plot's color mapper supports various color palettes. The DUALSHADE palette, for example, shades pixels from blue toward red as the values change from floor to ceiling. A GRayscale palette shades pixels from black to white.

## 8.5 Scalar Fields

157

```
1 plot.setPaletteType(ColorMapper.DUALSHADE);
```

Consult the `ColorMapper` class for other color palettes.

Custom palettes are created by passing an array of colors to the `setColorPalette` method. For example, the following code fragment creates a three color palette that divides the floor to ceiling range into three equal intervals. Note that five colors may appear on-screen because data values below the floor or above the ceiling are colored using special floor and ceiling colors.

```
1 plot.setColorPalette(new
  color []={ Color.RED, Color.GREEN, Color.BLUE});
```

It is easy to use a grid plot in place of a lattice. However, because a grid plot is designed to show a field of floating point numbers, it uses more memory than a lattice and it is difficult to check for equality of values due to roundoff. In general, a lattice is preferred when values are countable (ordinal).

An advantage of `GridPlot` is that it is very fast as shown in Table 8.2. We sometimes use it as an alternative to more complicated visualizations, such as contour plots, during an animation. Because a grid plot displays the structure of the underlying data, it is also useful for debugging. The grid plot clearly shows if an algorithm fails at a point because values are not interpolated or smoothed.

### Contour Plot

The `ContourPlot` class draws a contour plot of a two-dimensional scalar field.<sup>3</sup> In fact, the only change required to Listing 8.4 in order to produce Figure 8.5(b) is to change the type of component that is being constructed.

```
1 plot = new ContourPlot();
```

It is, of course, possible to customize a plot and these customizations will depend on the type of plot. The default number of contour lines is 12, but this number can be changed. Note that because the shape of a contour line is sensitive to a contour's *z*-level, it is often desirable not to autoscale the *z* axis. This is especially true for scalar fields that contain singularities, such as charged particles in electrostatics, because even a small change in the location of a grid point can produce large changes in the scalar field's values at the grid points. The following code fragment specifies five fixed contour levels at -2, -1, 0, 1, and 2.

```
1 // plot must be of type ContourPlot to set number of levels
2 plot = setNumberOfLevels(5);
3 plot.setAutoscaleZ(false, -2, 2); // disable autoscaling
```

Complex visualizations, such as contour plots, are computationally intensive and may not be well-suited for real-time animations. We can, however, use a `DrawableBuffer` introduced in Section 7.5. The buffer contains the contour plot

<sup>3</sup> `ContourPlot` and `SurfacePlot` are based on the Surface Plotter program by Yanto Suryono.

that provides the background image for the panel. The entire panel is repainted when the interactive circle is being dragged, but this is fast because repainting merely copies the background image and paints a circle. If the contour data were to change, we require that the program invoke the buffer's `invalidateImage` method to generate a new background. `BufferedContourApp` demonstrates this buffering technique and is available on the CD.

### Interpolated Plot

An interpolated plot is similar to a grid plot except that color values are interpolated at every pixel. This interpolation blurs the grid's cell boundaries. The only change required to Listing 8.4 to produce the interpolated plot shown in Figure 8.5(c) is to change the component being constructed.

```
1 plot= new InterpolatedPlot();
```

Because interpolation from the data grid is performed at every pixel, the time required to render an interpolated plot is insensitive to the number of grid points. Rendering time is, however, roughly proportional to the number of pixels in the drawing panel because an interpolation must be performed at every pixel.

### Surface Plot

A `SurfacePlot` shows a scalar field using a three-dimensional color-coded mesh. The height ( $z$  value) is proportional to the field magnitude. Again, the only change required to Listing 8.4 to produce the surface plot shown in Figure 8.5(d) is to change the component being constructed. Viewing angles and other display options can be set after the surface plot is instantiated.

```
1 surfacePlot= new SurfacePlot();
2 surfacePlot.setRotationAngle(125); // default is 125 degrees
3 surfacePlot.setElevationAngle(10); // default is 10 degrees
4 surfacePlot.setDistance(200.0); // default is 200 units
5 surfacePlot.set2DScaling(8.0); // default is 8 pixels per unit
```

Because it is desirable to be able to adjust a surface plot's viewing angle using a mouse, we have defined a `SurfacePlotMouseController` that implements the `Java MouseListener` and `MouseMotionListener` interfaces. This class handles mouse events to rotate, zoom, and translate the surface plot using drag, shift-drag, and control-drag mouse actions, respectively. Keyboard events are also supported. A complete surface plot test program is shown in Listing 8.5. Note that the `SurfacePlotMouseController` is added to the panel using standard Java `addMouseListener` and `addMouseMotionListener` methods.

Listing 8.5 Surface plot test program.

```
1 package org.opensourcephysics.manual.ch08;
2 import org.opensourcephysics.display.*;
```

## 8.6 Vector Fields

159

```

3 import org.opensourcephysics.display2d.*;
4
5 public class SurfacePlotApp {
6     static final int SIZE = 32;
7
8     public static void main(String[] args) {
9         DrawingPanel drawingPanel = new DrawingPanel();
10        DrawingFrame frame = new DrawingFrame(drawingPanel);
11        double[][] data = new double[SIZE][SIZE];
12        SurfacePlot plot = new SurfacePlot();
13        plot.setAll(data, -1.5, 1.5, -1.5, 1.5); // sets the data
14        and scale
15        for(int i = 0;i<SIZE;i++) {           // calculate field
16            double x = plot.indexToX(i);
17            for(int j = 0;j<SIZE;j++) {
18                double y = plot.indexToY(j);
19                data[i][j] = Math.exp(-(x*x+y*y)); // magnitude
20            }
21        }
22        plot.setAll(data); // sets the data
23        drawingPanel.addDrawable(plot);
24        SurfacePlotMouseController mouseController = new
25            SurfacePlotMouseController(drawingPanel,
26                                         plot);
27        drawingPanel.addMouseListener(mouseController);
28        drawingPanel.addMouseMotionListener(mouseController);
29        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30        frame.setVisible(true);
31    }
32 }
```

## 8.6 ■ VECTOR FIELDS

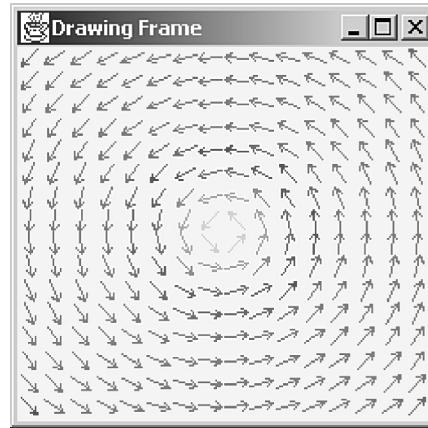
The VectorPlot class shows a vector field. To use this class we instantiate a multi-dimensional array to store components of the vector. The first array index indicates the component, the second index indicates the column or  $x$  position, and the third index indicates the row or  $y$  position. The vectors in the visualization are set by passing the array to the frame using the `setAll` method.

Because a Java multidimensional array is an array of arrays, it is easy to reference sub-arrays which contain data for a single component.

```

1 double[][][] data = new double[2][32][32];
2 double[][] xdata = data[0];
3 double[][] ydata = data[1];
```

Note that the `xdata` and `ydata` sub-arrays are not allocated. These identifiers are merely references (pointers) to memory that is allocated in the first statement.



**FIGURE 8.6** A vector plot with arrows of equal length. Running the `VectorPlotApp` program shows how color is used to show magnitude.

The `VectorPlotApp` program displays a circulating vector field whose magnitude increases in direct proportion to the distance from the origin.

$$\tilde{\mathbf{A}} = -\mathbf{r} \sin \theta \hat{\mathbf{x}} + \mathbf{r} \cos \theta \hat{\mathbf{y}} = -y \hat{\mathbf{x}} + x \hat{\mathbf{y}} \quad (8.4)$$

Because we have found that using color rather than length to represent field strength produces a more effective representation of magnitude over a wider range of values, the `VectorPlot` class uses color-coded arrows. These arrows have a fixed length that is chosen to fill the viewing area. Section 8.8 describes how to program a data model to obtain the traditional association between arrow length and field magnitude.

Listing 8.6 Vector field test program.

```

1 package org.opensourcephysics.manual.ch08;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.display2d.*;
4
5 public class VectorPlotApp {
6     static final int SIZE = 24;
7
8     public static void main(String[] args) {
9         DrawingPanel drawingPanel = new DrawingPanel();
10        DrawingFrame frame = new DrawingFrame(drawingPanel);
11        drawingPanel.setSquareAspect(true);
12        double[][][] data = new double[2][SIZE][SIZE];
13        VectorPlot plot = new VectorPlot();
14        plot.setAll(data, -1, 1, -1, 1);
15        for(int i = 0; i < SIZE; i++) {

```

## 8.7 Complex Fields

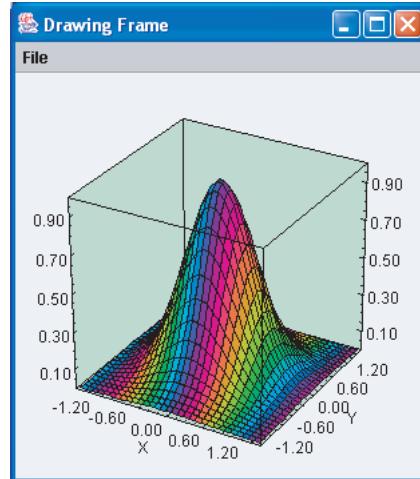
161

```

16     double x = plot.indexToX(i); // the x location
17     for(int j = 0;j<SIZE;j++) {
18         double y = plot.indexToY(j); // the y location
19         data[0][i][j] = -y;
20         data[1][i][j] = x;
21     }
22 }
23 plot.setAll(data);
24 drawingPanel.addDrawable(plot);
25 frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
26 frame.setVisible(true);
27 plot.showLegend();
28 }
29 }
```

## 8.7 ■ COMPLEX FIELDS

Fields of complex numbers, as shown in Figure 8.7, occur frequently in electromagnetism and in quantum mechanics. Because real and imaginary numbers are similar to vector components, complex scalar fields are stored similarly to vector fields. We again use an array with three indexes to store a complex field.



**FIGURE 8.7** A complex scalar field,  $\psi(x, y) = e^{-(x^2+y^2)}e^{i5x}$ . Phase is shown as color.

ComplexGridPlot and ComplexInterpolatedPlot use brightness to show a field's magnitude while ComplexSurfacePlot uses height. All three complex

field visualizations use color to show phase starting with blue for positive real, red for positive imaginary, yellow for negative real, and green for negative imaginary. Listing 8.7 shows how the complex surface plot is used to display a gaussian quantum wave function with a phase modulation (momentum boost) in the  $x$  direction.

$$\psi(x, y) = e^{-(x^2+y^2)}e^{i5x}. \quad (8.5)$$

Listing 8.7 Complex scalar field test program.

```

1 package org.opensourcephysics.manual.ch08;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.display2d.*;
4
5 public class ComplexSurfacePlotApp {
6     final static int SIZE = 32;
7
8     public static void main(String[] args) {
9         DrawingPanel drawingPanel = new DrawingPanel();
10        drawingPanel.setShowCoordinates(false);
11        DrawingFrame frame = new DrawingFrame(drawingPanel);
12        double[][][] data = new double[2][32][32];
13        double[][] xdata = data[0];
14        double[][] ydata = data[1];
15        ComplexSurfacePlot plot = new ComplexSurfacePlot();
16        plot.setAll(data, -1.5, 1.5, -1.5, 1.5);
17        for(int i = 0;i<SIZE;i++) {
18            double x = plot.indexToX(i); // the x location
19            for(int j = 0;j<SIZE;j++) {
20                double y = plot.indexToY(j); // the y location
21                double amp = Math.exp(-2*(x*x+y*y)); // magnitude
22                xdata[i][j] = amp*Math.cos(5*x); // real component
23                ydata[i][j] = amp*Math.sin(5*x); // imaginary
24                component
25            }
26        }
27        plot.setAutoscaleZ(false, 0, 1);
28        plot.setAll(data);
29        drawingPanel.addDrawable(plot);
30        drawingPanel.repaint();
31        SurfacePlotMouseController mouseController = new
32            SurfacePlotMouseController(drawingPanel,
33                plot);
34        drawingPanel.addMouseListener(mouseController);
35        drawingPanel.addMouseMotionListener(mouseController);
36        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
37        frame.setVisible(true);
38    }

```

37

}

**8.8 ■ DATA MODELS**

Storing data in an array of numbers is simple but lacks the flexibility of a data model that stores an array along with additional information such as a scale and color. The `GridData` interface (see Table 8.5) in the `display2d` package overcomes this limitation by modeling an arbitrary number of measurements (components) on a two-dimensional grid using a variety of different array configurations. Using a data model allows us to separate the data from visualizations such as `GridPlot`, `ContourPlot`, `SurfacePlot`, and `VectorPlot`.

**TABLE 8.5** Methods in the `GridData` interface.

<b><code>org.opensourcephysics.display2d.GridData</code></b>	
<code>getBottom</code>	Gets the world units of the last row of data.
<code>getData</code>	Gets the multidimensional array containing the raw data.
<code>getDx</code>	Gets the change in x between data rows.
<code>getDy</code>	Gets the change in y between data columns.
<code>getLeft</code>	Gets the world units of the first column of data.
<code>getNx</code>	Gets the number of entries along the x-ordinate.
<code>getNy</code>	Gets the number of entries along the y-ordinate.
<code>getRight</code>	Gets the world units of the last column of data.
<code>getTop</code>	Gets the world units of the first row of data.
<code>getZRange</code>	Gets the maximum and minimum values for a sample or component.
<code>indexToX</code>	Gets the x world coordinate for the given index. Similar method for y.
<code>interpolate</code>	Estimates a sample at an untabulated point, (x,y), using bilinear interpolation of grid point data.
<code>setScale</code>	Sets world coordinates for the grid's rows and columns assuming that the data values are for the edge of a cell.
<code>setCellScale</code>	Sets world coordinates for the grid's rows and columns by assuming that the data values are for the center of a cell.
<code>xToIndex</code>	Gets closest index from the given x world coordinate. Similar method for y.

There are many ways to encapsulate and store field data using a grid. We can allocate memory to store both the field and the coordinates of each grid point or we can store only the field and compute coordinates using the grid's stored minimum and maximum values. Likewise, we can store just the vector components or we can store components and computed values such as magnitude or phase, etc. The `GridData` interface in the `display2d` package is designed to support these types of options. The `ArrayData`, `PointData`, and `FlatData` classes are concrete implementations of the `GridData` interface.

The `VectorPlot` class described in Section 8.6 uses three values from the data model to draw an arrow at a grid point: a color parameter, the  $x$  component of the arrow, and the  $y$  component of the arrow. The `ArrayData` class stores these values using three  $n \times m$  sub-arrays. These sub-arrays are created within `ArrayData` by allocated a multi-dimensional array with three indices.

```
1 // values for n by m vector field using ArrayData
2 double[][][] data = new double[3][n][m];
```

Each sub-array can be accessed by de-referencing the first index.

```
1 double[][] arrow_color = new double[0];
2 double[][] arrow_x = new double[1];
3 double[][] arrow_y = new double[2];
```

Values at a grid point are obtained by accessing each sub-array.

```
1 // values at i, j grid point
2 double x:                                         // x coordinate must be
   computed
3 double y;                                         // y coordinate must be
   computed
4 double color = arrow_color[i][j]; // same as data[0][i][j]
5 double x_length = arrow_x[i][j]; // same as data[1][i][j]
6 double y_length = arrow_y[i][j]; // same as data[2][i][j]
```

The `PointData` class stores a vector field differently. This class models a grid point as a one-dimensional sub-array that includes the coordinate of the grid point as the first two elements of this array. Values at a grid point are obtained by de-referencing the data array's first two indices.

```
1 // values for n by m vector field using PointData
2 double[][][] data = new double[n][m][5];
3 // values at i, j grid point
4 double[] grid_point = data[i][j];
5 double x = grid_point[0];                         // same as data[i][j][0]
6 double y = grid_point[1];                         // same as data[i][j][1]
7 double color = grid_point[2];                     // same as data[i][j][2]
8 double x_length = grid_point[3]; // same as data[i][j][3]
9 double y_length = grid_point[4]; // same as data[i][j][4]
```

## 8.8 Data Models

165

A point's  $x$  and  $y$  values are computed using methods in the `PointData` class. These coordinate values are usually not changed outside the class but may be used by a computation that has access to the data array.

The `FlatData` class does away with multi-dimensional arrays and stores all values in a single one dimensional array. These values are stored in *row major* order.

```

1 // values for n by m vector field using FlatData
2 double[] data = new double[n*m*3];
3 // values at i,j grid point
4 double x; // x coordinate must be computed
5 double y; // y coordinate must be computed
6 double color=data[i*n+m];
7 double x_length=data[i*n+m+1];
8 double y_length=data[i*n+m+2];

```

The data model is usually passed to a visualization in its constructor but it can later be changed using the `setGridData` method. Note that the same `GridData` object can be used by more than one visualization.

Consider again the vector field that was used in Listing 8.7. This field is sampled on a  $32 \times 32$  grid spanning a region of physical space from  $x_{\min} < x < x_{\max}$  and  $y_{\min} < y < y_{\max}$ . A `GridPointData` model for this field is instantiated and used as follows:

```

1 GridData griddata = GridPointData(32,32,3); // 3 samples at
      every grid point
2 griddata.setScale(xmin,xmax,ymin,ymax); // compute
      coordinates for array
3 VectorPlot plot= new VectorPlot(griddata);

```

We break encapsulation and retrieve a reference to the storage object's data array using the `getData` method for computational speed:

```
1 double[][][] data = griddata.getData();
```

We must, of course, know the storage model that is being used to properly access the array. Because we know that we are using a `GridPointData` object, we can compute the vector field using the stored  $x$  and  $y$  values.

```

1 for(int i = 0, nx = data.length; i<nx; i++) {
2     for(int j = 0, ny = data[0].length; j<ny; j++) {
3         double x = data[i][j][0]; // grid point x location
4         double y = data[i][j][1]; // grid point y location
5         double r = Math.sqrt(x*x+y*y);
6         data[i][j][2] = r; // magnitude
           determines color
7         data[i][j][3] = (r==0)? 0 : -y/r; // horizontal arrow
           component
8         data[i][j][4] = (r==0)? 0 : x/r; // vertical arrow
           component

```

```

9 } }
10 }
11 VectorPlot plot = new VectorPlot(griddata);
12 drawingPanel.addDrawable(plot);

```

The color parameter is set equal to the field's magnitude and the arrow's  $x$  and  $y$  components are set equal to the direction cosines so that the `VectorPlot` draws arrows of equal length whose color is indicative of field strength. The following code fragment shows how to produce a visualization that uses field magnitude to determine the arrow length.

```

1 data[i][j][2] = 1;      // constant color
2 data[i][j][3] = -y;    // horizontal arrow component
3 data[i][j][4] = x;     // vertical arrow component

```

Because a complex field is similar to a vector field, complex-field visualizations also use three values from the data model: visual magnitude (intensity), a real component, and an imaginary component. Although these values are clearly redundant from a physics point of view, a third parameter allows us to use a non-linear intensity scale in the visualization in order to see detail in regions where the complex field is weak.

Scalar fields require only a single value from a data model. If we wish to store a scalar field using a  $32 \times 32$  grid that bounds a region two world units on a side, we can instantiate either a `GridPointData` object

```

1 GridData griddata=new GridPointData(32,32,1);
2 griddata.setScale(-1,1,-1,1); //world units

```

or an `ArrayData` object

```

1 GridData griddata=new ArrayData(32,32,1);
2 griddata.setScale(-1,1,-1,1); // world units

```

or a `FlatData` object

```

1 GridData griddata=new FlatData(32,32,1);
2 griddata.setScale(-1,1,-1,1); // world units

```

The number of data components is the last argument in the above constructors. We then create a visualization by instantiating a visualization and passing it the grid data.

```

1 ContourPlot contour = new ContourPlot(griddata);
2 DrawingPanel panel = new DrawingPanel();
3 panel.addDrawable(contour);

```

Because any data model can be used with a 2D visualization, we can choose a data model that fits the numerical algorithm and the physical model.



```

10    drawingPanel.setSquareAspect(false);
11    DrawingFrame frame = new DrawingFrame(drawingPanel);
12    GridPointData pointdata = new GridPointData(16, 16,
13                                              1); // scalar field
14    pointdata.setScale(-1, 1, -1, 1);
15    double[][][] data = pointdata.getData();
16    for(int i = 0, row = data.length; i < row; i++) {
17        for(int j = 0, col = data[0].length; j < col; j++) {
18            double x = data[i][j][0]; // the x location
19            double y = data[i][j][1]; // the y location
20            data[i][j][2] = y*x; // magnitude
21        }
22    }
23    InterpolatedPlot plot = new InterpolatedPlot(pointdata);
24    GridPointData vecData2d = Util2D.gradient(pointdata, 1);
25    VectorPlot vectorplot = new VectorPlot(vecData2d);
26    vectorplot.setPaletteType(VectorColorMapper.BLACK);
27    drawingPanel.addDrawable(plot); // add scalar field
28    drawingPanel.addDrawable(vectorplot); // overlay vector field
29    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30    frame.setVisible(true);
31}
32}

```

The ComplexContourPlot class in the plot2d package uses inheritance and composition to define an object that overlays contour lines to a complex interpolated plot. The interpolated plot shows the phase of the complex scalar field and the contour plot overlays lines showing the field's magnitude.

## 8.10 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch08` package.

### **BinaryLatticeAnimationApp**

`BinaryLatticeAnimationApp` test a binary lattice by rapidly assigning random values to each cell using an animation thread.

### **BinaryLatticeApp**

`BinaryLatticeApp` creates a  $32 \times 32$  lattice of random values and then sets a  $16 \times 16$  block of cells to zero. This program is described in Section 8.5.

### **CellLatticeApp**

`CellLatticeApp` tests the cell lattice by displaying the default color palette.

**BufferedContourApp**

BufferedContourApp superimposes a dragable circle on a static background image of a contour plot. This program is described in Section ??.

**ByteRasterApp**

ByteRasterApp tests the byte raster class by creating random data in a  $256 \times 256$  raster.

**CalcGridPlotApp**

CalcGridPlotApp tests a grid plot by allowing the user to enter an arbitrary function,  $f(x, y)$ , using a simple user interface. Other programs that plot scalar fields using arbitrary functions are CalcContourPlotApp and CalcSurfacePlotApp.

**GridPlotApp**

GridPlotApp displays a gaussian scalar field using a grid plot. This program is described in Section 8.8.

**ComplexSurfacePlotApp**

ComplexSurfacePlotApp displays a complex scalar field using a complex surface plot. This program is described in Section 8.7. Other programs that display complex scalar fields are available: ComplexGridPlotApp, ComplexContourApp, and ComplexInterpolatedApp.

**ContourPlotApp**

ContourPlotApp creates a contour plot of a scalar field,  $U(x, y) = x * y$ .

**DiffractionRasterApp**

DiffractionRasterApp demonstrates how to use RasterFrame by computing a single slit diffraction pattern.

**DLAApp**

DLAApp models diffusion limited aggregation using a byte raster.

**GameOfLifeApp**

GameOfLifeApp uses a binary lattice to model the game of life. Right-click on the lattice to toggle life on and off.

**GaussianSurfacePlotApp**

GaussianSurfacePlotApp displays gaussian scalar field using a surface plot. This program is described in Section 8.5. Other programs that display examples

of scalar fields are available: `GaussianGridPlotApp`, `GaussianContourApp`, and `GaussianInterpolatedApp`.

### **GrayscaleApp**

`GrayscaleApp` displays a scalar field using a grayscale plot. This plot looks similar to a grid plot with a gray scale color palette. However, it uses a different rendering model.

### **MandelbrotApp**

`MandelbrotApp` displays the Mandelbrot set using a byte raster. This program is described in Section 8.2.

### **RandomWalkApp**

`RandomWalkApp` uses a binary lattice to simulate a random walk.

### **SiteLatticeApp**

`SiteLatticeApp` tests the site lattice class by displaying the default color palette.

### **VectorPlotApp**

`VectorPlotApp` displays a vector field. This program is described in Section 8.6. The `CalcVectorPlotApp` program allows the user to enter an arbitrary vector field,  $\vec{A}$ , using a simple user interface.

## CHAPTER

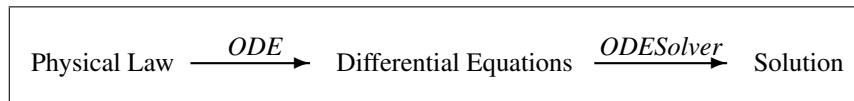
## 9

## Differential Equations and Dynamics

©2005 by Wolfgang Christian

We describe how to solve dynamical problems using systems of first-order differential equations. Ordinary differential equation, ODE, solvers are defined in the `org.opensourcephysics.numerics` package along with other analysis tools.

## 9.1 ■ OVERVIEW



**FIGURE 9.1** Differential equations can be solved using the *ODE* interface to define the physics and the *ODESolver* interface to define the numerical algorithm.

Differential equations, such as Newton's second law, describe the relationship between an unknown function and its derivatives. For example, we can compute the velocity  $v(t)$  of a particle falling near Earth as a function of time as follows:

$$\frac{dv}{dt} = -g - bv \quad (9.1)$$

where  $g = 9.8 \text{ m/s}^2$  is the constant acceleration of gravity. This equation is referred to as a *rate equation* because it expresses the rate of change of the variable as a function of variables and parameters. The quantity being differentiated (velocity) is called the *dependent variable* and the quantity with respect to which it is differentiated (time) is called the *independent variable*. Equation 9.1 is a *first-order* equation because the highest derivative is a first derivative.

Equation 9.1 has the following general form:

$$\dot{x} \equiv \frac{dx}{dt} = f(x, t). \quad (9.2)$$

We would like to solve Equation 9.2 to find  $x(t)$  for  $0 \leq t \leq t_{\text{final}}$ . The aim of this chapter is to approximate this solution by finding the value of  $x^{i+1}$  at time  $t_{i+1} = t_i + \Delta t$  given the value of  $x^i$  at time  $t_i$ . We start at  $x^0 = x(0)$  and obtain

$x^1 = x(\Delta t)$ . Once  $x^1$  is obtained, the process is repeated to find  $x^2$  and so on until the final time,  $t_{\text{final}}$ . Note that we use a superscript to specify the value of  $x$  at a particular time-step and so that we can later use a subscript to specify variables such as a position or velocity component using the generic  $x_i$ .

In general, Newton's second law for particle motion in one dimension is a second order differential equation that takes the form

$$\frac{d^2x}{dt^2} = F(x, v, t)/m. \quad (9.3)$$

This equation can be converted to two first order equations by considering both position  $x$  and velocity  $v$  to be unknown functions of time:

$$\dot{x} = v \quad (9.4a)$$

$$\dot{v} = F(x, v, t)/m. \quad (9.4b)$$

The initial position  $x^0$  and initial velocity  $v^0$  are advanced by  $\Delta t$  to obtain  $x^1$  and  $v^1$ , respectively. The process of defining extra variables such as  $v$  to reduce the order of a differential equation (but increase the number of equations) can be extended to higher order derivatives as needed. Because higher order differential equations can be converted into a system of first-order equations, we need only consider general methods of solving systems of first order equations.

Additional particles or additional spacial dimensions add additional equations. In general, a system of particles results in a system of first-order ordinary differential equations with an independent time  $t$  variable and with  $N$  dependent variables that can be written as:

$$\dot{x}_0 = f_0(x_0, x_1, \dots, x_{N-1}, t) \quad (9.5a)$$

$$\dot{x}_1 = f_1(x_0, x_1, \dots, x_{N-1}, t) \quad (9.5b)$$

$$\vdots$$

$$\dot{x}_{N-1} = f_N(x_0, x_1, \dots, x_{N-1}, t). \quad (9.5c)$$

This system is said to be autonomous if the functions  $f_i$  do not explicitly depend on the independent variable. If we think of the variables  $x_i$  as components of a state vector  $\mathbf{x}$ , this system can be written compactly as

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}). \quad (9.6)$$

The variable that is used to take the derivative can be any independent parameter but is often the time  $t$ . In order to treat real-world problems where the functions  $\mathbf{f}(\mathbf{x})$  are often explicit functions of the time, we treat the time as an additional variable and add the trivial rate equation  $dt/dt = 1$ . Our convention is to list the independent variable last in the state array. We often do this even for autonomous equations because there is little additional computation and because

## 9.2 ODE Interface

173

it allows us to automatically track the independent variable in the state array when using adaptive step size algorithms as described in Section 9.5.

In classical (newtonian) physics the position and velocity (momentum) space that we have defined is called *phase space*. The trajectory of a particle or a system of particles through phase space is completely determined if we know the initial state and an expression for the rate (force). Consider the equation of motion in two dimensions for a particle falling under the influence of gravity with a constant acceleration  $g = 9.8 \text{ m/s}^2$ . This equation can be written as:

$$\dot{x} = v_x \quad (9.7\text{a})$$

$$\dot{v}_x = 0 \quad (9.7\text{b})$$

$$\dot{y} = v_y \quad (9.7\text{c})$$

$$\dot{v}_y = -9.8 \quad (9.7\text{d})$$

$$\dot{t} = 1. \quad (9.7\text{e})$$

Another common example is the equation of motion for a driven one-dimensional harmonic oscillator with mass  $m$  and spring constant  $k$ . This equation can be written as:

$$\dot{x} = v \quad (9.8\text{a})$$

$$\dot{v} = -\frac{k}{m}x + A \sin \omega_0 t \quad (9.8\text{b})$$

$$\dot{t} = 1. \quad (9.8\text{c})$$

And finally, the equation of motion for a particle in orbit about a gravitational center of attraction  $\mathbf{F}_g(\mathbf{r}) = -(1/r^2)\hat{\mathbf{r}}$  can be written in dimensionless units as:

$$\dot{x} = v_x \quad (9.9\text{a})$$

$$\dot{v}_x = -\frac{x}{r^3} \quad (9.9\text{b})$$

$$\dot{y} = v_y \quad (9.9\text{c})$$

$$\dot{v}_y = -\frac{y}{r^3} \quad (9.9\text{d})$$

$$\dot{t} = 1 \quad (9.9\text{e})$$

where we have used  $\cos \theta = x/r$  and  $\sin \theta = y/r$  to resolve the unit vector  $\hat{\mathbf{r}}$  into its x-y components.

We will use the examples above to test numerical algorithms in the following sections.

## 9.2 ■ ODE INTERFACE

**Listing 9.1** The ODE interface is used to solve systems of first order differential equations.

```

1 package org.opensourcephysics.numerics;
2 public interface ODE {
3     public double[] getState();
4     public void getRate(double[] state, double[] rate);
5 }
```

The ordinary differential equation, ODE, interface defined in the numerics package enables us to encapsulate Equation 9.6 in a Java class. This interface contains two methods, `getState` and `getRate`, as shown in Listing 9.1. The `getState` method returns a state array  $(x_0, x_1, \dots, x_{n-1})$  where the variables represent either position or velocity. The `getRate` method evaluates the derivatives using the given state array and stores the result in the given rate array,  $(\dot{x}_0, \dot{x}_1, \dots, \dot{x}_{n-1})$ . Because most numerical algorithms evaluate the rate multiple times as they advance the system by  $\Delta t$ , the given state is usually not the current state of object.

A Java class that implements the ODE interface for the driven harmonic oscillator system (Equation 9.8) is shown in Listing 9.2. The `Falling` class uses the ODE interface to model a two-dimensional falling particle. This class is available in the `ch09` package on the CD but is not shown here because it is similar to Listing 9.2.

A convention that we find useful is to place the velocity rate immediately after

**LISTING 9.2** An implementation of the ODE interface that models the driven simple harmonic oscillator.

```

1 package org.opensourcephysics.manual.ch09;
2 import org.opensourcephysics.numerics.ODE;
3
4 public class SHO implements ODE {
5     // state array contains =[x, vx, t]
6     double[] state = new double[] {5.0, 0.0, 0.0};
7     double
8         omega = 1, amp = 1.0;
9     double
10        k = 1, m = 1.0;
11
12    public double[] getState() {
13        return state;
14    }
15
16    public void getRate(double[] state, double[] rate) {
17        rate[0] = state[1];
18        rate[1] = -k/m*state[0]+amp*Math.sin(omega*state[2]);
19        rate[2] = 1;
20    }
21 }
```

## 9.3 ODE Solver Interface

175

the position rate. For example, the dynamic variables for two particles should be stored in an array that is ordered as follows  $[x_1, v_{x1}, y_1, v_{y1}, x_2, v_{x2}, y_2, v_{y2}, t]$ . This ordering enables us to efficiently code certain numerical algorithms (such as the Verlet method) because the differential equation solver can assume that a velocity rate follows every position rate in the state array. Consult the documentation or the solver's source code to determine if a particular variable ordering is necessary.

## 9.3 ■ ODE SOLVER INTERFACE

**LISTING 9.3** The `ODESolver` interface defines numerical algorithms to solve differential equations.

```

1  public interface ODESolver {
2      public void initialize(double stepSize);
3      public double step(); // advances the state and returns the
4          step size
5      public void setStepSize(double stepSize);
6      public double getStepSize();
}
```

There are many possible algorithms to advance a system of first-order differential equations from an initial state to a final state. The `ODESolver` interface defines a set of methods that enables us to define algorithms to solve differential equations. These equations are implemented in another object that implements the ODE interface.

An ODE solver's `initialize` method sets the initial step size and allocates arrays to store temporary values. This method should be called after the ODE object has been created or if the number of equations within the ODE object changes. Because adaptive algorithms are free to change the step size, we provide the `setStepSize` and `getStepSize` methods to set and read the step size parameter. The `setStepSize` method does not allocate or initialize arrays within the solver.

Differential equations can be solved by creating a solver and repeatedly invoking the solver's `step` method. Consider again the equation of motion for the harmonic oscillator that is coded in Listing 9.2. These equations are solved in the `SHOApp` program shown in Listing 9.4. The important point to realize is that we simply need to instantiate a different solver to switch numerical methods. Run this example with various `ODESolvers` and note the different results. Note in particular that amplitude of oscillation increases when using the `Euler` solver. The explicit Euler method is often numerically unstable even with small values of the step size  $\Delta t$ . It is useful as a pedagogic tool, for debugging, and to perform the first step in numeric methods that are not self starting.

Listing 9.4 Driven harmonic oscillator program.

```

1 package org.opensourcephysics.manual.ch09;
2 import org.opensourcephysics.numerics.*;
3
4 public class SHOApp {
5     public static void main(String[] args) {
6         double time = 10; // solution range
7         double dt = 0.1; // ode step size
8         ODE ode = new SHO();
9         ODESolver ode_solver = new RK45(ode);
10        ode_solver.initialize(dt);
11        double[] state = ode.getState();
12        while(time > 0) {
13            String xStr = "x = "+state[0];
14            String vStr = " v = "+state[1];
15            String tStr = " t = "+state[2];
16            System.out.print(xStr+vStr+tStr+"\n");
17            time -= ode_solver.step();
18        }
19    }
20}

```

In summary, an ODE solver's step method advances the state of the oscillator by the time step,  $\Delta t$ , using a numeric method. Listing 9.5 in the next section shows the details of how the Euler ODE solver is written. It is, of course, up to the programmer to determine if the chosen ODE solver is accurate and stable when applied to the given problem. Section 9.4 describes various ODE solvers that are implemented in the numerics package.

## 9.4 ■ ALGORITHMS

The simplest method for solving ODEs is Euler's method, Equation 9.10.

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \mathbf{f}(\mathbf{x}^i)\Delta t \quad (9.10)$$

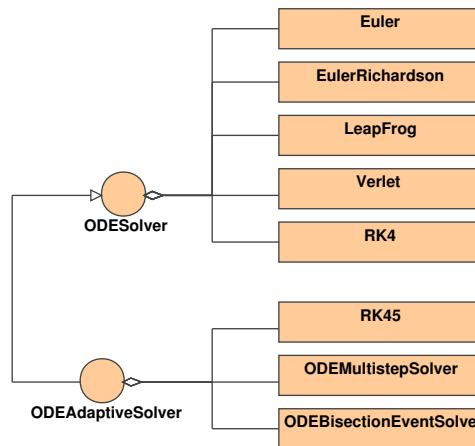
This algorithm is coded in Listing 9.5. It starts with the current value of the  $n$ -th variable at the  $i$ -th step,  $x_n^i$ , and assumes that the rate of change is constant over the interval  $\Delta t$ . The key to understanding this method is the *Taylor* series expansion of  $\mathbf{x}(t)$ .

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \dot{\mathbf{x}}(t) + \frac{\Delta t^2}{2!} \ddot{\mathbf{x}}(t) + \frac{\Delta t^3}{3!} \dddot{\mathbf{x}}(t) + \dots \quad (9.11)$$

We get the Euler method by discarding (truncating) all but the first two terms of the series. In other words, the Euler method only agrees with the exact solution if the first derivative is constant and higher order derivatives are zero.

## 9.4 Algorithms

177



**FIGURE 9.2** ODE algorithms in the numerics package implement the `ODESolver` interface and the `ODEAdaptiveSolver` interface.

The Euler method has an error per step that is proportional to the discarded term which is proportional to  $\Delta t^2$ . Halving the step size decreases this local error by a factor of four. Unfortunately, we also have to take twice as many steps so the *global error* over a fixed interval is proportional to  $\Delta t$ . The exponent that determines how the error depends on step size is known as the *order* of the algorithm. It is written as  $O(\Delta t^m)$ . Because error accumulates as the integration proceeds from  $t_0$  to  $t_{final}$ , the global error of the Euler method is first order,  $O(\Delta t^1)$ . Although it is tempting to choose a very small value of  $\Delta t$  to reduce this error, a great many time steps will be required. Because of the long computation time, the accumulation of arithmetic error due to a computer's finite precision, and the instability noted when solving the simple harmonic oscillator model, the Euler method is not recommended.

There are three general approaches to improving the accuracy of the Euler method:

- **Taylor** series methods, such as Euler-Richardson, can achieve higher accuracy through higher derivatives of  $f_i$ .
- **Multi-step** methods, such as Verlet, can achieve higher accuracy by using information about the rate at multiple time steps  $\{\dots t_{n-2}, t_{n-1}, t_n, t_{n+1} \dots\}$ .
- **Runge-Kutta** methods can achieve higher accuracy by evaluating the rate equation at intermediate steps between  $t_n$  and  $t_{n+1}$ .

**LISTING 9.5** Euler's method for solving differential equations. Only the `step` method is shown.

```

1 // Euler algorithm implemented using the ODESolver interface.
2 // ODESolver methods are also defined in the abstract superclass.
3 package org.opensourcephysics.numerics;
4
5 public class Euler extends AbstractODE {
6     public Euler(ODE ode) {
7         super(ode);
8     }
9
10    public double step() {
11        double[] state = ode.getState();
12        ode.getRate(state, rate);
13        for(int i = 0; i < numEqn; i++) {
14            state[i] = state[i] + stepSize * rate[i];
15        }
16        return stepSize;
17    }
18}

```

**TABLE 9.1** ODE solvers implemented in the `org.opensourcephysics.numerics` package.

<code>Euler</code>	Euler algorithm.
<code>Verlet</code>	Verlet algorithm.
<code>EulerRichardson</code>	Euler-Richardson 2-nd order algorithm with fixed step size.
<code>RK4</code>	Runge-Kutta 4-th order algorithm with fixed step size.
<code>RK45</code>	An adaptive step size algorithm based on 4-th and 5-th order Runge-Kutta methods run in tandem.
<code>ODEBisectionEventSolver</code>	A solver that deals with StateEvents using a bisection root finder.
<code>ODEMultistepSolver</code>	A solver that takes a succession of adaptive steps to produce a fixed step size.

### Taylor Series Methods

Taylor series methods include higher order terms of the Taylor expansion of the solution. Truncating the series expansion at second order is particularly intuitive for Newton's second law problems because the second derivative of the position is the force per unit mass,  $a = F/m$ . Physicists simplify this expansion further by dropping the third term in the velocity rate thereby assuming constant acceleration

## 9.4 Algorithms

179

throughout the time step,  $\Delta t$ . Using physics notation, the Taylor series expansion for position and velocity is written as:

$$x_n^{i+1} = x_n^i + v_n^i \Delta t + \frac{1}{2} a_n \Delta t^2 \quad (9.12a)$$

$$v_n^{i+1} = v_n^i + a_n \Delta t. \quad (9.12b)$$

The truncation error is now third order for position but only second order for velocity. The resulting approximation, known as the *midpoint* method, is very appealing because it can be shown to be equivalent to using the average velocity throughout the interval.

$$x_n^{i+1} = x_n^i + \frac{v_n^{i+1} + v_n^i}{2} \Delta t. \quad (9.13)$$

The midpoint algorithm is exact for projectile problems but it suffers from the same instability as Euler's method for other types of problems. A better Taylor-series method is the Euler-Richardson algorithm which uses the state at the beginning of the interval to estimate the rate at the midpoint.

$$x_{mid} = x_n^i + \frac{1}{2} v_n^i \Delta t \quad (9.14a)$$

$$v_{mid} = v_n^i + \frac{1}{2} a(x^i, v^i) \Delta t \quad (9.14b)$$

$$x_n^{i+1} = x_n^i + v_{mid} \Delta t + O(\Delta t)^3 \quad (9.14c)$$

$$v_n^{i+1} = v_n^i + a(x_{mid}, v_{mid}) \Delta t + O(\Delta t)^3. \quad (9.14d)$$

The numerics package contains an implementation of this algorithm.

### Multi-Step Methods

*Multistep* algorithms use information from multiple time steps to calculate a new state. If the algorithm only requires information from previous states to calculate, then the algorithm is said to be *explicit*. If the algorithm uses the new state, that is, if  $x^{i+1}$  appears within the rate equation, then the algorithm is said to be *implicit*. Although implicit algorithms require that a system of Equations 9.6 be solved for  $x^{i+1}$  at every time step, they are usually very stable. This section presents a very simple explicit algorithm known as the *Verlet* method. Even though the Verlet method is only third order in position and second order in velocity, it is widely used in molecular dynamics simulations.

The Verlet method is easy to justify although we will not do so here. It requires only a single evaluation of the rate and may be written as:

$$x_n^{i+1} = 2x_n^i - x_n^{i-1} + \frac{1}{2} a_n \Delta t^2 + O(\Delta t^4) \quad (9.15a)$$

$$v_n^{i+1} = \frac{x_n^{i+1} - x_n^{i-1}}{2\Delta t} + O(\Delta t^2). \quad (9.15b)$$

The Verlet algorithm is implicit because the new value  $x_n^{i+1}$  appears on the right hand side of the velocity equation. The system of linear equations is, however, trivial. The new position is computed first and this value is used to compute the velocity.

The Verlet method's position error is fourth order and, although the velocity is only second order, it does a good job of conserving energy. Implementing algorithm 9.15 does have a small disadvantage in that it is not self starting because the initial conditions are only known at  $t = 0$  and not at  $t = -\Delta t$ . We must use another method to compute the first step.

A mathematically equivalent version of the original Verlet algorithm is given by

$$x^{i+1} = x^i + v^i \Delta t + \frac{1}{2} a^i (\Delta t)^2 \quad (9.16a)$$

and

$$v^{i+1} = v^i + \frac{1}{2} (a^{i+1} + a^i) \Delta t. \quad (9.16b)$$

We see that (9.16), known as the *velocity* form of the Verlet algorithm, is self-starting and minimizes roundoff errors. (See *An Introduction to Computer Simulation Methods, Third Edition* by Harvey Gould, Jan Tobochnik, and Wolfgang Christian for a discussion of finite difference methods for the solution of Newton's equation of motion.) Because this form of the Verlet algorithm is the most commonly used, we have implemented it in the Verlet ODE solver in the numerics package. Because the Verlet method is implicit, the rate equation for position and velocity are different and the Verlet ODE solver requires that the state and rate arrays alternate position and velocity.

### Runge-Kutta Methods

One of the most popular methods for solving ODEs is Runge-Kutta. It achieves the accuracy of Taylor series methods without the calculation of higher derivatives. All variables are treated the same. Although there are many variations of the Runge-Kutta approach, we will only examine the *fourth-order* Runge-Kutta since it balances simplicity and power.

Runge-Kutta 4-th order is the “Volkswagen” of ODE solvers; it may not have extra horsepower but it often gets the job done. In the RK4 algorithm, the derivative is computed at the beginning of the time interval, twice in the middle, and again at the end of the interval.

The algorithm starts by calculating the change in state using  $\Delta t$  as the step size.

$$\mathbf{d}_1 = \mathbf{f}(\mathbf{x}^{(i)}) \Delta t. \quad (9.17)$$

Half this change in state is added to the initial state to produce an intermediate state,  $\mathbf{x}^{(i)} + \frac{\mathbf{d}_1}{2}$ . This intermediate state is then used to calculate another change

## 9.5 Adaptive Step Size

181

in state.

$$\mathbf{d}_2 = \mathbf{f}(\mathbf{x}^{(i)} + \frac{\mathbf{d}_1}{2})\Delta t. \quad (9.18)$$

This process of estimating intermediate state and rate is repeated

$$\mathbf{d}_3 = \mathbf{f}(\mathbf{x}^{(i)} + \frac{\mathbf{d}_2}{2})\Delta t. \quad (9.19)$$

$$\mathbf{d}_4 = \mathbf{f}(\mathbf{x}^{(i)} + \mathbf{d}_3)\Delta t. \quad (9.20)$$

before the final state is calculated using a weighted average of all four results.

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \frac{\mathbf{d}_1 + 2\mathbf{d}_2 + 2\mathbf{d}_3 + \mathbf{d}_4}{6}. \quad (9.21)$$

Mathematicians tell us—and we believe them—that this weighted average is 5-th order accurate in the step size for a single step. (The algorithm is gobally accurate to 4-th order.) The method that implements this algorithm is shown in Listing 9.6.

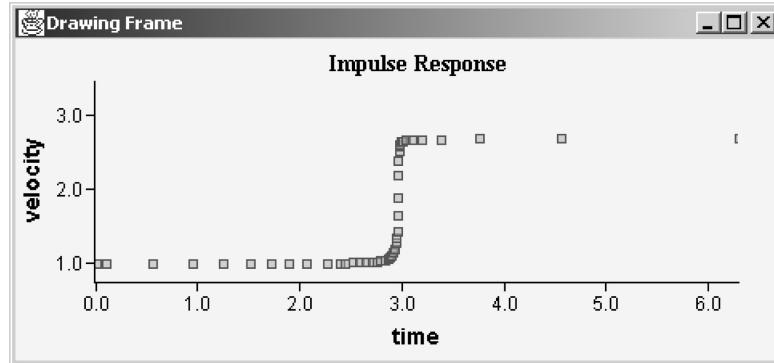
Listing 9.6 A fourth-order Runge-Kutta step.

```

1  public double step(){
2      double state []=ode.getState ();
3      ode.getRates (state , rates1 );
4      for (int i=0;i<numEqn; i++) {
5          k1 [i]=state [i]+stepSize*rates1 [i ]/2;
6      }
7      ode.getRates (k1 , rates2 );
8      for (int i=0;i<numEqn; i++) {
9          k2 [i]=state [i]+stepSize*rates2 [i ]/2;
10     }
11     ode.getRates (k2 , rates3 );
12     for (int i=0;i<numEqn; i++) {
13         k3 [i]=state [i]+stepSize*rates3 [i ];
14     }
15     ode.getRates (k3 , rates4 );
16
17     for (int i=0;i<numEqn; i++) {
18         state [i]=state [i]+stepSize*(rates1 [i ]+2*rates2 [i ]+2*rates3 [i ]+rates4 [i ]) /6;
19     }
20     return stepSize ;
21 }
```

## 9.5 ■ ADAPTIVE STEP SIZE

A major problem in solving ODEs lies in determining an appropriate step size. It is not efficient to use the same small time step throughout a simulation if the rate



**FIGURE 9.3** The velocity of a particle acted on by an impulsive force  $f(t) = e^{-t^2/\epsilon^2}$ . The time step is adjusted automatically to maintain a specified tolerance.

of change of the system is unknown or if the rate varies widely during the system's evolution. Instead an *adaptive* (variable) time step algorithm is suggested, as shown in Figure 9.3. Adaptive step size algorithms estimate the local error and increase or reduce the step size if the error is greater or less than a predetermined tolerance, respectively. The `AdaptiveSolver` interface shown in Listing 9.7 extends ODE solvers to express this additional capability.

Listing 9.7 The adaptive ODE solver interface.

```

1 public interface ODEAdaptiveSolver extends ODESolver {
2   public void setTolerance(double tol);
3
4   public double getTolerance();
5 }
```

One possible adaptive technique is to advance the state using different step sizes and compare results. The first computation is performed using the full time step,  $\Delta t$ . The second computation is done using two half-steps,  $\Delta t/2$ . If the two answers agree to within the specified tolerance, then we assume that the solution has the desired accuracy and accept the result. Otherwise, the time step is reduced and the process is repeated. Although this scheme usually works, it is very inefficient.

In the 1960s E. Fehlberg discovered that it was possible to evaluate five intermediate rates to obtain a Runge-Kutta solution that has the interesting property that the same intermediate rates can be used to evaluate a fourth order and a fifth order algorithm. It is, therefore, possible to run a 4-th order and a 5-th order numerical algorithm in tandem. Fehlberg recognized that the difference between these two algorithms provided a good estimate of the 4-th order method's error. Suppose that we have an estimated error of  $10^{-3}$  but that we require an error of no more than  $10^{-8}$ . Because the local error is  $O(\Delta t^5)$  we can decrease the step

## 9.5 Adaptive Step Size

183

size by a factor of 10.

$$\Delta t' = \left( \frac{10^{-8}}{10^{-3}} \right)^{1/5} \Delta t = \Delta t / 10 \quad (9.22)$$

Conversely, we would increase the step size if the error is too small. The RK45 ODE solver implements this technique to adjust the step size until a predetermined tolerance is reached. (See the source code for additional details.)

Figure 9.3 shows the velocity of a particle influenced by a force that varies rapidly,  $F(x) = \epsilon/(\epsilon^2 + x^2)$ . The particle has an initial velocity  $v = 1$ . Note the nonuniform increments along the time axis. At first there is little variation in the force and the step size increases. As the particle approaches  $x = 0$  the force changes rapidly and the adaptive solver increases and decreases the step size as needed. Finally, the particle resumes its uniform force-free motion and the step size becomes very large. Listing 9.8 shows the code for this example.

Listing 9.8 Simulation of a particle acted on by an impulsive force.

```

1 package org.opensourcephysics.manual.ch09;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.numerics.*;
4 import java.awt.Color;
5
6 public class AdaptiveStepApp {
7     public static void main(String[] args) {
8         PlottingPanel panel = new PlottingPanel("time", "velocity",
9                                         "Impulse Response");
10        DrawingFrame frame = new DrawingFrame(panel);
11        panel.setSquareAspect(false);
12        panel.setPreferredMinMaxY(0.8, 3.5);
13        Dataset dataset = new Dataset();
14        dataset.setMarkerShape(Dataset.SQUARE);
15        dataset.setMarkerColor(new Color(255, 128, 128, 128),
16                               Color.red);
17        panel.addDrawable(dataset);
18        ODE ode = new Impulse();
19        ODEAdaptiveSolver ode_solver = new RK45(ode);
20        ode_solver.initialize(0.1); // the initial step size
21        ode_solver.setTolerance(1.0e-4);
22        while(ode.getState()[0]<12) {
23            dataset.append(ode.getState()[2], ode.getState()[1]);
24            ode_solver.step();
25        }
26        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
27        frame.setVisible(true);
28        panel.repaint();
29    }
30}
```

```

31 // The test rate equation.
32 class Impulse implements ODE {
33     double epsilon = 0.01;
34     double[] state = new double[] {-3.0, 1.0, 0.0}; // x, v, t
35
36     public double[] getState() {
37         return state;
38     }
39
40     public void getRate(double[] state, double[] rate) {
41         rate[0] = state[1];
42         rate[1] = epsilon/(epsilon*epsilon+state[0]*state[0]);
43         rate[2] = 1;
44     }
45 }
```

## 9.6 ■ MULTI-STEPPING

Adaptive step size algorithms are a poor choice whenever we wish to display a system's state at predetermined times such as when we wish to show an integral table or an animation. We have defined two ODE solvers that perform multiple adaptive steps in order to advance the independent variable by a fixed step size. Because the step size is chosen so as to achieve the desired accuracy, the last step will almost always overshoot. The `ODEMultistepSolver` automatically reduces the step size if the remaining time is less than the optimum step size. The `ODEInterpolationSolver` always takes the optimum step size and then interpolates the final state. This interpolation does not affect the accuracy of subsequent steps because the solver maintains the exact state internally. These two ODE solvers are too long to list here but they are defined in the numerics package.

In order to test the accuracy of our multi-step ODE solvers, we recast a one-dimensional integral as a differential equation. Consider the following indefinite integral:

$$F(t) = \int_a^t f(x) dx \quad \text{where} \quad F(a) = 0. \quad (9.23)$$

Differentiating with respect to  $t$  produces the following first-order differential equation:

$$\frac{dF(t)}{dt} = f(t). \quad (9.24)$$

We now consider the *sine integral* because it cannot be evaluated using standard function but is tabulated in reference books such as the classic *Handbook of*

## 9.6 Multi-Stepping

185

x	Si(x)
0	0
1	0.946
2	1.605
3	1.849
4	1.758
5	1.55
6	1.425
7	1.455
8	1.574
9	1.665

**FIGURE 9.4** The  $Si$  function evaluated at ten points.*Mathematical Functions* (1964) by M. Abramowitz and I. A. Stegun.

$$Si(z) = \int_0^z \frac{\sin x}{x} dx \quad (9.25)$$

A typical value of this integral is  $Si(10) = 1.6583475942$ . Listing 9.6 solves this integral and tabulates the values. The result is shown using a data table in Figure 9.4.

```

1 package org.opensourcephysics.manual.ch09;
2 import org.opensourcephysics.frames.*;
3 import org.opensourcephysics.numerics.*;
4
5 public class SineIntegralApp implements ODE {
6     double[] state = new double[] {0.0, 0.0};
7
8     public double[] getState() {
9         return state;
10    }
11
12    public void getRate(double[] state, double[] rates) {
13        rates[0] = 1; // independent variable
14        rates[1] = integrand(state[0]); // function to integrate
15    }
16
17    public double integrand(double x) {
18        if (x==0) {
19            return 1;
20        } else {
21            return Math.sin(state[0])/state[0];
22        }
23    }

```

```

24
25 public static void main(String[] args) {
26     TableFrame frame = new TableFrame("Sine Integral: Si(x)");
27     frame.setColumnNames(0, "x");
28     frame.setColumnNames(1, "Si(x)");
29     ODE ode = new SineIntegralApp();
30     // RK45MultiStep hides the adaptive step size from the user
31     ODEAdaptiveSolver ode_method = new RK45MultiStep(ode);
32     ode_method.setTolerance(1.0e-5);
33     ode_method.setStepSize(1.0);
34     for(int i = 0;i<10;i++) {
35         frame.appendRow(new double[] {ode.getState()[0],
36             ode.getState()[1]});
37         ode_method.step();
38     }
39     frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
40     frame.setVisible(true);
41     frame.refreshTable();
42 }
43 }
```

## 9.7 ■ HIGH-ORDER ODE SOLVERS

**TABLE 9.2** High-order ODE solvers are implemented in the `org.opensourcephysics.ode` package.

Dopri5	Dorman Prince 5-th order ODE solver with variable step size.
Dopri853	Dorman Prince 8/5/3 (12 stage) ODE solver with adaptive step size.
ODEInterpolationSolver	Adjusts its internal step size in order to obtain the desired accuracy but returns an interpolated result at a fixed step size.
Radau5	Implicit Runge-Kutta method of order 5 for stiff equations.

Andrew Gusev and Yuri. B. Senichenkov at Saint Petersburg Polytechnic University, Russia, have contributed a collection of high-order differential equation solvers to the OSP library based on the Runge-Kutta method. These solvers are shown in Table 9.2 and are defined in the `org.opensourcephysics.ode` package. The code for this optional package and for the examples in this section is available on the CD in the `osp_ode.zip` archive.

`Dopri853` is an explicit Runge-Kutta adaptive step size method that integrates a system of ordinary differential equations using a Runge-Kutta approach developed by Dorman and Prince. (See P.J. Prince and J.R. Dorman (1981) High order

## 9.7 High-Order ODE Solvers

187

embedded Runge-Kutta formulae. J.Comp. Appl. Math., Vol. 7. p.67-75.) This solver is a 8th-order accurate integrator and requires 13 function evaluations per integration step. Additional information about the Dorman and Prince formulas can be found in Hairer, Norsett and Wanner (1993).

The Radau5 solver is designed to solve stiff systems of first order ordinary differential equations of the form  $M\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$  with a possibly singular matrix  $M$  using an implicit Runge-Kutta method of order 5. The method is described in Hairer and Wanner (1996).

Although the Radau5 is not recommended for non-stiff problems, both Dopri853 and Radau5 solvers can be used and the solutions carefully compared if the user does not know the stiffness of the equations. Because Radau5 is an implicit ODE solver it takes many more calculations to compute a model solution using this solver than with Dopri853. The user should balance the time required to calculate solutions against the need for the increased solution accuracy.

The Dopri853 and Radau5 solvers implement the `ODEAdaptiveSolver` interface and can be used directly if a fixed step size is not required.

```

1 // ode implements the ODE interface
2 ODEAdaptiveSolver ode_solver = new Dopri5( ode );
3 ODEAdaptiveSolver ode_solver = new Dopri853( ode );
4 ODEAdaptiveSolver ode_solver = new Radau5( ode );

```

These algorithms can also be multi-stepped to produce a fixed step size using static methods in the `MultistepSolvers` class.

```

1 // ode implements the ODE interface
2 ODEAdaptiveSolver ode_solver =
    MultistepSolvers.MultistepDopri5(ode);
3 ODEAdaptiveSolver ode_solver =
    MultistepSolvers.MultistepDopri853(ode);
4 ODEAdaptiveSolver ode_solver =
    MultistepSolvers.MultistepRadau5(ode);

```

Another solver developed by the St. Petersburg group combines the adaptive and fixed step size approaches. The `ODEInterpolationSolver` class adjusts its internal step size in order to obtain the desired accuracy and interpolates the final state in order to maintain a fixed step size.

```

1 // ode implements the ODE interface
2 ODEAdaptiveSolver
    ode_solver=ODEInterpolationSolver.InterpolationDopri5(ode);
3 ODEAdaptiveSolver
    ode_solver=ODEInterpolationSolver.InterpolationDopri853(ode);
4 ODEAdaptiveSolver
    ode_solver=ODEInterpolationSolver.InterpolationRadau5(ode);

```

The `ODEInterpolationSolver` class makes a copy of the initial state of the system and advances this state array. If the program requests the state at some

time  $t_f$  the internal state may, in fact, be at time greater than  $t_f$ . The return values are interpolated from previously computed states that are stored in the solver. This interpolation is accurate and fast and allows the solver to always integrate the differential equations at the optimal step size. Interpolation can produce a significant increase in performance. Run the short example shown in Listing 9.9. The test program evaluates the rate equation 218 times when using the interpolation solver and 613 times when using the multi-step solver.

Listing 9.9 ODEInterpolationSolverApp test program.

```

1 package org.opensourcephysics.manual.ch09;
2 import org.opensourcephysics.ode.*;
3 import org.opensourcephysics.numerics.ODEAdaptiveSolver;
4
5 public class ODEInterpolationSolverApp {
6     public static void main(String[] args) {
7         ODETest ode = new ODETest();
8         // Create one of the following solvers for testing.
9         // ODEAdaptiveSolver ode_solver=new
10            ODEInterpolationSolver(ode); // uses default solver
11        ODEAdaptiveSolver ode_solver =
12            ODEInterpolationSolver.Dopri53(
13                ode);
14        ode_solver.setStepSize(1.0);
15        ode_solver.setTolerance(1e-6);
16        double time = 0;
17        double[] state = ode.getState();
18        while(time<25) {
19            System.out.println("x1 = "+state[0]+" \t t="+time
20                +" \t step size="+ode_solver.getStepSize());
21            time += ode_solver.step();
22        }
23    }
}

```

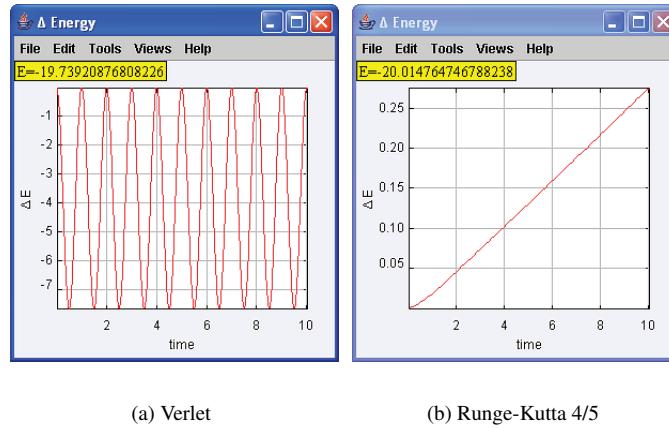
## 9.8 ■ CONSERVATION LAWS

Although it may appear that we should always use the highest order ODE solver available, this is not the case. Consider a system of  $N$  particles with pair-wise interactions. A three dimensional model with  $n$  particles has  $6 * n$  first order differential equations and computing the force for any given state requires evaluation of  $\sim n^2/2$  pairwise interactions. The force computation will, of course, be performed many times in a single step if a high-order method is used.

Because the inverse square law in classical mechanics is the prototype for pair-wise interactions, we study it to learn about the stability and accuracy of differential equation solvers. The `InverseSquare` class implements the `ODE` interface

## 9.8 Conservation Laws

189



**FIGURE 9.5** The error in total energy depends on the ODE solver.

to solve the central force problem (9.9). This class is available in the ch09 package on the CD but is not shown here because it is similar to other ODE models such as the simple harmonic oscillator. The `InverseSquareApp` program shown in Listing 9.10 uses the `InverseSquare` class to show a simulation of a particle acted on by a  $1/r^2$  force pulling toward the center.

InverseSquareApp allows the user to select one of five numerical methods: Euler, Verlet, RK4, RK45, or RK45MultiStep. Default initial conditions are set so as to produce a circular orbit with radius  $r = 1$ . Running the program shows both the particle motion and the variation in total energy  $\Delta E$  as a function of time where the total energy is given by the value of the Hamiltonian  $H$  in dimensionless units:

$$H = \frac{1}{2}v^2 + \frac{1}{r}. \quad (9.26)$$

Figure 9.5 compares the energy variation during ten orbits for the Verlet and RK45 solvers using a time step of  $\Delta t = 0.01$  and an ODE tolerance of  $10^{-3}$ . Because the RK45 algorithm uses an adaptive step size, we read the tolerance parameter from the control when instantiating this solver. The tolerance parameter has no effect on fixed step size algorithms such as Verlet. Running the program shows that the adaptive step size algorithm produces a faster animation because  $\Delta t$  is increased to achieve the desired tolerance. The total energy is, however, not constant but slowly drifts to lower values. Decreasing the tolerance lowers the rate of energy loss but does not change the overall behavior.

**Listing 9.10** InverseSquareApp creates a simulation of a particle acted on by a  $1/r^2$  force pulling toward the center.

package org.opensourcephysics.manual.ch09;

```

2  import org.opensourcephysics.controls.*;
3  import org.opensourcephysics.display.*;
4  import org.opensourcephysics.frames.*;
5  import org.opensourcephysics.numerics.*;

6
7  public class InverseSquareApp extends AbstractSimulation {
8      DisplayFrame orbitFrame = new DisplayFrame("x (AU)", "y (AU)",
9                                         "Particle Orbit");
10     PlotFrame energyFrame = new PlotFrame("time", "$\\Delta$ E",
11                                         "$\\Delta$ Energy");
12     InverseSquare particle = new InverseSquare();
13     double initialEnergy;

14
15    public InverseSquareApp() {
16        orbitFrame.addDrawable(particle);
17        orbitFrame.setPreferredMinMax(-2.5, 2.5, -2.5, 2.5);
18        orbitFrame.setSquareAspect(true);
19        energyFrame.setConnected(true);
20        energyFrame.setMarkerShape(0, Dataset.NO_MARKER);
21    }

22
23    public void doStep() {
24        particle.doStep(); // advances time
25        double energy = particle.getEnergy();
26        energyFrame.append(0, particle.getTime(),
27                           initialEnergy-energy);
28        orbitFrame.setMessage("t = "
29                            +decimalFormat.format(particle.state[4]));
29        energyFrame.setMessage("E=" +energy, DrawingPanel.TOP_LEFT);
30    }

31
32    public void initialize() {
33        String solver = control.getString(
34            "ODE Solver").toLowerCase().trim();
35        if(solver.equals("euler")) {
36            particle.odeSolver = new Euler(particle);
37        } else if(solver.equals("verlet")) {
38            particle.odeSolver = new Verlet(particle);
39        } else if(solver.equals("rk4")) {
40            particle.odeSolver = new RK45(particle);
41        } else if(solver.equals("rk45")) {
42            ODEAdaptiveSolver adaptiveSolver = new RK45(particle);
43            adaptiveSolver.setTolerance(
44                control.getDouble("adaptive stepsize solver
45                    tolerance"));
46            particle.odeSolver = adaptiveSolver;
47        } else if(solver.equals("rk45multistep")) {
48            ODEAdaptiveSolver adaptiveSolver = new
49                RK45MultiStep(particle);

```

## 9.8 Conservation Laws

191

```

48     adaptiveSolver.setTolerance(
49         control.getDouble("adaptive stepsize solver
50             tolerance"));
51     particle.odeSolver = adaptiveSolver;
52 } else {
53     control.print(
54         "Solver not found. Valid solvers are: Euler, Verlet,
55             RK4, RK45, RK45MultiStep");
56 }
57 particle.odeSolver.setStepSize(control.getDouble("dt"));
58 double x = control.getDouble("x");
59 double vx = control.getDouble("vx");
60 double y = control.getDouble("y");
61 double vy = control.getDouble("vy");
62 particle.initialize(new double[] {x, vx, y, vy, 0});
63 orbitFrame.setMessage("t = 0");
64 initialEnergy = particle.getEnergy();
65 energyFrame.setMessage("E="+initialEnergy ,
66     DrawingPanel.TOP_LEFT);
67 }
68
69 public void reset() {
70     control.setValue("ODE Solver", "Verlet");
71     control.setValue("x", 1);
72     control.setValue("vx", 0);
73     control.setValue("y", 0);
74     control.setValue("vy", "2*pi");
75     control.setValue("dt", 0.01);
76     control.setValue("adaptive stepsize solver tolerance", 1e-3);
77     enableStepsPerDisplay(true);
78     initialize();
79 }
80
81 public static void main(String[] args) {
82     SimulationControl.createApp(new InverseSquareApp(), args);
83 }

```

Although the Verlet algorithm also does not keep the total energy constant, it is better than the higher order algorithm because the energy oscillates thereby conserving the average energy. Because speed of computation and the conservation of energy are crucial, the Verlet algorithm is often used in molecular dynamics simulations with large numbers of particles (molecules). Statistical averages are of primary interest while the detailed trajectories of individual particles are unimportant in these types of simulations.

Figure 1.1 shows the trajectories of two electrons in a classical Helium model. The electrons repel each other and are attracted toward the central core. The program that produced this figure is named HeliumApp and is available in the ch0 9

package. The rate equation is defined int the Helium class and is hard-wired for the Helium model as follows:

```

1 public void getRate(double [] state , double [] rate ){
2   // state []: x1, vx1, y1, vy1, x2, vx2, y2, vy2, t
3   double deltaX=(state[4]-state[0]);           // x12 separation
4   double deltaY=(state[6]-state[2]);           // y12 separation
5   double dr_2=(deltaX*deltaX+deltaY*deltaY); // r12 squared
6   double dr_3=Math. sqrt(dr_2)*dr_2;          // r12 cubed
7
8   rate[0]=state[1];                           // x1 rate
9   rate[2]=state[3];                           // y1 rate
10
11  double r_2=state[0]*state[0]+state[2]*state[2]; // r1 squared
12  double r_3=r_2*Math. sqrt(r_2);              // r1 cubed
13  rate[1]=-2*state[0]/r_3-deltaX/dr_3;         // vx1 rate
14  rate[3]=-2*state[2]/r_3-deltaY/dr_3;         // vy1 rate
15
16  rate[4]=state[5];                           // x2 rate
17  rate[6]=state[7];                           // y2 rate
18  r_2=state[4]*state[4]+state[6]*state[6];    // r2 squared
19  r_3=r_2*Math. sqrt(r_2);                  // r2 cubed
20  rate[5]=-2*state[4]/r_3+deltaX/dr_3;        // vx2 rate
21  rate[7]=-2*state[6]/r_3+deltaY/dr_3;        // vy2 rate
22  rate[8]=1;                                // time rate
23 }
```

The complexity of the Helium rate makes it apparent that a more systematic approach is needed to model the n-body problem. The PlanarNBodyApp class in the ch09 models a number of n-body problems with periodic orbits. The `getRate` method in the `PlanarNBody` class is much simpler because this class implements the `computeForce` method to compute the net force on each particle. Each particle has  $x, v_x, y, v_y$  dynamical variables and these variables are accessed one particle at a time in a loop.

```

1 public void getRate(double [] state , double [] rate ) {
2   computeForce(state); // force array alternates fx and fy
3   for(int i=0; i<n; i++){
4     int i4=4*i;
5     rate[i4] = state[i4+1];      // x rate is vx
6     rate[i4+1] = force[2*i];    // vx rate is fx
7     rate[i4+2] = state[i4+3];    // y rate is vy
8     rate[i4+3] = force[2*i+1];  // vy rate is fy
9   }
10  rate[state.length -1]=1;      // time rate is last
11 }
```

The NBodyApp program models an arbitrary number of gravitationally interacting particles but includes a number of additional features such as the ability to

save and restore systems of particles using an extensible markup language (XML) data file as discussed in Chapter 12. The `NBodyApp` program is used as a curriculum development example in Chapter 15.

## 9.9 ■ COLLISIONS AND STATE EVENTS

Solving differential equations in the presence of collisions is difficult. One way to model a collision is to add a force component perpendicular to the surface of contact. Consider a particle colliding with the floor. We can add a spring force to the rate equation with a spring constant  $k$  so that the particle will be pushed away from the floor when it falls below the surface

$$F_{\text{spring}}(z) = -kz \quad (z < 0). \quad (9.27)$$

We can avoid interpenetration of the particle into the floor by choosing a large value for  $k$  but we will then have to take many time steps during the collision resulting in what is known as a *stiff* ODE. A stiff ODE is characterized by more than one time scale. In our example, the time scale for projectile motion is of the order of seconds whereas the time scale for the collision is of the order of milliseconds. Even sophisticated ODE solvers, such as Runge-Kutta, may experience difficulty in solving this type of dynamics because they assume that values in the state array vary smoothly. Non-penetrating collisions violate this assumption.

A better way to model a non-penetrating collision is to detect the collision and then respond to it using the impulse approximation to instantaneously change the particle's velocity. This is a typical example of what we call a *state event*. The simplest collision (state event) to consider is an *elastic collision* without friction. The parallel component of the particle's velocity is unaffected and the normal component of the velocity is negated. In an *inelastic collision* the normal component is multiplied by the negative of a number  $r$  between zero and one called the *coefficient of restitution*. If  $r = 1$  the collision is elastic and if  $r = 0$  the collision is said to be totally inelastic.

If a collision occurs at time  $t_c$  we must stop the ODE solver and compute new velocities based on the physics of the collision. However, before we can solve the physics we must deal with the geometric issue of detecting the time of contact between the bodies. Suppose that  $t_c$  lies between  $t_i$  and  $t_{i+1} = t_i + \Delta t$ . A simple way of determining  $t_c$  is to use the method of *bisection*. If we detect interpenetration at  $t_i + \Delta t$  we reset the ODE solver back to  $t_i$  and compute the state at  $t_i + \Delta t/2$ . If the interpenetration has not yet occurred we know the collision lies between  $t_i + \Delta t/2$  and  $t_i + \Delta t$  and we advance the simulation from  $t_i + \Delta t/2$  to  $t_i + 3\Delta t/4$ . Otherwise it lies between  $t_i$  and  $t_i + \Delta t/2$  and we advance the simulation from  $t_i$  to  $t_i + \Delta t/4$ . This process is repeated until the collision time is computed to within some suitable numerical tolerance. The `ODEBisectionEventSolver` class in the numerics package implements this algorithm.

The `ODEBisectionEventSolver` is a differential equation solver that responds to an illegal state by determining the time when the state becomes illegal

and taking an appropriate corrective action. The illegal condition and the action are encapsulated in an object that implements the `StateEvent` interface. The key to this approach is to define an event function  $f(t)$  that returns  $f(t) \geq 0$  when a state is legal and  $f(t) < 0$  when a state is illegal.

**Listing 9.11** A `StateEvent` defines an illegal condition and a corrective action for an ODE solver.

```

1 package org.opensourcephysics.numerics;
2 public interface StateEvent {
3     public double getTolerance();
4
5     public double evaluate(
6         double[] state); // illegal state returns negative
7
8     public boolean action(); // action when state is illegal
9 }
```

The `evaluate` method returns a value of an event function  $f(t)$ . Finding the time when a state becomes illegal consists of finding the root of this function.

In order to properly deal with numerical approximations, an event is assumed to have taken place when  $f(t) \leq -\epsilon$  and a given state will be considered legal if  $f(t) \geq +\epsilon$  where the tolerance  $\epsilon$  is the value returned by the `getTolerance` method. The event solver will attempt to reduce  $\delta t$  using the method of bisection to find a time  $t_c$  that brings the system to a state where  $|f(t_c)| < \epsilon$ . The event solver then invokes the `action` method.

The `action` method must bring the system to a state such that either  $f(t_c) \geq \epsilon$  or that this condition will become true after a finite number of iterations.

Several possible state events may be defined for the same ODE. In this case, if more than one event takes place in the same interval  $[t, t + \delta t]$ , the solver must search for the one that takes place first and apply the corresponding action. If two of the events take place at the very same instant, they will be dealt with by the solver in an arbitrary order. Care must be taken not to produce infinite loops by events that trigger a second event which in turn trigger again the first event. This will result in the computer hanging at this point. A second caveat consists in the so-called *Zeno* problem described below.

### Elastic Collision Event

The `ElasticCollision` class models a floor-ball collision using the height of the ball above the floor minus the ball's radius as the event function. Notice that we only trigger an event if the ball is falling.

**Listing 9.12** An elastic collision state event.

```

1 class ElasticCollision implements StateEvent {
2
3     public double getTolerance() {
4         return TOL;
```

## 9.9 Collisions and State Events

195

```

5    }
6
7    public double evaluate(double[] state) {
8        return(state[1]<0)           // v must be negative
9            ? state[0]-radius // y should be >= radius
10           : TOL;           // return a legal state
11    }
12
13   public boolean action() {
14       state[1] = -state[1]; // make vy positive
15       stopAtCollision = true; // change this to false to
16           continue simulation
17       return stopAtCollision; // stop the integration step at the
18           collision
19   }
20 } // end of inner class ElasticCollision

```

The state event action is triggered when the event function is less than zero. This reverses the velocity, sets the `stopAtCollision` boolean, and returns. The returned value tells the solver whether it should stop the computation at the exact moment of the event or continue solving the ODE for the rest of the prescribed step  $\Delta t$ .

The `BallFloorCollision` class uses the `ElasticCollision` class as an inner class to trigger events in an `ODEBisectionEventSolver`. The solver is instantiated using an RK4 algorithm and the elastic collision event is created and added to the solver in the constructor. The `BallFloorCollisionApp` is available in the `ch09` package but is not shown here because it is similar to other applications.

Listing 9.13 A floor-ball collision can be modeled using a `StateEvent`.

```

1 package org.opensourcephysics.manual.ch09;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.numerics.*;
4 import java.awt.*;
5
6 public class BallFloorCollision implements ODE, Drawable {
7     static final double TOL = 0.001;
8     static final double g = 9.8;           // acceleration of
9         gravity
10    double[] state = new double[] {10, 0, 0}; // y, v, t
11    double
12        radius = 1, dt = 0.1;
13    boolean stopAtCollision = false;
14    DrawableShape box = DrawableShape.createRectangle(0, -0.5, 10,
15        1);
16    DrawableShape ball = DrawableShape.createCircle(0, 0,
17        2*radius);
18    // a solver that supports state events

```

```

16      ODEBisectionEventSolver solver = new
17          ODEBisectionEventSolver(this,
18              RK4.class);
19
20  public BallFloorCollision() {
21      // choose one of the following two StateEvents
22      // solver.addEvent(new ElasticCollision());
23      solver.addEvent(new InelasticCollision());
24      solver.initialize(dt);
25      ball.setMarkerColor(new Color(128, 128, 255), Color.BLUE);
26  }
27
28  void doStep() {
29      solver.step();
30  }
31
32  public double[] getState() {
33      return state;
34  }
35
36  public void getRate(double[] state, double[] rate) {
37      rate[0] = state[1]; // dy/dt = v
38      rate[1] = -g;        // fails with inelastic collisions due
39                      // to Zeno effect
40      rate[2] = 1.0;       // dt/dt = 1
41  }
42
43  public void draw(DrawingPanel panel, Graphics g) {
44      ball.setXY(0, state[0]);
45      ball.draw(panel, g);
46      box.draw(panel, g);
47  }
48
49  private class ElasticCollision implements StateEvent {
50      public double getTolerance() {
51          return TOL;
52      }
53
54      public double evaluate(double[] state) {
55          if(state[1]<0)           // v must be negative
56              state[0]-=radius;    // y should be >= radius
57          else
58              state[0]+=radius;    // return a legal state
59      }
60
61      public boolean action() {
62          state[1] = -state[1];   // make vy positive
63          stopAtCollision = true; // change this to false to
64                          // conintue

```

## 9.9 Collisions and State Events

197

```

62     return stopAtCollision; // stop the integration step at
63     // the collision
64 }
65 } // end of inner class ElasticCollision
66
67 private class InelasticCollision implements StateEvent {
68     double r = 0.8; // coefficient of restitution
69
70     public double getTolerance() {
71         return TOL;
72     }
73
74     public double evaluate(double[] state) {
75         return (state[1]<0) // v must be negative
76             ? state[0]-radius // y should be >= radius
77             : TOL;           // return a legal state
78     }
79
80     public boolean action() {
81         state[1] = -r*state[1]; // make vy positive and reduce
82         // its value
83         stopAtCollision = false; // change this to true to stop
84         return stopAtCollision; // stop the integration step at
85         // the collision
86     }
87 } // end of inner class InelasticCollision
88

```

**Bisection**

Although the bisection method is one with a rather slow convergence rate for the general problem, it presents several advantages for this particular situation. First, it only assumes that  $f$  is continuous when computing the root; actually, in many real problems, our `evaluate` method returns non-differentiable values.<sup>1</sup> Second, the lack of speed is compensated by the smaller number of evaluations of the function  $f(t)$ . When evaluating the function is expensive (in terms of computations), the bisection method compares better to faster convergent methods (which require several evaluations of  $f(t)$  for each iteration step). Third, the method allows simple separation of events when more than one take place in the same interval. Finally, because the step size  $\Delta t$  for most ODEs is already small, convergence is usually quickly attained.

The `BallFloorCollision` class contains a second state event named `InelasticCollision`. This event's `action` method reduces the velocity of the falling ball at every bounce in order to model more realistic physics. The model will, however, fail if the elastic event solver is used due to what is known as as a

<sup>1</sup>The state function  $f(t)$  may even be discontinuous if it does not trigger a state event near the discontinuity. The `InelasticCollision` example is discontinuous when the velocity changes sign.

*Zeno* problem. In the real world, the ball's velocity is zero as the ball reaches its equilibrium state.

Events take place at decreasingly smaller time intervals as the ball approaches equilibrium. In other words, when the vertical velocity is very small, rebounding takes place in a sequence of  $t_n$  such that  $t_{n+1} - t_n \rightarrow 0$ . The solution to this problem is to change the physics embodied in the `getRate` method when the ball is in resting contact with the floor. Otherwise we obtain infinite force during interpenetration or we must model the deformation of the ball and the floor. The net force (acceleration) of the ball is zero when the ball is in resting contact with the floor and the computation of `rate[1]` should reflect this condition. Inserting the following code fragment into the `BallFloorCollision` class solves the Zeno problem because the ball no longer falls below the floor and therefore does not trigger state events when in resting contact.

```

1 rate[1] = (state[0]-radius>0) // resting on floor changes physics
2     ? -g      // dv/dt = - g when falling
3     : 0;      // dv/dt = 0 when resting

```

There is a vast body of literature dealing with collision detection and resting contact. Needless to say, we have only scratched the surface. If the motion occurs in two dimensions, then the constraints act normal to the surface of contact. Simple examples of two-dimensional simulations are the `BallBoxCollisionApp` and the `BallsInBoxApp` programs available in the `ch09` package on the CD.

## 9.10 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch09` package.

### **AdaptiveStepApp**

`AdaptiveStepApp` demonstrates the advantage of an adaptive step size ODE-Solver by integrating a particle's response to short-duration force.

### **BallBoxCollisionApp**

`BallBoxCollisionApp` simulates a bouncing ball in a box using `StateEvent` objects to constrain the position of the ball.

### **BallFloorCollisionApp**

`BallFloorCollisionApp` models a particle colliding with the floor using a `StateEvent`.

### **BallsInBoxApp**

`BallsInBoxApp` simulates interacting (colliding) hard disks constrained within a box and falling under the influence of gravity.

**HeliumApp**

HeliumApp models electron orbits in a classical Helium atom.

**InverseSquareApp**

InverseSquareApp models a particle orbiting under the influence of an inverse square  $1/r^2$  force.

**NBodyApp**

NBodyApp models an arbitrary number of particles interacting through a  $1/r^2$  force.

**PlanarNBodyApp**

PlanarNBodyApp models special cases of the gravitational few body problem with closed periodic orbits.

**SHOApp**

SHOApp tests the ODESolver interface using a simple harmonic oscillator.

**SineIntegralApp**

SineIntegralApp creates a table of values by integrating the function  $\sin x/x$ .

The following examples require the `org.opensourcephysics.ode` package from Saint Petersburg Polytechnic University, Russia. This package is available on the OSP web site.

**ComparisonApp**

ComparisonApp displays the difference between an exact solution and a numeric solution.

**DopriApp**

DopriApp tests Dorman-Prince ODESolver implementations.

**ODEInterpolationSolverApp**

ODEInterpolationSolverApp tests ODE interpolation solver implementations.

**ODEMultistepSolverApp**

ODEMultistepSolverApp tests multi-step solver implementations.

## CHAPTER

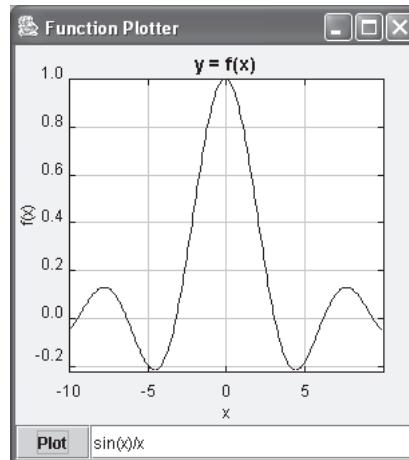
## 10

## Numerics

©2005 by Wolfgang Christian, 19 July 2005

Numerical tools are defined in the `org.opensourcephysics.numerics` package. Because we do not seek to implement a large scientific subroutine library, this package emphasizes numerical algorithms and APIs that we have found to be important in the teaching of computational physics.

### 10.1 ■ FUNCTIONS



**FIGURE 10.1** A function plotter uses a parser to convert a string of characters into a function.

One of the most fundamental concepts in mathematics is that of a function. A *function* is a rule,  $f$ , that gives a well-defined output,  $y$ , corresponding to some well-defined input,  $x$ , specifically

$$y = f(x) \quad \text{for some domain of } x. \quad (10.1)$$

The variable  $y$  is called the dependent variable and the variable  $x$  is called the independent variable. The set of permissible values of  $x$  is called the *domain* and

## 10.1 Functions

201

the set of all possible outputs is called the *range*. Figure 10.1 shows a simple function plotter that is able to evaluate keyboard input.

The numerics package defines the `Function` interface that allows us to program functions with domain and range restricted to values allowed by Java data type `double`.

Listing 10.1 Function interface.

```

1 package org.opensourcephysics.numerics;
2 public interface Function {
3     public double evaluate(double x);
4 }
```

We can use the `Function` interface to implement real-valued functions such as the amplitude of a single slit diffraction pattern ( $f(x) = \sin x/x$ ).

```

1 public class SlitPattern implements Function {
2     public double evaluate (double x) {
3         return (x==0)? 1 :Math.sin (x)/x;
4     }
5 }
```

The numerics package also contains interfaces for functions of more than one variable. `MultiVarFunction` defines an interface for an object that accepts an array of numbers and returns a number, while `VectorFunction` defines an interface for an object that accepts an array of numbers and returns an array of numbers. The `RectangularPattern` class models a multi-variable function for the diffraction pattern from a rectangular aperture  $f(x, y) = (\sin x/x)(\sin y/y)$  with a height to width ratio of 0.4.

```

1 public class RectangularPattern implements MultiVarFunction {
2     double ratio =0.4;
3     public double evaluate (double[] x) {
4         SlitPattern sp=new SlitPattern ();
5         return sp.evaluate(x[0])*sp.evaluate(ratio*x[1]);
6     }
7 }
```

To display a function in a drawing panel, we must evaluate and plot a series of points  $(x, f(x))$  for a given interval. Although we can use a loop to add points to a dataset, we can also use the `FunctionDrawer` class in the display package. The `FunctionDrawer` evaluates the given function by computing values at the specified number of points within the domain.

```

1 // drawingPanel is a DrawingPanel
2 boolean filled=false; // area under function not filled
3 int n =200;           // number of data points
4 double xmin=-10, xmax=10; // plotting domain
5 Function f = new SlitPattern ();
6 drawingPanel.addDrawable(
```

```
7 new FunctionDrawer(f, xmin, xmax, n, filled));
```

If the `FunctionDrawer` is instantiated with only a single parameter  $f(x)$ , then the domain is undefined. The `FunctionDrawer` draws the function by evaluating points within the drawing panel's x-minimum and maximum values.

The `functionFill` utility method in the numerics `Util` class saves us the trouble of writing a loop to evaluate a function and storing values in an array. This utility method takes the following input parameters: (1) a function, (2) minimum and maximum values for the independent variable, and (3) an integer specify the number of data points. It returns a two-column array containing  $x$  and  $y$  values which can then be passed to a dataset or a data table for display.

```
1 // f is an object that implements Function
2 double[][] array=Util.functionFill(f,-10,10,new double[2][21]);
```

x	sin(x)/x
-4.0	-0.18920062382...
-3.0	0.04704000268...
-2.0	0.45464871341...
-1.0	0.84147098480...
0.0	1.0
1.0	0.84147098480...
2.0	0.45464871341...
3.0	0.04704000268...
4.0	-0.18920062382...
5.0	-0.19178485493...
6.0	-0.04656924969...

**FIGURE 10.2** A table of single slit diffraction values.

Creating the table of the single slit diffraction values shown in Figure 10.2 takes just a few lines of code as shown in Listing 10.2. The code uses a `TableFrame` (see Section 3.8) and an anonymous class to define the `Function`.

Listing 10.2 `SlitTableApp` creates a data table using a `Function` to fill an array.

```
1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.frames.*;
3 import org.opensourcephysics.numerics.*;
4
5 public class SlitTableApp {
6     public static void main(String[] args) {
7         TableFrame frame = new TableFrame("Single Slit Diffraction");
8         Function f = new Function() { // create anonymous class
9             public double evaluate(double x) {
10                 return (x==0) ? 1 : Math.sin(x)/x;
11             }
12         };
13         frame.setFunction(f);
14         frame.show();
15     }
16 }
```

## 10.2 Derivatives

203

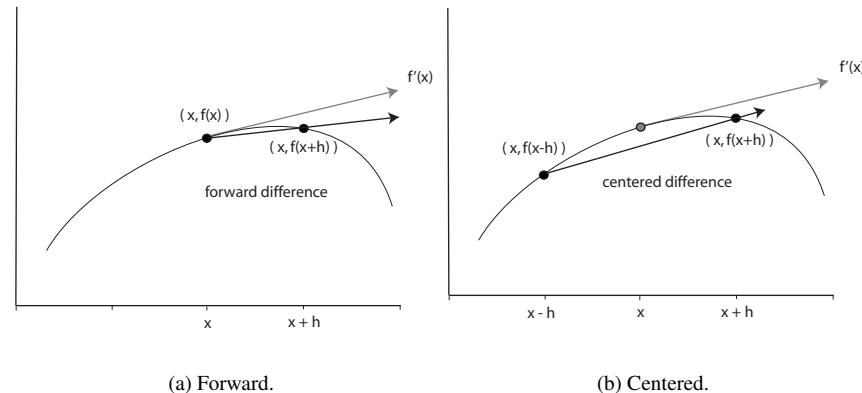
```

11    }
12  };
13  double [][] array = Util.functionFill(f, -10, 10,
14          new double [2][21]);
15  frame.setRowNumberVisible(false);
16  frame.appendArray(array);
17  frame.setColumnNames(0, "x");
18  frame.setColumnNames(1, "sin(x)/x");
19  frame.setVisible(true);
20  frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
21}
22}

```

Many classes in the numerics package define methods that instantiate `Function` objects or themselves implement `Function`. For example, the `Util` class contains convenience methods for creating a constant function  $f(x) = c$ , a linear function  $f(x) = mx + b$ , and a Gaussian  $f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-x_0)^2/2\sigma^2}$ . The `Polynomial` and `CubicSpline` classes implement `Function`. We can take the derivative, integrate, or Fourier analyze these objects because the algorithms that perform these mathematical operations are written independent of the type of function that they will receive.

## 10.2 ■ DERIVATIVES



**FIGURE 10.3** The true derivative of a function, shown as a light arrow, can be approximated by evaluating a function at two or more points. The (a) forward and (b) centered approximations to the first derivative are shown as dark arrows.

If a function  $f(x)$  is evaluated at two points  $x$  and  $x + h$ , we can compute the

slope of the line that passes through those points. The value of the slope as  $h \rightarrow 0$ , shown a light arrow in Figure 10.3, is the first derivative of the function:

$$f'(x) = \frac{df(x)}{dx} \equiv \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (\text{first derivative}). \quad (10.2)$$

We can approximate this definition by using a small value of  $h$ , subtracting the terms in the numerator, and performing the division. This approximation is known as a first-order forward finite difference. Higher order approximations can be made. The `Derivative` class in the numerics package defines second-order approximations to the first derivative by evaluating the function  $f(x)$  at various points in the vicinity of  $x$  where  $h$  is the interval over which the approximation is made.

The following first-derivative approximations are second order in  $h$  but use the intervals to the right of  $x$ , centered on  $x$ , and to the left of  $x$ . The centered approximation is the most efficient but the other methods are useful if the function  $f(x)$  is only defined to one side of  $x$ .

$$\frac{df(x)}{dx} \approx \frac{-f(x+2h) + 4f(x+h) - 3f(x)}{2h} \quad (10.3a)$$

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x-h)}{2h} \quad (\text{centered}) \quad (10.3b)$$

$$\frac{df(x)}{dx} \approx \frac{f(x-2h) - 4f(x-h) + 3f(x)}{2h}. \quad (10.3c)$$

These formulas are coded in the `Derivative` class.

```

1 static public double forward(Function f, double x, double h){
2     return (-f.evaluate(x+2*h) +
3             4*f.evaluate(x+h) - 3*f.evaluate(x))/h/2.0;
4 }
5
6 static public double centered(Function f, double x, double h){
7     return (f.evaluate(x + h) - f.evaluate(x - h)) / h/2.0;
8 }
9
10 static public double backward(Function f, double x, double h){
11     return (f.evaluate(x-2*h) -
12             4*f.evaluate(x-h) + 3*f.evaluate(x))/h/2.0;
13 }
```

The `Derivative` class also implements a first-derivative algorithm based on Romberg's implementation of Richardson's deferred approach to the limit. This algorithm computes the derivative multiple times using ever smaller values of  $h$ . The sequence of results can be extrapolated to the limit  $h \rightarrow 0$ , and this extrapolation can also be used to estimate an error. If the estimated error is greater than the given tolerance, then the calculation is repeated and another term is added to

## 10.2 Derivatives

205

the sequence. This algorithm is implemented in the `romberg` method. Computing multiple approximations and calculating the tolerance does, of course, require extra computer cycles.

Finite difference approximations can be derived for higher order derivatives and for partial derivatives of multi-variable functions. The most useful is the centered finite difference approximation for the second derivative:

$$\frac{d^2 f(x)}{dx^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{2h}. \quad (10.4)$$

This formula is implemented in the `Derivative` class.

```
1 static public double second(Function f, double x, double h) {
2     return (f.evaluate(x+h)-2*f.evaluate(x)+f.evaluate(x-h))/h/h;
3 }
```

Static methods in the `Derivative` class are useful for evaluating derivatives at points, but it is sometimes convenient to create a derivative `Function`. The `getFirst` method returns a `Function` object that does just that. (See Section 10.6 how to compute the derivative of a polynomial.)

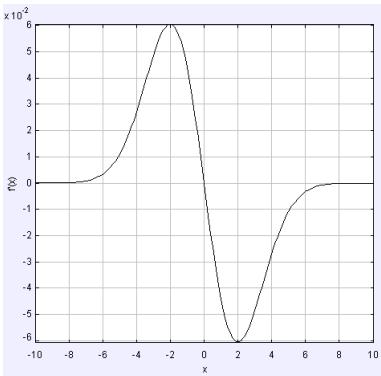
```
1 static public Function getFirst(final Function f, final double
2                                h) {
3     return new Function() {
4         public double evaluate(double x) { // in-line the code for
5             speed
6             return (f.evaluate(x+h)-f.evaluate(x-h))/h/2.0;
7         }
8     };
9 }
```

The `getSecond` method is defined in a similar manner and returns a second derivative `Function`.

Gaussian functions (normal distributions) occur frequently in physics and the `Util` class in the numerics package defines the `gaussian` method that instantiates a Gaussian `Function`  $f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-x_0)^2/2\sigma^2}$  with a specified center  $x_0$  and width  $\sigma$ . The full width at half maximum is  $2\sqrt{2\ln 2}\sigma \approx 2.3548\sigma$ . We plot the derivative of this function as shown in Figure 10.4 using numerical derivatives and a `FunctionDrawer` as shown in Listing 10.3. Note the high level of abstraction that has been achieved by using the `Function` interface.

Listing 10.3 `GaussianDerivativeApp` plots of the numerical derivative of a Gaussian `Function`.

```
1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.frames.*;
4 import org.opensourcephysics.numerics.*;
```

**FIGURE 10.4** The numerical derivative of a Gaussian Function.

```

6  public class GaussianDerivativeApp {
7      public static void main(String[] args) {
8          PlotFrame frame = new PlotFrame("x", "f'(x)",
9              "Gaussian Derivative");
10         Function f = Util.gaussian(0, 2.0); // center=0; sigma=2
11         Function df = Derivative.getFirst(f, 0.001);
12         frame.addDrawable(new FunctionDrawer(df, -10, 10, 200,
13             false));
14         frame.setVisible(true);
15     }
16 }
```

### 10.3 ■ INTEGRALS

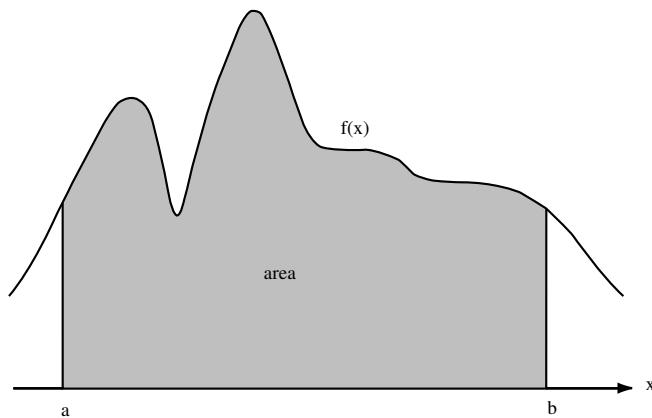
Evaluating a definite integral for an arbitrary function  $\int_a^b f(x) dx$  is more complicated than evaluating the derivative. The integral of a function can be visualized as the area under the curve from  $a$  to  $b$  as shown in Figure 10.5. Unfortunately, even simple functions, such as  $\sin x/x$ , cannot be integrated analytically and we must resort to numerical methods. The `Integral` class implements a number of algorithms for the approximation of integrals including the trapezoidal method, Simpson's method, Romberg's method, and the ODE approach described in Section 9.5.

The rectangular approximation is the simplest integral approximation. It computes the area assuming the value of the function  $f(x)$  is constant between  $x$  and  $x + \Delta x$ .

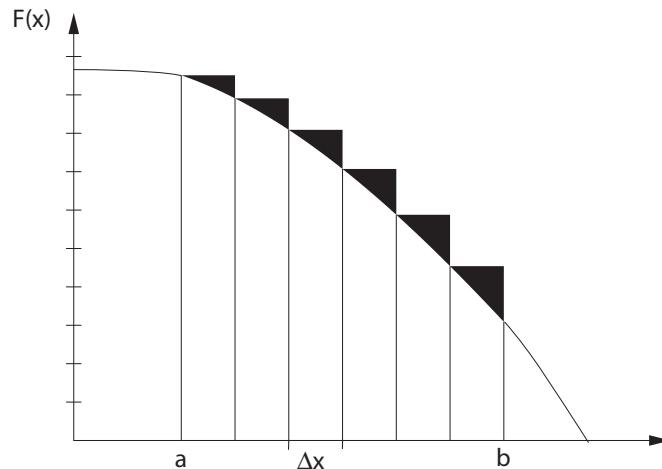
$$\int_a^b f(x) \approx \sum_{i=0}^{n-1} f(x_i) \Delta x \quad (10.5)$$

## 10.3 Integrals

207

**FIGURE 10.5** The integral  $F$  equals the area under the curve  $f(x)$ .

Because the error shown in Figure 10.6 is first order in  $\Delta x$ , the rectangular approximation is used only for illustration. The trapezoidal approximation assumes a linear change between  $f(x)$  and  $f(x + \Delta x)$ , and Simpson's approximation assumes a polynomial fit between sampled points.

**FIGURE 10.6** The rectangular approximation for  $f(x)$  for  $a \leq x \leq b$ . The error is shaded.

`IntegralApp` shown in Listing 10.4 tests the accuracy of algorithms in the `Integral` class and estimates their efficiency by printing the number of function evaluations needed to obtain the desired precision. Run the test program and no-

tice that Romberg's method is the most efficient for high precision results. Simpson's methods are more robust because they merely evaluate a sum, and these algorithms are usually sufficient.

Listing 10.4 IntegralApp test the integration algorithms.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.numerics.*;
3
4 public class IntegralApp {
5     static final double LN2 = Math.log(2);
6
7     public static void main(String[] Args) {
8         Function f = new IntegralTestFunction();
9         double
10            a = 1, b = 2;
11        double tol = 1.0e-10; // double has 16 significant digits
12        double area = Integral.ode(f, a, b, tol);
13        System.out.println("ODE area="+area+" err="+((area-LN2)));
14        System.out.println("counter="+IntegralTestFunction.c);
15        IntegralTestFunction.c = 0;
16        area = Integral.trapezoidal(f, a, b, 2, tol);
17        System.out.println("Trapezoidal area="+area+
18            " err="+((area-LN2)));
19        System.out.println("counter="+IntegralTestFunction.c);
20        IntegralTestFunction.c = 0;
21        area = Integral.simpson(f, a, b, 2, tol);
22        System.out.println("Simpson area="+area+" err="+((area-LN2)));
23        System.out.println("counter="+IntegralTestFunction.c);
24        IntegralTestFunction.c = 0;
25        area = Integral.romberg(f, a, b, 2, tol);
26        System.out.println("Romberg area="+area+" err="+((area-LN2)));
27        System.out.println("counter="+IntegralTestFunction.c);
28        IntegralTestFunction.c = 0;
29    }
30
31    class IntegralTestFunction implements Function {
32        static long c = 0;
33
34        public double evaluate(double x) {
35            c++;
36            return 1.0/x;
37        }
38    }

```

## 10.4 Parsers

209

**TABLE 10.1** The OSP expression parser accepts common one and two parameter functions.

abs(a)	acos(a)	acosh(a)	asin(a)	asinh(a)
atan(a)	atanh(a)	ceil(a)	cos(a)	cosh(a)
exp(a)	frac(a)	floor(a)	int(a)	log(a)
random(a)	round(a)	sign(a)	sin(a)	sinh(a)
sqr(a)	sqrt(a)	step(a)	tan(a)	tanh(a)
atan2(a,b)	max(a,b)	min(a,b)	mod(a,b)	

**10.4 ■ PARSERS**

An object that evaluates a string of characters and produces a numeric result is known as a mathematical expression *parser*. Parsers are very useful since they allow a user to change a mathematical function without recompiling the program. Two expression parsers are included on the Open Source Physics CD. One parser was written by Yanto Suryono and another by Nathan Frank. The first parser is very fast but is restricted to real numbers while the second supports both real and complex data types.

To hide implementation details and to have a consistent—albeit minimal—API to parse functions of real variables, we defined an abstract class named `MathExpParser`. A `MathExpParser` class implements both the `Function` and `MultiVarFunction` interfaces and declares abstract methods to set the function and the independent variable strings. The `SuryonoParser` class is a concrete implementation of a `MathExpParser` and this class is used whenever we need a real-valued parser. The `ParsedFunction` class that is used in Section 2.3 to process user input contains a `SuryonoParser` object.

```

1 String fx="sin(x)/x"; // usually read from a keyboard
2 try {
3     function = new ParsedFunction(fx, "x");
4 } catch(ParserException ex) { // user input errors are
5     common
6     function = Util.constantFunction(0); // set f(x)=0 if there is
      an error
}
```

Using a parser is straightforward. Define a character string and pass it to the parser either in the constructor or using the `setFunction` method as shown in Listing 10.5. Because users often make typing mistakes when entering mathematical expressions, the `setFunction` method throws an exception if the string does not represent a valid expression. Catching the exception allows the program to inform the user of the mistake and take corrective action.

Listing 10.5 `ParseRealApp` parses a string to produce a function.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.numerics.*;
```

```

3
4 public class ParseRealApp {
5     public static void main(String[] args) {
6         MathExpParser parser = MathExpParser.createParser();
7         try {
8             parser.setFunction("sin(2*pi*x)", new String[] {"x"});
9         } catch(ParserException ex) {
10             System.out.println(ex.getMessage());
11         }
12         for(double x = 0, dx = 0.1;x<1;x += dx) {
13             System.out.println("x="+Util.f2(x)+"\t"
14                 +parser.evaluate(x));
15         }
16     }
}

```

The complex expression parser is distributed in the `jep.zip` code archive. You must download and import this package into your Java development environment before running the `ParseComplexApp` example shown in Listing 10.6

Listing 10.6 `ParseComplexApp` parses a string to produce a function.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.numerics.*;
3 import org.nfunk.*;
4 import org.nfunk.jep.type.Complex;
5
6 public class ParseComplexApp {
7     public static void main(String[] args) {
8         JEPParser parser = null;
9         try {
10             parser = new JEPParser("e^(i*x)", "x",
11                 JEPParser.MAKECOMPLEX);
12         } catch(ParserException ex) {
13             System.out.println(ex.getMessage());
14         }
15         for(double x = 0, dx = 0.1;x<1;x += dx) {
16             Complex z = parser.evaluateComplex(x);
17             System.out.println("x="+Util.f2(x)+"\t"
18                 +Util.f3(z.re())
19                 +" \t im="+Util.f3(z.im()));
20         }
}

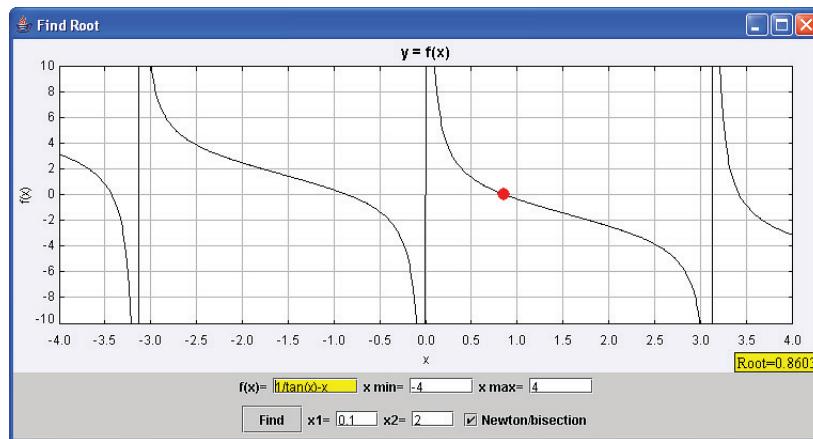
```

The `RealWaveApp` and `ComplexWaveApp` programs are more elaborate examples that use real and complex parsers to animate time dependent wave functions  $f(x, t)$ . They are not shown here but are available on the CD.

## 10.5 Finding the Roots of a Function

211

## 10.5 ■ FINDING THE ROOTS OF A FUNCTION



**FIGURE 10.7** Finding the roots of a function is difficult because a function such as  $f(x) = \cot x - x = 0$  can have many roots.

The roots of a function  $f(x)$  are the values of the variable  $x$  for which the function  $f(x)$  is zero. Even an apparently simple problem, such as finding the energy roots of the quantum finite well,

$$f(x) = \cot x - x = 0 \quad (10.6)$$

cannot be solved analytically for  $x$ .

Whatever the function and whatever the approach to root finding, the first step should be to learn as much as possible about the function or functions involved. Analyzing the corresponding physics and mathematics is the best approach. One technique is to plot a function to help guess the approximate locations of the roots.

Newton's (or the Newton-Raphson) method for finding a root is based on replacing the function by the first two terms of the Taylor expansion of  $f(x)$  about the root  $x$ . If our initial guess for the root is  $x_0$ , we can write  $f(x) \approx f(x_0) + (x - x_0)f'(x_0)$ . If we set  $f(x)$  equal to zero and solve for  $x$ , we find  $x = x_0 - f(x_0)/f'(x_0)$ . If we made a good choice for  $x_0$ , the resultant value of  $x$  should be closer to the root. The general procedure is to calculate successive approximations as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (10.7)$$

If this series converges, it converges very quickly. However, if the initial guess is poor or if the function has closely spaced multiple roots, the series may not converge.

The Newton's algorithm is implemented in the `Root` class in the numerics package with two different signatures. The more efficient method requires a derivative function. The other method takes a numeric derivative using the Romberg's method. Both methods return not-a-number (`Double.NaN`) if the method fails to converge.

**Listing 10.7** The `newton` method is defined in the `Root` class in the numerics package.

```

1  public static double newton( Function f, double x, double tol) {
2      int count = 0;
3      while (count < MAX_ITERATIONS) {
4          double xold = x; // save the old value to test for
5              convergence
6          double df = 0;
7          try {
8              df = Derivative.romberg(f, x,
9                  Math.max(0.001, 0.001 * Math.abs(x)), tol / 10);
10         }
11         catch (NumericMethodException ex) {
12             return Double.NaN; // did not converge
13         }
14         x -= f.evaluate(x) / df;
15         if (Util.relativePrecision(Math.abs(x - xold), x) <
16             tol) return x;
17         count++;
18     }
19     return Double.NaN; // did not converge in max iterations
20 }
21
22 public static double newton(final Function f, final Function df,
23     double x, final double tol) {
24     int count = 0;
25     while (count < MAX_ITERATIONS) {
26         double xold = x; // save the old value to test for
27             convergence
28         // approximate the derivative using centered difference
29         x -= f.evaluate(x) / df.evaluate(x);
30         if (Util.relativePrecision(Math.abs(x - xold), x) <
31             tol) return x;
32         count++;
33     }
34     return Double.NaN; // did not converge in max iterations
35 }
```

The simplest root-finding algorithm is the *bisection* method. The algorithm works as follows:

1. Choose two values,  $x_{\text{left}}$  and  $x_{\text{right}}$ , with  $x_{\text{left}} < x_{\text{right}}$ , such that the

## 10.5 Finding the Roots of a Function

213

product  $g^{(p)}(x_{\text{left}})g^{(p)}(x_{\text{right}}) < 0$ . There must be a value of  $x$  such that  $g^{(p)}(x) = 0$  in the interval  $[x_{\text{left}}, x_{\text{right}}]$ .

2. Choose the midpoint,  $x_{\text{mid}} = x_{\text{left}} + \frac{1}{2}(x_{\text{right}} - x_{\text{left}}) = \frac{1}{2}(x_{\text{left}} + x_{\text{right}})$ , as the guess for  $x^*$ .
3. If  $g^{(p)}(x_{\text{mid}})$  has the same sign as  $g^{(p)}(x_{\text{left}})$ , then replace  $x_{\text{left}}$  by  $x_{\text{mid}}$ ; otherwise, replace  $x_{\text{right}}$  by  $x_{\text{mid}}$ . The interval for the location of the root is now reduced.
4. Repeat steps 2 and 3 until the desired level of precision is achieved.

The bisection algorithm is also implemented in the `Root` class in the numerics package. It can be used with any function but it requires that the root be bracketed by two values having opposite sign.

**Listing 10.8** The bisection method is defined in the `Root` class in the numerics package.

```

1  public static double bisection(final Function f, double x1,
2                                double x2, final double tol) {
3      int count = 0;
4      int maxCount = (int)(Math.log(Math.abs(x2 -
5                                         x1)/tol)/Math.log(2));
6      maxCount = Math.max(MAX_ITERATIONS, maxCount) + 2;
7      double y1 = f.evaluate(x1), y2 = f.evaluate(x2);
8      if (y1 * y2 > 0) { // y1 and y2 must have opposite sign
9          return Double.NaN; // interval does not contain a root
10     }
11     while (count < maxCount) {
12         double x = (x1 + x2) / 2;
13         double y = f.evaluate(x);
14         if(Util.relativePrecision(Math.abs(x1-x2), x)<tol) return x;
15         if (y * y1 > 0) { // replace end-point that has the same sign
16             x1 = x;
17             y1 = y;
18         } else {
19             x2 = x;
20             y2 = y;
21         }
22         count++;
23     }
24     return Double.NaN; // did not converge in max iterations
25 }
```

There are many root-finding techniques but all can be made to fail with appropriately chosen functions. The bisection algorithm is guaranteed to converge if we can find an interval where the function changes sign. However, it may be slow. Newton's algorithm is fast but may not converge. A hybrid algorithm that keeps track of the interval between the most negative and the most positive values

while iterating is a good approach. The algorithm starts with Newton's method. If the solution jumps outside the given interval, Newton's method is interrupted and the bisection algorithm is used for one step. But any root algorithm can fail so be sure and test the root at the end of the iteration process to check that the algorithm actually converged.

The RootApp program compares Newton's algorithm and the bisection method for finding the root(s) of  $f(x) = x^2 - 2$ . Both algorithms succeed in finding the positive root  $\sqrt{2}$  with the given parameters. However, if we set  $x_1=0$  Newton's algorithm fails and if we set  $x_1=-10$  and  $x_2=10$  the bisection algorithm fails for obvious reasons. The RootFinderApp program provides a graphical user interface for exploring the properties of these two algorithms. This program is now shown here but is available on the CD.

**Listing 10.9** RootApp compares Newton's algorithm and the bisection root finding methods.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.numerics.*;
3
4 public class RootApp {
5     public static void main(String[] Args) {
6         Function f = new Function() {
7             public double evaluate(double x) {
8                 double f = x*x-2;
9                 return f;
10            }
11        };
12        double x1 = 1;
13        double x2 = 2;
14        double tol = 1.0e-3;
15        double root = Root.bisection(f, x1, x2, tol);
16        System.out.println("bisection method root="+root);
17        root = Root.newton(f, x1, tol);
18        System.out.println("Newton's method root="+root);
19    }
20}

```

## 10.6 ■ POLYNOMIALS

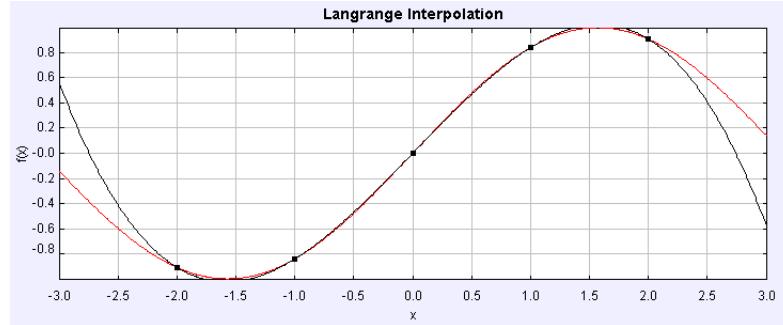
A polynomial is a function that is expressed as

$$p(x) = \sum_{i=0}^n a_i x^i, \quad (10.8)$$

where  $n$  is the *degree* of the polynomial and the  $n$  constants  $a_i$  are the *coefficients*. The evaluation of (10.8) as written is very inefficient because  $x$  is repeatedly

## 10.6 Polynomials

215



**FIGURE 10.8** A polynomial approximation to the sin function. The polynomial's coefficients were computed by constructing a Lagrange interpolating polynomial passing through the points indicated by black markers. The computed coefficients are:  $c[0]=0.0$ ,  $c[1]=0.9704117419395818$ ,  $c[2]=4.85722573273506E-17$ ,  $c[3]=-0.12894075713168524$ ,  $c[4]=6.938893903907228E-18$ .

multiplied by itself and the entire sum requires  $\mathcal{O}(N^2)$  multiplications. A more efficient algorithm was published in 1819 by W. G. Horner.<sup>1</sup> It uses a factored polynomial and requires only  $n$  multiplications and  $n$  additions and is now known as *Horner's rule*. It is written as follows:

$$p(x) = a_0 + x[a_1 + x[a_2 + x[a_3 + \dots]]]. \quad (10.9)$$

Using the correct algorithm for this simple task can dramatically reduce processor time if large polynomials are repeatedly evaluated.

Polynomials are important computationally because most analytic functions can be approximated as a polynomial using a Taylor series expansion. Polynomials can be added, multiplied, integrated, and differentiated analytically and the result is still a polynomial. This property makes them ideally suited for object-oriented programming. The `Polynomial` class in the numerics package implements many of these algebraic operations (see Table 10.2). Listing 10.10 shows a test program that demonstrates how to calculate and display a polynomial's roots.

Listing 10.10 The `PolynomialApp` class tests the `Polynomial` class.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.*;
4 import org.opensourcephysics.numerics.*;
5
6 public class PolynomialApp extends AbstractCalculation {
7     PlottingPanel drawingPanel = new PlottingPanel("x", "f(x)",
8                                         "Polynomial Visualization");
9     DrawingFrame drawingFrame = new DrawingFrame(drawingPanel);

```

<sup>1</sup>This method of evaluating polynomials by factoring was already known to Newton.

**TABLE 10.2** Methods for manipulating polynomial coefficients are defined in the `Polynomial` class.

<b>org.opensourcephysics.numerics.Polynomial</b>	
<code>add(double a)</code>	Adds a scalar to this polynomial and returns a new polynomial.
<code>add(Polynomial p)</code>	Adds a polynomial to this polynomial and returns a new polynomial.
<code>deflate(double r)</code>	Reduces the degree of this polynomial by removing the given root $r$ .
<code>derivative()</code>	Returns the derivative of this polynomial.
<code>integral(double a)</code>	Returns the integral of this polynomial having the value $a$ at $x = 0$ .
<code>subtract(double a)</code>	Subtracts a scalar from this polynomial and returns a new polynomial.
<code>subtract(Polynomial p)</code>	Subtracts a polynomial from this polynomial and returns a new polynomial.

```

10  double xmin , xmax ;
11  Polynomial p;
12
13  public void resetCalculation() {
14      control.setValue("coefficients", "-2,0,1");
15      control.setValue("xmin", -10);
16      control.setValue("xmax", 10);
17  }
18
19  public void calculate() {
20      xmin = control.getDouble("xmin");
21      xmax = control.getDouble("xmax");
22      String[] coef = control.getString("coefficients").split(",");
23      p = new Polynomial(coef);
24      plotAndCalculateRoots();
25  }
26
27  void plotAndCalculateRoots() {
28      drawingPanel.clear();
29      drawingPanel.addDrawable(new FunctionDrawer(p));
30      double[] range = Util.getRange(p, xmin, xmax, 100);
31      drawingPanel.setPreferredMinMax(xmin, xmax, range[0],
32          range[1]);
33      drawingPanel.repaint();
34      double[] roots = p.roots(0.001);
35      control.clearMessages();
36      control.println("polynomial="+p);
37      for(int i = 0, n = roots.length;i<n;i++) {
            control.println("root="+roots[i]);
    }
}

```

## 10.6 Polynomials

217

```

38    }
39 }
40
41 public void derivative() {
42     p = p.derivative();
43     plotAndCalculateRoots();
44 }
45
46 public static void main(String[] args) {
47     Calculation app = new PolynomialApp();
48     CalculationControl c = new CalculationControl(app);
49     c.addButton("derivative", "Derivative",
50                 "The derivative of the polynomial.");
51     app.setControl(c);
52 }
53 }
```

It is always possible to construct a polynomial that passes through a set of  $n$  data points  $(x_i, y_i)$  by creating a *Lagrange interpolating polynomial* as follows:

$$p(x) = \sum_{i=0}^n \frac{\prod_{i \neq j} (x - x_j)}{\prod_{i \neq j} (x_i - x_j)} y_i. \quad (10.10)$$

For example, three data points generate the second-degree polynomial:

$$p(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_3. \quad (10.11)$$

Note that terms multiplying the  $y$  values will be zero at the sample points except for the term multiplying the sample point's abscissa  $y_i$ . Various computational tricks can be used to speed the evaluation of (10.10), but these will not be discussed here. See Bessel or Press *et al.* We have implemented polynomial interpolation using a generalized Horner expansion in the `LagrangeInterpolator` class in the numerics package. Listing 10.11 tests this implementation by sampling the  $\sin x$  at five data points and fitting these points to a polynomial. The output is shown in Figure 10.8.

Listing 10.11 The `LagrangeInterpolatorApp` class tests the `LagrangeInterpolator` class by sampling an arbitrary function and fitting the samples by a polynomial.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.*;
4 import org.opensourcephysics.numerics.*;
5
6 public class LagrangeInterpolatorApp extends AbstractCalculation
{
```

```

7 PlottingPanel drawingPanel = new PlottingPanel("x", "f(x)",
8                                     "Langrange Interpolation");
9 DrawingFrame drawingFrame = new DrawingFrame(drawingPanel);
10 Dataset dataset = new Dataset();
11
12 public LagrangeInterpolatorApp() {
13     dataset.setConnected(false);
14 }
15
16 public void resetCalculation() {
17     control.setValue("f(x)", "sin(x)");
18     control.setValue("sample start", -2);
19     control.setValue("sample stop", 2);
20     control.setValue("n", 5);
21     control.setValue("random y-error", 0);
22     calculate();
23 }
24
25 public void calculate() {
26     String fstring = control.getString("f(x)");
27     double a = control.getDouble("sample start");
28     double b = control.getDouble("sample stop");
29     double err = control.getDouble("random y-error");
30     int n = control.getInt("n"); // number of intervals
31     double dx = (n>1) ? (b-a)/(n-1) : 0;
32     Function f;
33     try {
34         f = new ParsedFunction(fstring);
35     } catch(ParserException ex) {
36         control.println(ex.getMessage());
37         return;
38     }
39     drawingPanel.clear();
40     dataset.clear();
41     double[] range = Util.getRange(f, a, b, 100);
42     drawingPanel.setPreferredMinMax(a-(b-a)/4, b+(b-a)/4,
43         range[0],
44         range[1]);
45     drawingPanel.addDrawable(dataset);
46     FunctionDrawer func = new FunctionDrawer(f);
47     func.color = java.awt.Color.RED;
48     drawingPanel.addDrawable(func);
49     double x = a;
50     for(int i = 0;i<n;i++) {
51         dataset.append(x,
52             f.evaluate(x)*(1+err*(-0.5+Math.random())));
53         x += dx;
54     }
55     LagrangeInterpolator interpolator = new

```

## 10.6 Polynomials

219

```

54     LagrangeInterpolator(dataset
55                         .getXPoints(), dataset
56                         .getYPoints());
57     drawingPanel.addDrawable(new FunctionDrawer(interpolator));
58     drawingPanel.repaint();
59     drawingFrame.setVisible(true);
60     double[] coef = interpolator.getCoefficients();
61     for(int i = 0;i<coef.length;i++) {
62         control.println("c["+i+"]="+coef[i]);
63     }
64 }
65 public static void main(String[] args) {
66     Calculation app = new LagrangeInterpolatorApp();
67     Control c = new CalculationControl(app);
68     app.setControl(c);
69 }
70 }
```

Lagrange interpolation should be used cautiously. If the degree of the polynomial is high, if the distance between points is large, or if the points are subject to experimental error, the resulting polynomial can oscillate wildly. Press *et al.* recommend that the degree of an interpolating polynomials be small. If the the data table is accurate but large, we often use multiple polynomials constructed from a small number of nearest neighbor points. *Cubic spline* interpolation uses polynomials in this way.

A cubic spline is a third-order polynomial that is required to have a continuous second derivative with neighboring splines at its end points. Because it would be inefficient to store a large number of *Polynomial* objects, the *CubicSpline* class in the numerics package stores the coefficients for the multiple polynomials needed to fit a data set in a single array. The *CubicSplineApp* program tests this class but it is not shown here because it too is similar to *Lagrange-InterpolatorApp*.

If the sample data are inaccurate, we often compute the coefficients for a polynomial of lower degree that passes as close as possible to the sample points. This fitting procedure often is used to construct an *ad hoc* function that describes experimental data. The *PolynomialLeastSquareFit* class in the numerics package implements such a fitting algorithm (see Bessel) and the *PolynomialFit-App* program tests this class. It is not shown because it is similar to *Lagrange-InterpolatorApp*.

Suppose you are given a table of  $y_i = f(x_i)$  and are asked to determine the value of  $x$  that corresponds to a given  $y$ . In other words, how do we find the inverse function  $x = f^{-1}(y)$ ? An interpolation routine that does not require evenly spaced ordinates, such as the *CubicSpline* class, provides an easy and effective solution. The following code uses this technique to define an arcsin function.

Listing 10.12 The *Arcsin* class demonstrates how to use interpolation to define

an inverse function.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.numerics.*;
3
4 public class Arcsin {
5     static Function arcsin;
6
7     private Arcsin() {} // prohibit instantiation because all
8     methods are static
9
10    static public double evaluate(double x) {
11        if(x<-1||x>1) {
12            return Double.NaN;
13        } else {
14            return arcsin.evaluate(x);
15        }
16    }
17
18    static { // static block is executed when class is first loaded
19        int n = 10;
20        double[] xd = new double[n];
21        double[] yd = new double[n];
22        double
23            x = -Math.PI/2, dx = Math.PI/(n-1);
24        for(int i = 0;i<n;i++) {
25            xd[i] = x;
26            yd[i] = Math.sin(x);
27            x += dx;
28        }
29        arcsin = new CubicSpline(yd, xd);
30    }
}

```

## 10.7 ■ FAST FOURIER TRANSFORM (FFT)

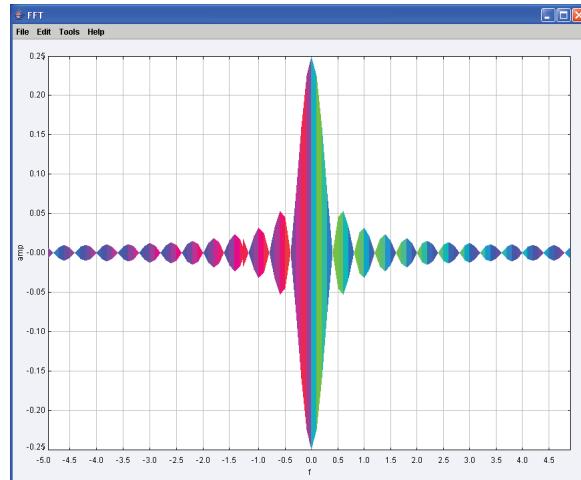
The discrete Fourier transform is used throughout science and mathematics to convert a grid of spacial or temporal data into the frequency domain. If we sample a complex function (take data) at evenly spaced intervals  $\Delta$ , we generate a sequence of  $N$  complex numbers

$$\{g_n\} = \{g_0, g_1, \dots, g_{N-2}, g_{N-1}\} \quad (10.12)$$

where every value  $g_n$  has a real and an imaginary part. Set the imaginary part equal to zero if the data are real-valued. (We describe a better way to handle real-valued data later.) We compute Fourier coefficients  $h_k$  by evaluating the following

## 10.7 Fast Fourier Transform (FFT)

221

**FIGURE 10.9** The Fourier transformation of a Heaviside step function.

sum

$$h_k = \sum_{n=0}^{N-1} g_n e^{-i2\pi nk/N} \quad (10.13)$$

thereby producing a new sequence of complex numbers

$$\{h_k\} = \{h_{-N/2}, h_{-N/2+1}, \dots, h_0, \dots, h_{N/2-1}, h_{N/2}\}. \quad (10.14)$$

This new sequence can be converted back into the original sequence using the inverse Fourier transform:

$$g_n = \frac{1}{N} \sum_{k=-N/2}^{N/2} h_k e^{i2\pi nk/N}. \quad (10.15)$$

The sequence of values (10.14) correspond to the amplitudes of complex exponentials,  $e^{i2\pi g_n x}$ , that make up the original data (10.12). We use complex exponentials because these functions are the most general and because sine and cosine transformations can be performed by reordering the data. Sine, cosine, and exponential functions are, of course, related through Euler's (De Moivre's) formula:

$$e^{ix} = \cos x + i \sin x. \quad (10.16)$$

If  $\Delta$  represents the spacial interval in 10.12 measured in meters, then the interval between values in 10.14 is in cycles per meter. If  $\Delta$  represents a temporal interval measured in seconds, then the interval in 10.14 is in cycles per second.

We make use of the fast Fourier transform (FFT) algorithm to implement the discrete Fourier transformation. A detailed discussion of this transformation may be found in most numerical methods textbooks including *Numerical Recipes* by Press *et al.* We have adapted an open source implementation of this algorithm for use in the numerics package.<sup>2</sup> The FFTApp program shown in Listing 10.13 tests the FFT class using a single frequency to generate a sequence of Fourier coefficients (10.14).

Listing 10.13 The FFTApp program test the FFT class by transforming a sinusoidal function into the frequency domain.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.numerics.*;
3
4 public class FFTApp {
5     public static void main(String[] args) {
6         int numpts = 10;
7         double[] data = new double[2*numpts];
8         FFT fft = new FFT(numpts);
9         double
10            x = 0, dx = 1.0/numpts;
11        double cycles = 1;
12        for(int i = 0;i<numpts;i++) {
13            data[2*i] = Math.cos(cycles*2*Math.PI*x);
14            data[2*i+1] = Math.sin(cycles*2*Math.PI*x);
15            x += dx;
16        }
17        System.out.println("Data before FFT.");
18        printData(data);
19        fft.transform(data);
20        System.out.println("Data after FFT.");
21        printData(data);
22        fft.inverse(data);
23        System.out.println("Data after FFT inverse.");
24        printData(data);
25    }
26
27    static void printData(double[] data) {
28        for(int i = 0, n = data.length/2;i<n;i++) {
29            String re = Util.f4(data[2*i]);
30            String im = Util.f4(data[2*i+1]);
31            System.out.println("i="+i+"\t re="+re+"\t im="+im);
32        }
33        System.out.println(); // blank line separator
34    }
35}

```

<sup>2</sup> The implementation is based on code contributed by Brian Gough to the GNU Scientific Library (GSL). It was converted to Java by Bruce Miller at NIST.

## 10.7 Fast Fourier Transform (FFT)

223

**TABLE 10.3** Wrap-around ordering of data in the space (or time) domain and the frequency domain.

index	space or time	frequency
0	$g[x = 0]$	$h[f = 0]$
1	$g[x = \Delta]$	$h[f = f_0]$
2	$g[x = 2\Delta]$	$h[f = 2f_0]$
.	.	.
$N/2$	$g[x = N * \Delta/2]$	$h[f = N * f_0/2 = -N * f_0/2]$
.	.	.
$N - 2$	$g[x = (N - 2) * \Delta]$	$h[f = -2 * f_0]$
$N - 1$	$g[x = (N - 1) * \Delta]$	$h[f = -f_0]$

Run the FFTApp program with various input frequencies (number of cycles) and observe the following:

- Complex data are represented by two double values in the input array. Thus,  $N$  data points are represented by a double array with dimension  $2N$ .
- $N$  complex data points produce  $N$  complex frequency coefficients.
- The transform of the given harmonic function produces a single nonzero coefficient if the data contains an integer number of cycles. Otherwise, there will be leakage into adjacent coefficients. Although the input amplitude equals one, the output amplitude is greater than one and is equal to the number of data points.
- The ordering of the output does not correspond to the frequency. There is a discontinuity in frequency half way through the transformed data.
- The inverse transform reproduces the original data.

The ordering and scaling conventions of FFT algorithms vary across disciplines and this often causes unnecessary confusion. We use a common numerical convention known as the engineering convention that scales the FFT output by the number of data points  $N$ . The output frequencies are arranged in what is known as *wrap-around* order. (See Table 10.3.)

Wrap-around ordering may appear strange but it is computationally efficient. We can easily filter an audio signal by transforming the data, removing the unwanted frequencies, and applying the inverse inverse transformation if we know where a given frequency occurs in the output array. The `getWrappedDf` convenience method generates an array of frequencies in wrap-around order.

Wrap-around order is, however, awkward when visualizing the frequency spectrum and we have therefore implemented the `toNaturalOrder` method to rearrange and normalize the data. The `getNaturalDf` generates the corresponding array of frequencies. The `FFTPlotApp` program shown in Listing 10.14 plots the Fourier coefficients of a simple step function in natural order.

The Fourier transformation has a number of symmetries that can easily be checked using the FFTApp program. For example:

- If the data are real-valued and even, then transform is real and the coefficient of any positive frequency equals the coefficient of the corresponding negative frequency.
- If the data are imaginary-valued and even, then transform is imaginary and the coefficient of any positive frequency equals the coefficient of the corresponding negative frequency.
- If the data are real-valued and odd, then transform is imaginary and the coefficient of any positive frequency equals the negative of the coefficient of the corresponding negative frequency.
- If the data are imaginary-valued and odd, then transform is real and the coefficient of any positive frequency equals the negative of the coefficient of the corresponding negative frequency.

These symmetries are evident if we transform a Heaviside step function as shown in Listing 10.14. The complementary colors (phase) in the output graph (Figure 10.9) illustrate the change in the sign of positive and negative frequency coefficients.

Listing 10.14 The FFTPlotApp plots the Fast Fourier Transform (FFT) the a step function.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.numerics.FFT;
4
5 public class FFTPlotApp {
6     public static void main(String[] args) {
7         PlottingPanel panel = new PlottingPanel("f", "amp", null);
8         DrawingFrame frame = new DrawingFrame("FFT", panel);
9         ComplexDataset cdataset = new ComplexDataset();
10        panel.addDrawable(cdataset);
11        int numpts = 100;
12        double[] data = new double[2*numpts];
13        double
14            xmin = -5, xmax = 5;
15        double
16            x = xmin, dx = (xmax-xmin)/numpts;
17        for(int i = 0; i<numpts; i++) {
18            data[2*i] = (x<Math.abs(2.5)) ? 1 : -1; // real
19            data[2*i+1] = 0; // imaginary
20            x += dx;
21        }
22        FFT fft = new FFT();
23        fft.transform(data);

```

## 10.7 Fast Fourier Transform (FFT)

225

```

24     fft.toNaturalOrder(data);
25     cdataset.append(fft.getNaturalFreq(dx), data);
26     frame.setVisible(true);
27     frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
28 }
29 }
```

If the data to be transformed are real-valued, it is inefficient to construct and transform an array of complex numbers. Because doubling the size of the input array by adding complex components all of which are zero produces a symmetric frequency distribution, half of the FFT output is redundant. A more efficient algorithm transforms an input array of real numbers as if it were an array of complex numbers and then uses symmetries in the Fourier transformation to unscramble the resulting array thereby constructing an array containing only the positive frequency coefficients. Note that the number of real data points must be even. An alternative interpretation of the output is that the even numbered elements in the output array are the coefficients of the cosine terms and the odd numbered elements are the coefficients of the sine terms in a Fourier sine-cosine series expansion.

$$f_k = a_0 + a_1 \cos(Nkx/2) + \sum_{n=1}^{N/2-1} [a_{2n} \cos(nkx) + a_{2n+1} \sin(nkx)]. \quad (10.17)$$

The first element  $a_0$  is the zero frequency (DC) component and the second element  $a_1$  is the Nyquist frequency component. The Nyquist frequency is the highest frequency contained in the data due to the finite number of data points.

The FFTReal class in the numerics package computes the FFT of a real valued function and returns a complex array containing the positive frequency coefficients. The FFTRealApp program shown in Listing 10.15 tests the FFTReal class by transforming a sinusoidal function. The FFTRealPlotApp program (available on the CD) transforms and plots the Heaviside step function.

Listing 10.15 The FFTRealApp computes the Fourier transformation of a sinusoidal function.

```

1 package org.opensourcephysics.manual.ch10;
2 import org.opensourcephysics.frames.*;
3 import org.opensourcephysics.numerics.*;
4
5 public class FFTRealApp {
6     public static void main(String[] args) {
7         TableFrame frame = new TableFrame("Real FFT");
8         frame.setColumnNames(0, "frequency");
9         frame.setColumnNames(1, "cos");
10        frame.setColumnFormat(1, "#0.00");
11        frame.setColumnNames(2, "sin");
12        frame.setColumnFormat(2, "#0.00");
13        int n = 16;
```

```

14  double [] data = new double [ n ];
15  double
16      x = 0, dx = 1.0/n;
17  for (int i = 0; i < n; i++) {
18      data [ i ] = Math . sin (2*Math . PI*x);
19      x += dx;
20  }
21  FFTReal realFFT = new FFTReal (n);
22  realFFT . transform ( data );
23  double [] f = realFFT . getNaturalFreq (dx);
24  for (int i = 0; i < n; i += 2) {
25      frame . appendRow (new double [] {f [ i /2], data [ i ], data [ i +1]} );
26  }
27  frame . setVisible (true);
28  frame . setDefaultCloseOperation (javax . swing . JFrame . EXIT_ON_CLOSE );
29 }
30 }
```

## 10.8 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch10` package.

### **ArcsinApp**

`ArcsinApp` compares the interpolated inverse sine function to the exact result as described in Section 10.6.

### **CubicSplineApp**

`CubicSplineApp` implements a visualization of cubic spline interpolation. Cubic splines are described in Section 10.6.

### **FFTApp**

`FFTApp` tests the fast Fourier transform (FFT) as described in Section 10.7.

### **FFTPlotApp**

`FFTPlotApp` computes and plots the fast Fourier transform (FT) of the Heaviside step function as described in Section 10.7.

### **FFTRealApp**

`FFTRealApp` computes the Fourier coefficients of a real-valued function as described in Section 10.7.

**GaussianDerivativeApp**

`GaussianDerivativeApp` plots the derivative of a Gaussian function centered at the origin as described in Section 10.7.

**IntegralApp**

`IntegralApp` tests numerical integration algorithms in the `Integral` class. See Section 10.3.

**LagrangeInterpolatorApp**

`LagrangeInterpolatorApp` implements a visualization of Lagrange interpolating polynomials as described in Section 10.6.

**ParseComplexApp**

`ParseComplexApp` parses a string of characters to produce a complex-valued function that can be evaluated. This example uses the optional `JEParse` package developed by Nathan Funk.

**ParseRealApp**

`ParseRealApp` parses a string of characters to produce a math function that can be evaluated as described in Section 10.4.

**PolynomialApp**

`PolynomialApp` test the `Polynomial` class described in Section 10.6.

**RootApp**

`RootApp` compares Newton's method and the bisection method for finding the root of a function as described in Section 10.5.

**RootFinderApp**

`RootFinderApp` is a program for exploring the properties of root-finding algorithms. See Section 10.5.

**SlitTableApp**

`SlitTableApp` creates a table of values for the function  $f(x) = \sin x/x$ .

## CHAPTER

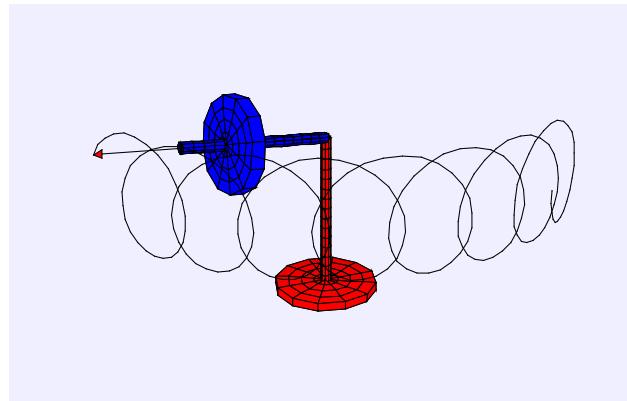
## 11

## Three Dimensional Modeling

©2005 by Wolfgang Christian and Francisco Esquembre, July 3, 2005

The Open Source Physics library defines a 3D API to create, organize and manipulate 3D worlds for physics simulations. Although high-level 3D packages such as Java 3D or JOGL can be used in this framework, familiarity with these packages is neither assumed nor required for the use of OSP 3D.

### 11.1 ■ OVERVIEW



**FIGURE 11.1** A top precessing due to an external torque.

The Open Source Physics 3D API uses high-level constructs to create, organize and manipulate 3D models. Because this API is defined in the `org.opensourcephysics.display3d.core` package using Java interfaces, it can be implemented using almost any 3D library. OSP 3D simplifies 3D concepts to aid programmers in the fast, accurate creation of physics simulations. Familiarity with add-on libraries is neither assumed nor required for the use of OSP 3D.

Programming 3D physics simulations presents two challenges. The first challenge is understanding the physics. Interesting three dimensional phenomena, such as the rigid-body dynamics of a spinning top shown in Figure 11.1, are mathematically challenging because they often require solving differential equations

## 11.2 Simple 3D

229

in reference frames that are attached to moving bodies. A physics-oriented 3D toolkit can help by providing facilities for transforming the dynamics to and from these non-inertial frames.

The second challenge is the visualization itself. Advanced 3D libraries such as Java 3D or Java bindings for OpenGL (JOGL) are full-featured 3D programming languages. A programmer can use these advanced libraries to create textures, morphs, and animations that rival the latest Hollywood special effects. A basic scene, however, takes numerous lines of code to create. A *virtual universe* must be created, then a *view* into that universe, then a *canvas 3D* object for rendering on-screen. The list goes on, and we can't even see anything because we have no light source yet. All of these objects are created one at a time because each is highly customizable. While this structural concept makes the Java 3D or JOGL libraries extremely versatile, scene creation can become tedious for beginning programmers. In many physics simulations we aren't concerned with these effects.

As users become familiar with 3D programming they can pick a particular implementation of OSP 3D API and use features in that implementation. We show you how to refine the appearance and put some special touches on a program using JOGL-specific features later in this chapter, but initially we just want to program interesting physics using the simple3d package.

## 11.2 ■ SIMPLE 3D

The OSP 3D API is defined in the `org.opensourcephysics.display3d.-core` package using Java interfaces and abstract classes. This API can be implemented using any library and we have implemented it in the `org.opensourcephysics.display3d.simple3d` package using only standard Java. We use the `simple3d` package to study the OSP 3D API because programs written using this implementation will run on any Java enabled computer. To use other OSP 3D implementations, one must simply install the Java library that communicates with the graphics hardware and import the corresponding OSP 3D package.

`Display3DFrame` in the `frames` package makes it easy to create a `simple3d` visualization as shown in Listing 11.1. This frame is composed of lower level components that will be described in subsequent sections.

Listing 11.1 The `Box3DApp` class creates a box within a `Display3DFrame`.

```

1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.display3d.simple3d.*;
3 import org.opensourcephysics.frames.*;
4 import java.awt.*;
5 import javax.swing.*;

6
7 public class Box3DApp {
8     public static void main(String[] args) {
9         // create a drawing frame and a drawing panel

```

```

10  Display3DFrame frame = new Display3DFrame("3D Demo");
11  frame.setPreferredMinMax(-10, 10, -10, 10, -10, 10);
12  frame.setDecorationType(VisualizationHints.DECORATION_AXES);
13  frame.setAllowQuickRedraw(false); // use shading when
14      rotating
15  Element block = new ElementBox();
16  block.setXYZ(0, 0, 0);
17  block.setSizeXYZ(6, 6, 3);
18  block.getStyle().setFillColor(Color.RED);
19  block.getStyle().setResolution(new Resolution(6, 6,
20      3)); // divide block into sub-blocks
21  frame.addElement(block);
22  frame.setVisible(true);
23  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24 }
}

```

It is not more difficult to create a 3D visualization than it is to create a 2D visualization because the `Display3DFrame` class behaves very much like its two-dimensional counterpart. The 2D `Drawable` interface has been replaced by the `Element` interface which is designed to mimic geometric shapes in the physical world. Elements have position and size properties and contain a `Style` object that controls an element's visual appearance such as color and resolution.

Some methods in the OSP 2D API, such as `setPreferredMinMax` have been extended by adding parameters for the third dimension. New objects, such as `Camera`, enable the programmer to control the three-dimensional viewing perspective.

```

1 // camera position can be set using Cartesian or polar
2     coordinates
3 frame.getCamera().setXYZ(0.0, 50.0, 0.0);
4 frame.getCamera().setAzimuthAndAltitude(0, Math.PI/4);

```

Three-dimensional elements, such as `ElementBox`, `ElementCylinder`, and `ElementEllipsoid`, are added to a container using the `addElement` method. Just as in the two-dimensional case, the high-level `Display3DFrame` is composed of lower level objects such as the `DrawingPanel3D` that fills the frame's viewing area. For changes to take effect, the program must draw the frame by invoking the `render` method. This method is invoked automatically by the `AbstractSimulation` class when the simulation is initialized and after every `doStep` method call.

Although OSP 3D is designed for three-dimensional visualizations, it also can show two-dimensional projections. These projections are available at runtime using the frame's menu. We can also project onto a coordinate plane by setting the projection mode programmatically:

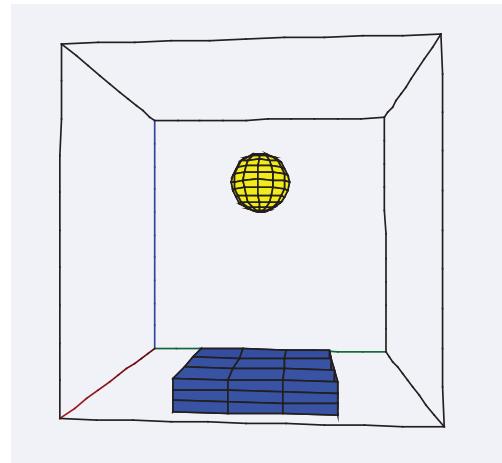
```

1 frame.getCamera().setProjectionMode(Camera.MODE_PLANAR_YZ); .

```

## 11.3 Drawing Panel 3D

231

**11.3 ■ DRAWING PANEL 3D****FIGURE 11.2** A 3D visualization of bouncing ball.

The `DrawingPanel3D` class defines a three-dimensional view with cartesian world coordinates as seen in Figure 11.2. This panel can be added to any `JFrame` but we usually place it into a `DrawingFrame3D` because this frame has been customized with a menu bar and other components that interact with a 3D panel.

```

1 // import a concrete implementation of OSP 3D
2 import org.opensourcephysics.display3d.simple3d.*;
3 // create a 3D world
4 DrawingPanel3D panel = new DrawingPanel3D();
5 DrawingFrame3D frame = new DrawingFrame3D();
6 frame.setDrawingPanel3D(panel);
7 // additional code places Elements into the 3D view

```

The `Display3DFrame` in the `frames` package that we introduced in Section 11.2 used the `simple3d` implementation of OSP 3D, and we now use this same package in order to study the API in detail.

Because 3D scenes are almost always animated and because a 3D drawing may be complex, the `DrawingPanel3D` class always uses buffering as described in Chapter 7. The 3D scene is drawn into a hidden image that has the same dimensions as the panel, and this image is later copied to the screen by the operating system. Because the panel keeps track of changes to its properties and to its 3D objects by setting an internal boolean variable named `dirtyImage`, a program need only invoke the panel's `repaint` method and the hidden image will be updated automatically. This updating works in the following manner. The event

dispatch thread invokes the `paintComponent` method, and this method quickly copies the background image to the screen. If the `dirtyImage` boolean is set, a timer event is initiated and, after a short delay, a new background image is rendered and copied. Multiple timer actions are combined (coalesced) into a single image update and this rendering is aborted if the scene is repainted by another thread such as an animation thread.

Although `repaint` will work, programmers should invoke the panel's `render` method from within an animation thread for smoother animation and consistent frame rates. An easy way to do this is to subclass `AbstractSimulation` and let the simulation thread's `run` method automatically call the `render` method for animated frames. We do not need to explicitly set the `animated` property in the `DrawingFrame3D` because it is set in the frame's constructor. Listing 11.2 creates a simple bouncing ball simulation using the simulation architecture developed in Chapter 7. This animation may gain energy because we don't detect the exact moment of the collision as described in Section 9.9.

Listing 11.2 A simple bouncing ball animation using the OSP 3D API.

```

1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.controls.AbstractSimulation;
3 import org.opensourcephysics.display3d.simple3d.*;
4
5 public class BounceApp extends AbstractSimulation {
6     DrawingPanel3D panel = new DrawingPanel3D();
7     DrawingFrame3D frame = new DrawingFrame3D(panel);
8     Element ball = new ElementEllipsoid();
9     Element floor = new ElementBox();
10    double velocity = 0;
11
12    public BounceApp() {
13        panel.setPreferredMinMax(-5, 5, -5, 5, 0, 10);
14        ball.setXYZ(0.0, 0.0, 9.0);
15        ball.setSizeXYZ(2, 2, 2);
16        ball.getStyle().setFillColor(java.awt.Color.YELLOW);
17        // floor
18        floor.setXYZ(0.0, 0.0, 0.0);
19        floor.setSizeXYZ(5.0, 5.0, 1.0);
20        // Add the objects to panel
21        panel.addElement(ball);
22        panel.addElement(floor);
23        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
24        frame.setVisible(true);
25    }
26
27    protected void doStep() {
28        double
29        z = ball.getZ(), dt = 0.05;
30        z += velocity*dt; // moves the ball

```

## 11.4 Cameras

233

```

31   velocity -= 9.8*dt; // acceleration changes the velocity
32   if(z<=1.0&&velocity <0) {
33     velocity = -velocity;
34   }
35   ball.setZ(z);
36 }
37
38 public static void main(String[] args) {
39   (new BounceApp()).startSimulation();
40 }
41 }
```

DrawingPanel3D has a number of built in visual “hints” that help users orient their view of the 3D scene. There are, for example, axis decorations that show the cartesian coordinate directions using arrows or using a rectangular bounding box. Red, green, and blue sides on the cube indicate the x-, y-, and z-directions.

```

1 // decoration types are: DECORATION_NONE, DECORATION_AXES,
2 // DECORATION_CUBE
3 panel.getVisualizationHints().
4   setDecorationType(VisualizationHints.DECORATION_CUBE);
```

The corners of the decoration cube show the locations of the world’s minimum and maximum values. These values are set as follows:

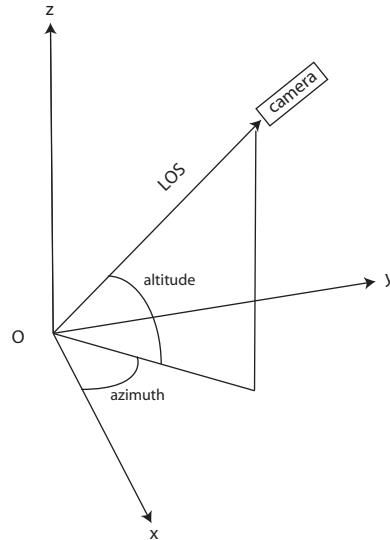
```
1 panel.setPreferredMinMax( -5.0, 5.0, -5.0, 5.0, 0.0, 10.0);
```

Some visualization hints affect appearance and performance. If the scene is simple, it may not be necessary to switch to wire-mesh rendering when dragging with the mouse to position the camera. This quick-draw mode can be disabled if the scene is simple.

```
1 panel.getVisualizationHints().setAllowQuickRedraw(false);
```

## 11.4 ■ CAMERAS

We view a 3D world through a Camera that has a location and points in a direction known as the line of sight (LOS). The default LOS points from the camera toward the center of the drawing as determined by the average of the extrema along each coordinate. This makes it easy to rotate the camera around a physical model that has been created near the center without loosing it. Click-dragging left to right rotates the azimuthal angle and click-dragging up to down rotates the altitude angle as shown in Figure 11.3. The default camera location is on the x-axis with a LOS along the axis toward the center. The location on the axis depends on the view’s extremum values.



**FIGURE 11.3** A Camera determines the projection of an OSP 3D scene onto a 2D view. The default line of sight (LOS) points from the camera toward the center of the drawing.

Although most users will adjust a 3D view using the mouse, it is straightforward to set the camera programmatically. Camera properties are set using accessor methods after obtaining a reference to the camera from the 3D panel:

```

1 Camera camera = (Camera) panel.getCamera();
2 camera.setPosition(x,y,z);
3 // an alternate to the above
4 camera.setAzimuthAndAltitude(theta,phi);
```

The `setPosition` method sets the camera's position in space using cartesian coordinates. The `setAzimuthAndAltitude` method positions the camera using spherical polar coordinates. The azimuth is the rotation angle about the z-axis and the altitude is the angle above or below the x-y plane.

Just as a physical camera can have telephoto and wide angle lenses that accentuate or diminish the three-dimensional perspective, our computer model can project the three dimensional world onto the computer screen a number of ways. Because a camera uses a mathematical transformation to map (project) a three dimensional space onto a two dimensional plane, these options are accessed using the `Camera` interface. (See Section 11.8 for a discussion of transformations.)

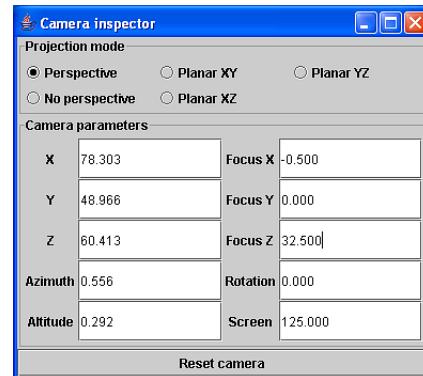
```

1 // projection modes are: MODE_PERSPECTIVE, MODE_NO_PERSPECTIVE,
2 // MODE_PLANAR_XY, MODE_PLANAR_XZ, and MODE_PLANAR_YZ
3 panel.getCamera().setProjectionMode(Camera.MODE_PLANAR_YZ);
```

## 11.5 Elements and Groups

235

The camera's `getTransformation` method allows a program to project points from the world to screen coordinates should you need this low-level functionality.



**FIGURE 11.4** The camera inspector can be used to adjust the viewing position.

Although adjusting the camera location is fairly intuitive using the mouse, a `CameraInspector` is available from the menu (see Figure 11.4) and can be used for precise positioning. Cartesian and angle coordinates are, of course, interrelated so setting a value can affect the other values. The `CameraApp` test program available on the CD may also be helpful in understanding the camera's API.

## 11.5 ■ ELEMENTS AND GROUPS

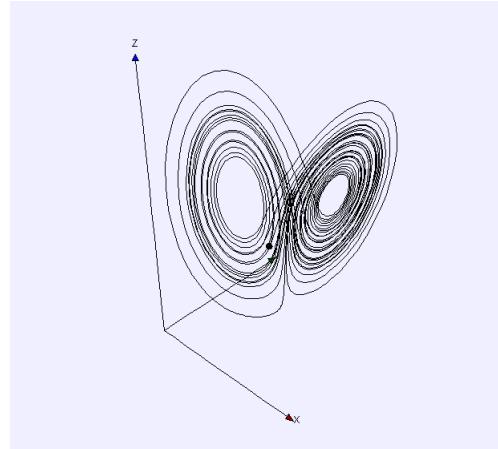
A 3D simulation creates objects that implement the `Element` interface and adds these objects to a 3D container. The `ElementApp` program on the CD creates a 3D scene containing common 3D elements. The bouncing ball program shown in Listing 11.2 uses elements in a simulation. Objects are instantiated, added to a panel, and moved using a thread. Because `BounceApp` extends `AbstractSimulation` and the frame's `animated` property is set, it is not necessary to call the `render` method after objects have been moved.

Although specifying the orientation (rotation) of an element requires a modest amount of mathematics and will be described more fully in Section 11.8, sizing and positioning an element is easy using accessor methods such as `setXYZ` and `setSizeXYZ`.

```

1 Element box = new ElementBox();
2 box.setXYZ(10,10,0);           // position of center
3 box.setSizeXYZ(4, 4, 1);        // lengths of sides
4 Element arrow = new ElementArrow();
5 arrow.setZ(5);                 // x and y remain at zero
6 arrow.setSizeXYZ(4, 4, 1);      // components along axes

```



**FIGURE 11.5** A 3D view of the Lorenz attractor.

The position of a symmetric object, such as a box, is its center of mass, and the position of an arrow is its tail.

An element contains a `Style` object that determines visual properties such as color. These properties are set as follows:

```

1 Style style = (Style) box.getStyle();
2 style.setFillColor(Color.RED);
3 style.setLineColor(Color.BLACK);
4 style.setLineWidth(3);
5 style.setResolution( new Resolution(5,5,2)); // divides box in
      sub-blocks.

```

Most properties are self explanatory but the resolution property needs explanation because it is implementation dependent.

Three-dimensional objects are constructed using polygons, and it is sometimes difficult deciding how many polygons to use. The `Resolution` class stores information that can help an Element divide itself into smaller pieces that have a size no larger than max units. The precise meaning of the parameters passed into the `Resolution` constructor is left to each element and to each 3D implementation, but a parameter typically specifies the number of divisions in a coordinate direction. Arrows and lines use a `Resolution` object with a single parameter that specifies how many times they should divide themselves along their length.

Because the `simple3d` package orders all polygons from far to near and then paints the polygons in that order, you may notice rendering artifacts when objects constructed with large polygons intersect. Increasing the resolution will reduce these artifacts although too high a resolution will increase rendering time. If truly high resolution images are needed, it is best to use a 3D implementation such as JOGL or Java 3D that renders and smooths polygons using the graphics card.

**TABLE 11.1** Elements defined in the OSP 3D API.

<b>org.opensourcephysics.display3d.core</b>	
ElementArrow	An arrow with its tail located at its position. The <code>setSizeXYZ</code> method sets the arrow's components.
ElementBox	A box. The top and bottom can be open or closed.
ElementCircle	A circle that is drawn without any subdivisions or perspective.
ElementCone	A cone. The <code>setSizeXYZ</code> method sets the dimensions of the elliptical base and the height.
ElementEllipsoid	An ellipsoid that is drawn with subdivisions and perspective.
ElementImage	An image that is always drawn (projected) perpendicular to the line of sight.
ElementPlane	A plane.
ElementPolygon	A polygon that is created from an array of data points. A polygon can be closed (solid) or open (polyline).
ElementSegment	A straight line segment similar to arrow but without an arrowhead.
ElementSphere	An ellipsoid whose dimensions are of equal size. Changing any one dimension changes all other dimensions.
ElementSpring	A coil spring.
ElementSurface	A wire-mesh surface created from an array of data points.
ElementText	Text that can be justified left, right, and center with respect to the position.
ElementTrail	A trail of points. The maximum number of points that can be stored can be set.
Group	An element that is made up of other elements.

Almost all modern graphics cards support hardware accelerated rendering using OpenGL graphics language drivers.

The Lorenz class uses these ideas to show a three dimensional graphical model of a two-dimensional layer of fluid that is heated from below. Using a system of differential equations, the model has three state variables that are referred to as  $x$ ,  $y$ , and  $z$  although they have nothing to do with physical space. The  $x$  variable is a measure of the fluid flow velocity circulating around the cell,  $y$  is a measure of the temperature difference between the rising and falling fluid regions, and  $z$  is a measure of the difference in the temperature profile between the bottom and the top from the normal equilibrium temperature profile. The evolution of the fluid in

this state space is defined by (11.1).

$$\frac{dx}{dt} = -\sigma x + \sigma y \quad (11.1a)$$

$$\frac{dy}{dt} = -xz + rx - y \quad (11.1b)$$

$$\frac{dz}{dt} = xy - bz, \quad (11.1c)$$

Typical input parameters for the Lorenz model are  $\sigma = 10$ ,  $b = 8/3$ , and  $r = 28$  and the initial condition  $x_0 = 1$ ,  $y_0 = 1$ ,  $z_0 = 20$ . The `getRate` and `getState` methods in the `Lorenz` class implement the `ODE` interface as described in Chapter 9.

The visualization shows the current  $(x, y, z)$  value as a small red ellipsoid and past positions as a trail. These objects are instantiated in the model's constructor and added to a `DrawingPanel3D`. The model solves the differential equations and uses the `trail.addPoint(x, y, z)` method to create the trail and the `ball.setXYZ(x, y, z)` method to update the ellipsoid.

The `LorenzApp` class does the usual housekeeping by extending `AbstractSimulation` to create a simulation framework and instantiating a Lorenz model. It is available on the CD but is not shown here because it is similar to other simulations.

A `Group` is a special `Element` that allows us to combine multiple 3D elements to form a compound object. Because a `Group` is an `Element` it acts like a single 3D object. Elements placed within a group are positioned and sized relative to their group's position and size. In addition, transforming a group (see Section 11.8) transforms every element within the group.

An easy way to create a compound object is to extend the 3D `Group` class and create the objects in the constructor. You should not, however, add the same `Element` to a group and a `DisplayPanel3D` or to two groups but you can nest groups within groups. The `GroupApp` program on the CD demonstrates the use of a group by creating a large sphere and ten smaller spheres that rotate together.

## 11.6 ■ INTERACTIONS

Because users may wish to interact with a three-dimensional model in many different ways, we have defined a more sophisticated interaction API for 3D objects. This framework consists of the following interfaces in the `org.opensource-physics.display3d.core.interaction` package.

- `InteractionSource` The interface for objects that can generate interaction events and send them to listeners.
- `InteractionListener` The interface for objects that can listen to interaction events.

## 11.6 Interactions

239

Listing 11.3 An implementation of the Lorenz model for convection in a two-dimensional fluid.

```

1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.display3d.simple3d.*;
3 import org.opensourcephysics.numerics.*;
4
5 public class Lorenz implements ODE {
6     DrawingPanel3D panel = new DrawingPanel3D();
7     DrawingFrame3D frame = new DrawingFrame3D(panel);
8     double[] state = new double[4];
9     double a = 28.0;
10    double b = 2.667;
11    double c = 10.0;
12    ODESolver ode_solver = new RK45MultiStep(this);
13    Element ball = new ElementEllipsoid();
14    ElementTrail trace = new ElementTrail();
15
16    public Lorenz() {
17        frame.setTitle("Lorenz Attractor");
18        panel.setPreferredMinMax(-15.0, 15.0, -15.0, 15.0, 0.0,
19            50.0);
20        panel.getVisualizationHints().setDecorationType(
21            VisualizationHints.DECORATION_AXES);
22        ball.setSizeXYZ(1, 1, 1);
23        ball.getStyle().setFillColor(java.awt.Color.RED);
24        panel.addElement(trace);
25        panel.addElement(ball);
26        ode_solver.setStepSize(0.01);
27    }
28
29    protected void doStep() {
30        for(int i = 0;i<10;i++) {
31            ode_solver.step();
32            trace.addPoint(state[0], state[1], state[2]);
33            ball.setXYZ(state[0], state[1], state[2]);
34        }
35        panel.setMessage("t="+Util.f3(state[3]));
36    }
37
38    public double[] getState() {
39        return state;
40    }
41
42    public void initialize(double x, double y, double z) {
43        state[0] = x;
44        state[1] = y;
45        state[2] = z;
46        state[3] = 0; // time
47        trace.clear();
48        trace.addPoint(x, y, z);
49        ball.setXYZ(x, y, z);
50        frame.setVisible(true);
51    }
52
53    public void getRate(double[] state, double[] rate) {
54        rate[0] = -(state[0]-state[1])*c; // x rate
55        rate[1] = -state[1]-state[0]*state[2]+state[0]*a; // y rate
56        rate[2] = (state[0]*state[1]-state[2])*b; // z rate
57        rate[3] = 1; // time
58    }
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999

```

- **InteractionEvent** The class for the event generated when the user interacts with a source.
- **InteractionTarget** The interface for the particular target that has been hit in a source.

`BallInteractionApp` shown in Listing 11.4 demonstrates how this framework operates by enabling an interaction and providing a listener that restricts dragging to the x-y plane. The constructor registers the program as the listener and enables the ball's position interaction target. The `interactionPerformed` method implements the `InteractionListener` interface. Note that the listener sets the ball's z-coordinate to zero in order to disable dragging in this direction. If we wish to drag the particle in x, y, and z, we would still enable the position target but we could remove the implementation of `InteractionListener` so as not to intervene in the 3D panel's positioning of the ball.

Listing 11.4 A simple program that allows users to drag a ball.

```

1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.display3d.core.interaction.*;
3 import org.opensourcephysics.display3d.simple3d.*;
4
5 public class BallInteractionApp implements InteractionListener {
6     DrawingPanel3D panel = new DrawingPanel3D();
7     DrawingFrame3D frame = new DrawingFrame3D(panel);
8     Element ball = new ElementEllipsoid();
9
10    public BallInteractionApp() {
11        ball.addInteractionListener(
12            this); // sends interactions to this object
13        // enables interactions that change positions
14        ball.getInteractionTarget(Element.TARGET_POSITION).setEnabled(
15            true);
16        panel.addElement(ball);
17        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
18        frame.setVisible(true);
19    }
20
21    public void interactionPerformed(InteractionEvent _event) {
22        switch(_event.getID()) {
23            case InteractionEvent.MOUSE_DRAGGED :
24                System.out.println("Ball x="+ball.getX());
25                System.out.println("Ball y="+ball.getY());
26                ball.setZ(0); // always set z to zero
27                break;
28            default :
29                break;
30        }
31    }

```

```

32
33 public static void main(String[] args) {
34     new BallInteractionApp();
35 }
36 }
```

### Interaction Source

An object that implements the `InteractionSource` interface is an object that can generate interaction events in response to user actions and report these events to a list of registered interaction listeners. The implementing class typically keeps an internal list of the listeners that will be informed whenever an action event takes place. A listener only appears once in this list even if it has been added multiple times.

A source can contain one or more targets, that is, hot spots that trigger interaction events. (See `InteractionTarget` below for details.) For example, an arrow has targets that enable a listener to move the arrow when you drag its tail and to resize the arrow when you drag its head. The `InteractionEvent` contains information about which target was selected by the user.

`InteractionSource` defines the following methods:

```

1 public interface InteractionSource {
2     InteractionTarget getInteractionTarget (int target);
3     void addInteractionListener (InteractionListener listener);
4     void removeInteractionListener (InteractionListener listener);
5 }
```

The first method takes an integer that identifies the access to one of the targets. Sources should document their targets and provide integer numbers to reference them. The second and third methods add and remove interaction listeners to the source.

Notice that it is customary that source interactions are disabled by default. This helps avoid undesired interaction events. Notice also that adding a listener doesn't enable the target(s) of a source. The user must activate them explicitly using the target's `setEnabled` method.

### Interaction Listener

`InteractionListener` is an interface that defines a single method. Classes implementing this interface need to register themselves with an `InteractionSource` using the `addInteractionListener` method.

```

1 public interface InteractionListener {
2     void interactionPerformed(InteractionEvent event);
3 }
```

A source will call the `interactionPerformed` method whenever an interaction occurs. The provided event includes relevant information about the interaction that can be used by the listener to determine what to do in response.

### Interaction Event

An `InteractionEvent` passes information to an `InteractionListener` that describes what interaction has taken place with the `InteractionSource` and which target has been hit. This class has six public named constants and two public methods.

```

1  public class InteractionEvent extends ActionEvent {
2
3      static public final int MOUSE_PRESSED;
4      static public final int MOUSE_DRAGGED;
5      static public final int MOUSE_RELEASED;
6      static public final int MOUSE_ENTERED;
7      static public final int MOUSE_EXITED;
8      static public final int MOUSE_MOVED;
9
10     public Object getInfo();
11     public MouseEvent getMouseEvent();
12 }
```

`InteractionEvent` is a subclass of `java.awt.event.ActionEvent`. It adds an `Object` that contains information about the target that has been hit to the event. Sources should document how the information in the passed object should be accessed.

The `InteractionEvent` superclass defines the `getID` method. Elements set the ID to one of the named constants with obvious meanings. Notice how the ID is accessed in `BallInteractionApp`.

### Interaction Target

The `InteractionTarget` interface is the most technical of the interfaces in this subsection because sources can have more than one target and because sources can be nested within groups. A target is a hot spot in the source that is sensible to user interaction and is identified by a named constant such as `TARGET_POSITION` or `TARGET_SIZE`.

The `InteractionTarget` interface differentiates what target in the source was hit and helps the target affect the underlying source during the interaction. This interface is not listed here because we do not show you how to define your own interactive targets. The `setEnabled` method is the most important. This method enables us to activate a target within an `Element` so that the object can respond to mouse actions.

```
1 element.getInteractionTarget(Element.TARGET_POSITION).setEnabled(true);
```

## 11.6 Interactions

243

The `setAffectsGroup` method sets the scope of an interaction when an element belongs to a group.

```
1 element.getInteractionTarget(Element.TARGET_POSITION).setAffectsGroup(true);
```

If the user drags the element then the entire group will move. You can, for example, build a cart consisting of a block and two wheels and then move the entire cart by dragging the block.

### Interaction Example

Users will typically deal with interaction by implementing the `InteractionListener` interface in one or more objects and adding these listeners to one or more `InteractionSources`. Interaction sources are `DrawingPanel3D` and `Element` objects within the panel. The elements within a panel will report mouse actions that hit any of their possible targets and `DrawingPanel3D` reports actions that take place on its empty space. The `getInfo` method returns null if the user rotates or zooms the camera by dragging the mouse and holding the control or shift keys, respectively. The user can, however, select a point in 3D space by holding the “Alt” key in which case the `getInfo` method returns a `double[3]` array identifying the point.

Listing 11.5 shows how a listener can discriminate between interaction on the different parts of a scene. Notice, in particular, how the listener distinguishes between the panel and the objects within the panel.

Listing 11.5 `Interaction3DApp` demonstrates the OSP 3D interaction framework.

```
1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.display3d.core.interaction.*;
3 import org.opensourcephysics.display3d.simple3d.*;
4 import org.opensourcephysics.frames.*;

5
6 public class Interaction3DApp implements InteractionListener {
7     Element particle = new ElementCircle();
8     ElementArrow arrow = new ElementArrow();

9
10    Interaction3DApp() {
11        Display3DFrame frame = new Display3DFrame("3D Interactions");
12        frame.setPreferredMinMax(-2.5, 2.5, -2.5, 2.5, -2.5, 2.5);
13        particle.setSizeXYZ(1, 1, 1);
14        particle.getInteractionTarget(Element.TARGET_POSITION).setEnabled(
15            true); // enables interactions that change positions
16        particle.addInteractionListener(
17            this); // accepts interactions from the particle
18        frame.addElement(particle); // adds the particle to the panel
19        arrow.getInteractionTarget(Element.TARGET_SIZE).setEnabled(
20            true); // enables interactions that
                  change the size
```

```

21    arrow.addInteractionListener(
22        this); // accepts interactions from the arrow
23    frame.addElement(arrow); // adds the arrow to the panel
24    frame.enableInteraction(
25        true); // enables interactions with the 3D
26        Frame
27    frame.addInteractionListener(
28        this); // accepts interactions from the frame's
29        DrawingPanel3D
30    frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
31    frame.setVisible(true);
32}
33
34public void interactionPerformed(InteractionEvent evt) {
35    Object source = evt.getSource();
36    if(evt.getID()
37        ==InteractionEvent
38            .MOUSE_PRESSED) { // check for a particular mouse
39                action
40                System.out.println("Mouse Pressed");
41            }
42        if(source==particle) { // check for a particular element
43            System.out.println("A particle has been hit");
44        }
45    }
46
47}

```

## 11.7 ■ VECTORS

Vector operations can be performed in more than one way in Java. A simple and straightforward way is to represent the components of an n-dimensional vector using an n-component array of numbers. Another approach is to define a class that stores components as instance variables. Both representations are useful. The first is easy to use while the second provides better encapsulation and is slightly faster when doing arithmetic because instance variables are accessed more quickly than array elements. The array-based implementation uses static methods and is defined in the VectorMath class and the object-based implementation uses instance variables and is defined in the Vec3D class. The Vec3D constructor simply copies the components and is coded as follows:

```

1 double x, y, z; // instance variables
2

```

## 11.7 Vectors

245

**TABLE 11.2** Vector operations for 3D modeling are defined in the numeric package.

---

**org.opensourcephysics.numerics.VectorMath**  
**org.opensourcephysics.numerics.Vec3D**


---

cross2d	Computes the cross product of two 2D vectors. The value returned is the component of the vector that is perpendicular to the given vectors.
cross3d	Computes the cross product of two 3D vectors.
dot	Computes the dot product of two vectors.
magnitude	Computes the magnitude (length) of a vector.
magnitudeSquared	Computes the magnitude squared of a vector.
normalize	Normalizes a vector.
perp	Computes the perpendicular part of one vector with respect to another vector.
plus	Adds two vectors.
project	Computes the projection of one vector onto another vector.

---

```

3 | public Vec3D(double x, double y, double z) { // constructor
4 |     this.x = x;
5 |     this.y = y;
6 |     this.z = z;
7 | }

```

The dot (Euclidian inner) product of n-dimensional vectors **a** and **b** is a scalar *c* such that

$$c = \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta \quad (11.2)$$

where  $\theta$  is the angle between the vectors. This expression is equivalent to

$$c = \sum_{i=0}^{n-1} a_i b_i \quad (11.3)$$

where each vector is expressed using components  $\mathbf{a} = (a_0, a_1, \dots, a_{n-1})$  in a Cartesian basis set. The static dot method in the VectorMath class computes the dot product between arrays of numbers.

```

1 | static public double dot(double[] a, double[] b) {
2 |     int aLength = a.length;
3 |     if(aLength != b.length) {
4 |         throw new UnsupportedOperationException("ERROR:
5 |             Vectors must be of equal dimension in dot
6 |             product.");
7 |     }
8 |     double sum = 0;

```

```

8   for (int i = 0; i < aLength; i++) {
9     sum += a[i]*b[i];
10    }
11   return sum;
12 }
```

Because objects (other than arrays) are not involved, the `VectorMath` class is easy to use. A dot product, for example, is computed as follows:

```

1 double [] a = new double [] {1,2,3};
2 double [] b = new double [] {4,5,6};
3 double c = VectorMath. dot(a,b);
```

The `magnitude` method computes the vector magnitude (length) by taking the square root of the sum of the squares of the vector components.

$$|a| = \left( \sum_{i=0}^{n-1} a_i^2 \right)^{1/2} \quad (11.4)$$

There is also a `magnitudeSquared` method that returns the dot product of a vector with itself. This method is useful when doing comparison between lengths. Computing the square root is computationally expensive and often unnecessary because  $|a| > |b|$  if  $a^2 > b^2$ .

The following code fragment computes a dot product using the `Vec3D` class.

```

1 Vec3D a = new Vec3D(1,2,3);
2 Vec3D b = new Vec3D(4,5,6);
3 double c = a. dot(b);
```

Because this is conceptually similar to code shown for the `VectorMath` class, we will usually only show code for the `VectorMath` class.

The projection of vector **a** onto vector **b** is a vector with magnitude  $\mathbf{a} \cdot \mathbf{b}$  in the direction of **b**.

$$\mathbf{a}_{\parallel b} = \frac{\mathbf{a} \cdot \mathbf{b}}{|b|^2} \mathbf{b} \quad (11.5)$$

The component of **a** that is perpendicular to vector **b** is useful and can be computed by subtracting  $\mathbf{a}_{\parallel b}$  from **a**.

$$\mathbf{a}_{\perp b} = \mathbf{a} - \frac{\mathbf{a} \cdot \mathbf{b}}{|b|^2} \mathbf{b} \quad (11.6)$$

These operations are implemented in the `project` and `perp` methods in the `VectorMath` class.

The cross (vector) product of 3-dimensional vectors **a** and **b** is another vector **c** that is perpendicular to **a** and **b** and has a magnitude *c* given by

$$c = |\mathbf{a} \times \mathbf{b}| = |a| |b| \sin \theta \quad (11.7)$$

## 11.8 Transformations

247

where  $\theta$  is again the angle between the two vectors. The cross product can be evaluated as follows using vector components:

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - b_y a_z, a_z b_x - b_z a_x, a_x b_y - b_x a_y). \quad (11.8)$$

The cross product is often used to create a new vector that is perpendicular (orthogonal) to two others.

## 11.8 ■ TRANSFORMATIONS

A function whose domain is a n-dimensional space and whose range is a m-dimensional space is known as a *transformation*. In the context of three-dimensional simulation and visualization, the domain and the range are arrays of numbers representing coordinates along a set of spacial axes (basis set). This is expressed in standard mathematical notation as  $\mathcal{T} : \mathcal{R}^n \rightarrow \mathcal{R}^m$ . In the current context, we transform a 3D point into another 3D point so  $m = n = 3$ .

The Transformation interface in the numerics package abstracts these mathematical concepts using one-dimensional arrays of numbers.

```

1 package org.opensourcephysics.numerics;
2 public interface Transformation extends Cloneable {
3     public Object clone();
4     public double[] direct (double[] point);
5     public double[] inverse (double[] point) throws
6         UnsupportedOperationException;
}
```

The `direct` and `clone` methods are guaranteed to succeed, but the `inverse` method may fail and then throw an `UnsupportedOperationException`. The `direct` method transforms the values and returns the given array object; the `clone` method creates a copy of the `Transformation` object.

Creating a new object using the `clone` method is an important Java concept. The `clone` method must insure that the new object's internal fields (variables) contain the same values as the original object and that objects within the original object are also cloned. If an object's `clone` method satisfies this criterion, then it can declare itself to be `Cloneable`. The following code fragment shows how the `clone` method is used to create a copy of a point and return transformed values in that point.

```

1 // transformation implements the Transformation interface;
2 // array is of type double[]
3 double[] transformedArray = transformation( (double[])
4     array.clone());
```

Translation class shown in Listing 11.6 is a concrete example of a `Transformation` that adds an offset to each coordinate value.

$$\mathcal{T}(x, y, z) \rightarrow (x + a, y + b, z + c) \quad (11.9)$$

The `main` method applies this transformation to a 3D box.

Listing 11.6 An implementation of the `Transformation` interface.

```

1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.display3d.simple3d.*;
3 import org.opensourcephysics.numerics.Transformation;
4
5 public class Translation implements Transformation {
6     double a, b, c;
7
8     public Translation(double a, double b, double c) {
9         this.a = a;
10        this.b = b;
11        this.c = c;
12    }
13
14    public Object clone() {
15        return new Translation(a, b, c);
16    }
17
18    public double[] direct(double[] point) { // assumes point is
19        double [3]
20        point[0] += a;
21        point[1] += b;
22        point[2] += c;
23        return point;
24    }
25
26    public double[] inverse(
27        double[] point) throws UnsupportedOperationException {
28        point[0] -= a;
29        point[1] -= b;
30        point[2] -= c;
31        return point;
32    }
33
34    static public void main(String args[]) {
35        Element box = new ElementBox();
36        Transformation translation = new Translation(2, -4, 1);
37        box.setTransformation(translation);
38        System.out.println("box x=" + box.getX());
39    }
}

```

Run the `Translation` program, and notice that the x-coordinate of the box has not changed the default value of zero. The transformation changes the drawing geometry but not the position and size of the `Element`. OSP 3D elements clone the given transformation whenever their `setTransformation` method is

## 11.9 Matrix Transformations

249

invoked. Thus, changing the original transformation has no effect unless a new `setTransformation` is invoked again.

Although the concept of a transformation is very general, we need it most often to rotate and translate points and vectors in three-dimensional space. These transformations are examples of *linear transformations* because the transformation associates with addition  $\mathbf{v}_1 + \mathbf{v}_2$  and commutes with scalar multiplication.

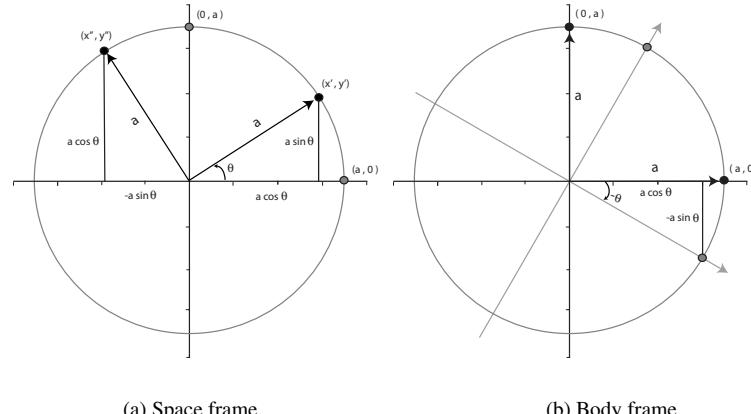
$$\mathcal{T}(\mathbf{v}_1 + \mathbf{v}_2) = \mathcal{T}(\mathbf{v}_1) + \mathcal{T}(\mathbf{v}_2) \quad (11.10a)$$

$$\mathcal{T}(a\mathbf{v}_1) = a\mathcal{T}(\mathbf{v}_1). \quad (11.10b)$$

where  $\mathbf{v}_1$  and  $\mathbf{v}_2$  are points or vectors in three-dimensional space.

Three-dimensional transformations can be described in many different ways including spoken language: “ Rotate the cube by thirty degrees about the y-axis.” Physicists and mathematicians prefer a mathematical description of this process. Two common implementations are rotation matrices and quaternions and the OSP numerics package contains both implementations. Users should select whichever implementation fits their model. Because many physics texts solve dynamics problems using a sequence of rotations about three cartesian axes (Euler angles), we’ll study rotation matrices first.

## 11.9 ■ MATRIX TRANSFORMATIONS



**FIGURE 11.6** The effect of rotation about the z-axis can be derived using simple trigonometry. The figure shows two points being rotated through an angle  $\theta$  as seen in the (a) space and (b) body frames.

A matrix can be used to implement the `Transformation` interface using the usual rules for matrix multiplication:

$$\begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{bmatrix} = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (11.11)$$

where

$$\begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{bmatrix} = \begin{bmatrix} xm_{0,0} + ym_{0,1} + zm_{0,2} \\ xm_{1,0} + ym_{1,1} + zm_{1,2} \\ xm_{2,0} + ym_{2,1} + zm_{2,2} \end{bmatrix}. \quad (11.12)$$

This can be written compactly as

$$\bar{\mathbf{v}} = \mathcal{M}\mathbf{v} \quad (11.13)$$

where  $\mathcal{M}$  represents the entire matrix and  $\mathbf{v}$  is again a point or a vector.

Rotation transformations can be expressed in matrix form using simple trigonometry as shown in Figure 11.6. Rotating an object through an angle  $\theta$  (with respect to the origin) transforms a point  $(a, 0)$  into a new point  $(x', y')$ . The coordinates of this point are now  $(a \cos \theta, a \sin \theta)$  with respect to the original axes. Similarly, a point on the y-axis  $(0, a)$  transforms into a point  $(x'', y'')$  with coordinates  $(-a \sin \theta, a \cos \theta)$ .

An alternative view of this process is to think of the transformation as a change of coordinate system rather than as a rotation of the object. The frame of reference attached to the object  $(\bar{x}, \bar{y})$  is referred to as the body frame and the original coordinate system  $(x, y)$  is referred to as the space frame. The coordinates of a given point of the body (such as  $(a, 0)$ ) do not change in the body frame, but they change (in this case to  $(a \cos \theta, -a \sin \theta)$ ) with respect to the space frame.

The transformation from the space frame to the body frame due to a rotation about the z-axis can be expressed as a matrix as follows:

$$\mathcal{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (11.14)$$

where

$$\bar{\mathbf{v}} = \mathcal{R}_z\mathbf{v}. \quad (11.15)$$

The inverse transformation (body to space) is almost the same except that the direction of rotation (sign of  $\theta$ ) changes thereby changing the sign of the  $\sin \theta$  terms.

$$\mathcal{R}_z^{-1}(\theta) = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (11.16)$$

## 11.10 Angle-Axis Representation

251

The  $\mathcal{R}_z$  matrix has a number of very important properties that will be true for all rotation matrices. These properties are:

- The dot product of any row or column with itself is one. Matrices that have this property are said to be normalized.
- The dot product of any row (column) with a different row (column) is zero. Matrices that have this property are said to be orthogonal.
- Every column of  $\mathcal{R}_z$  represents the coordinates of a unit vector along a rotated coordinate axis. In other words, column one is the transformed unit vector  $(1, 0, 0)$ , column two is the transformed unit vector  $(0, 1, 0)$ , and column three is the transformed unit vector  $(0, 0, 1)$ . Note that the unit vector along the z-axis (column three) remains unchanged because the rotation is about the z-axis.
- The off-diagonal elements of the matrix change sign if the row and column indices are reversed  $m_{i,j} = m_{j,i}$  where  $i \neq j$ .
- The transpose of the rotation matrix  $\mathcal{R}_z^t$  is the matrix inverse so that  $\mathcal{R}_z \mathcal{R}_z^t = 1$ . This property is easy to prove using the first two properties and the usual rules of matrix multiplication.

Additional rotation matrices about the other axes can be derived. The rotation about the x axis through an angle  $\theta$  is

$$\mathcal{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix} \quad (11.17)$$

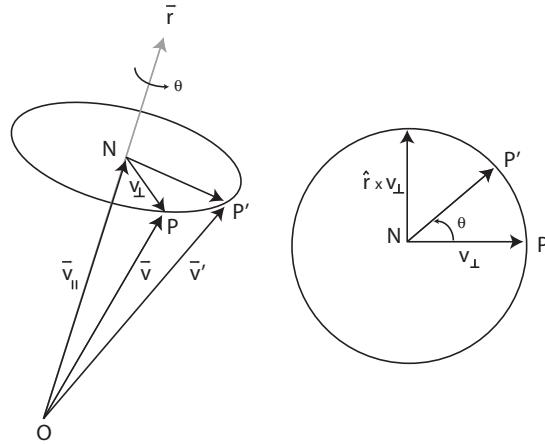
and a rotation about the y axis through an angle  $\theta$  is

$$\mathcal{R}_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix} \quad (11.18)$$

## 11.10 ■ ANGLE-AXIS REPRESENTATION

Euler showed that the orientation of a rigid body can be expressed as a rotation by an angle about a fixed axis. This construction of a rotation is known as the *angle-axis* representation. Although rotations about one of the coordinate axes are easy to derive and can be combined using the rules of linear algebra to produce such an arbitrary rotation, the derivation of the rotation matrix for a rotation about the origin by an angle  $\theta$  around an axis (unit length direction vector) with arbitrary orientation  $\hat{\mathbf{r}}$  is not hard to derive.

The strategy is to decompose the vector  $\mathbf{v}$  (or point  $P$ ) that will be rotated into components that are parallel and perpendicular to the direction  $\hat{\mathbf{r}}$  as shown



**FIGURE 11.7** A rotation of a vector  $\mathbf{v}$  about an axis  $\hat{\mathbf{r}}$  to produce  $\mathbf{v}'$  can be decomposed into parallel  $\mathbf{v}_{\parallel}$  and perpendicular  $\mathbf{v}_{\perp}$  components.

in Figure 11.7. The parallel part  $\mathbf{v}_{\parallel}$  does not change while the perpendicular part  $\mathbf{v}_{\perp}$  is a two-dimensional rotation in a plane perpendicular to  $\hat{\mathbf{r}}$ . The parallel part is the projection of  $\mathbf{v}$  onto the unit vector  $\hat{\mathbf{r}}$

$$\mathbf{v}_{\parallel} = (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}, \quad (11.19)$$

and the perpendicular part is what remains of  $\hat{\mathbf{v}}$  after we subtract the parallel part

$$\mathbf{v}_{\perp} = \mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}. \quad (11.20)$$

To calculate the rotation of  $\mathbf{v}_{\perp}$ , we need two perpendicular basis vectors in the plane of rotation. If we use  $\mathbf{v}_{\perp}$  as the first basis vector, we can then take the cross product of  $\mathbf{v}_{\perp}$  with  $\hat{\mathbf{r}}$  to produce a vector  $\mathbf{w}$  that is guaranteed to be perpendicular to  $\mathbf{v}_{\perp}$  and  $\hat{\mathbf{r}}$

$$\mathbf{w} = \hat{\mathbf{r}} \times \mathbf{v}_{\perp} = \hat{\mathbf{r}} \times \mathbf{v}. \quad (11.21)$$

The rotation of  $\mathbf{v}_{\perp}$  can now be expressed in terms of this new basis.

$$\mathbf{v}' = \mathcal{R}(\mathbf{v}_{\perp}) = \cos \theta \mathbf{v}_{\perp} + \sin \theta \mathbf{w}. \quad (11.22)$$

The final result is the sum of this rotated vector and the parallel part that does not change:

$$\mathcal{R}(\mathbf{v}) = \mathcal{R}(\mathbf{v}_{\perp}) + \mathbf{v}_{\parallel} \quad (11.23a)$$

$$= \cos \theta \mathbf{v}_{\perp} + \sin \theta \mathbf{w} + \mathbf{v}_{\parallel} \quad (11.23b)$$

$$= \cos \theta [\mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}] + \sin \theta (\hat{\mathbf{r}} \times \mathbf{v}) + (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} \quad (11.23c)$$

$$= [1 - \cos \theta](\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} + \sin \theta (\hat{\mathbf{r}} \times \mathbf{v}) + \cos \theta \mathbf{v}. \quad (11.23d)$$

## 11.10 Angle-Axis Representation

253

Equation (11.23d) is known as the *Rodrigues formula* and provides a way of constructing rotation matrices in terms of the direction of the axis of rotation  $\hat{\mathbf{r}} = (r_x, r_y, r_z)$ , the cosine of the rotation angle  $c = \cos \theta$ , and the sine of the rotation angle  $s = \sin \theta$ . If we expand the vector products in (11.23d), we obtain the matrix:

$$\mathcal{R} = \begin{bmatrix} tr_x r_x + c & tr_x r_y - sr_z & tr_x r_z + sr_y \\ tr_x r_y + sr_z & tr_y r_y + c & tr_y r_z - sr_x \\ tr_x r_z - sr_y & tr_y r_z + sr_x & tr_z r_z + c \end{bmatrix}, \quad (11.24)$$

where  $t = 1 - \cos \theta$ .

The `Matrix3DTransformation` class in the numerics package provides a ready-to-use implementation of Transformation using Equation 11.24. This class also defines static convenience methods that create the simple rotation matrices  $\mathcal{R}_x$ ,  $\mathcal{R}_y$ , and  $\mathcal{R}_z$  introduced in Section 11.9.

Listing 11.7 uses the `Matrix3DTransformation` to rotate an ellipsoid about the  $(0.5, 0.5, 1)$  axis. The direction specifying the rotation need not be normalized.

Listing 11.7 `MatrixRotationApp` rotates an ellipsoid using the axis-angle representation.

```

1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.display3d.simple3d.*;
3 import org.opensourcephysics.frames.Display3DFrame;
4 import org.opensourcephysics.numerics.*;
5
6 public class MatrixRotationApp {
7     public static void main(String[] args) {
8         Display3DFrame frame = new Display3DFrame("Axis-angle
9             Rotation");
10        frame.setDecorationType(VisualizationHints.DECORATION_AXES);
11        Element ellipsoid = new ElementEllipsoid();
12        Element arrow = new ElementArrow();
13        ellipsoid.setSizeXYZ(0.4, 0.4, 1.0);
14        frame.addElement(arrow);
15        frame.addElement(ellipsoid);
16        frame.setVisible(true);
17        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
18        double theta = 0;
19        double[] axis = new double[] {0.5, 0.5, 1}; // rotation axis
20        arrow.setSizeXYZ(axis);
21        while(true) { // animate until the program exits
22            try {
23                Thread.sleep(100);
24            } catch(InterruptedException ex) {}
25            theta += Math.PI/40;
26            Transformation transformation =
27                Matrix3DTransformation.rotation(
28                    theta, axis);

```

```

27     ellipsoid.setTransformation(transformation);
28     frame.render();
29   }
30 }
31 }
```

Suppose that we wish to orient an object such that a direction within the object points in the direction of another vector  $\mathbf{b}$ . How do we create a transformation that performs this rotation? Because the axis of rotation must be perpendicular to both vectors, we use the cross product to create an axis array. The cosine of the rotation angle is easily computed by normalizing the two vectors and taking the dot product. The static `createAlignmentTransformation` method in the `Matrix3DTransformation` class constructs the needed transformation. It is implemented as follows:

```

1 public static Matrix3DTransformation
2     createAlignmentTransformation(double [] v1, double v2[]){
3         v1 = VectorMath.normalize((double []) v1.clone());
4         v2 = VectorMath.normalize((double []) v2.clone());
5         double theta = Math.acos(VectorMath.dot(v1, v2));
6         double [] axis = VectorMath.cross3D(v1, v2);
7         return Matrix3DTransformation.Rotation(theta, axis);
}
```

`AlignmentApp` uses the `createAlignmentTransformation` method to align the z-axis of a cylinder with a vector at an angle of  $45^\circ$  in the x-y plane. This program is available on the CD although it is not shown here because it is similar to Listing 11.7.

### 11.11 ■ EULER ANGLE REPRESENTATION

Euler angles are generally described in physics texts as a group of three rotations about a set of body-frame axes. An object is created with the body frame aligned with the world's cartesian coordinate system. The first rotation is about the body frame's y axis; the second rotation is about the new z axis, and the third rotation is about the new y axis. Other definitions of Euler angles are possible. For example, the Java 3D API defines Euler angles as three rotations about a fixed set of axes. All possible positions of an object can be represented using either of these conventions.

In order to construct a rotation using Euler angles we need to create transformations that rotate an object about the x-, y-, and z-axes. The following analysis shows the steps required. A full rotation matrix  $A$  can be obtained by writing the product of three individual rotation matrices.

$$A = R_1(\psi)R_2(\theta)R_3(\phi) \quad (11.25)$$

## 11.12 Quaternion Representation

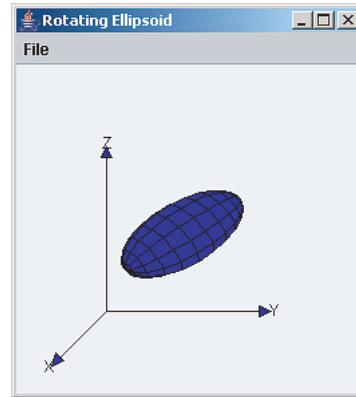
255

The first rotation is about the  $y$ -axis through an angle  $\phi$ . The second rotation is about the  $z$ -axis through an angle  $\theta$ . The final rotation is a rotation about the new  $y$ -axis through an angle  $\psi$ . Thus the third rotation and first rotation matrices are similar, but because matrix multiplication does not commute, the result can be quite complicated. This sequence of rotations gives the final transformation matrix using the physics convention for Euler angles:

$$R_1(\psi)R_2(\theta)R_3(\phi) = \begin{bmatrix} -\sin\phi\sin\psi + \cos\phi\cos\theta\cos\psi & -\sin\theta\cos\phi & \sin\phi\cos\psi + \sin\psi\cos\phi\cos\theta \\ \sin\theta\cos\psi & \cos\theta & \sin\theta\sin\psi \\ -\sin\psi\cos\phi - \sin\phi\cos\theta\cos\psi & \sin\phi\sin\theta & \cos\phi\cos\psi - \sin\phi\sin\psi\cos\theta \end{bmatrix} \quad (11.26)$$

Unfortunately, the matrix in (11.26) is singular when  $\sin\theta = 0$ . Because we must invert the above matrix to solve problems in rigid body dynamics, the Euler-angle approach becomes unstable whenever  $\theta$  approaches 0 or  $\pi$ . A better approach for numerical computation is to abandon Euler angles and to use *quaternions*.

## 11.12 ■ QUATERNION REPRESENTATION



**FIGURE 11.8** The orientation of an object can be set using quaternions to represent rotations.

Quaternions were invented by Hamilton as an elegant extension of complex numbers. Although quaternions may be unfamiliar, they are easy to use and can be related to the axis-angle representation introduced in Section 11.10.

A quaternion can be represented in terms of real and *hyper-complex* numbers  $i$ ,  $j$ , and  $k$  as

$$\hat{q} = q_0 + iq_1 + jq_2 + kq_3 = (q_0, q_1, q_2, q_3), \quad (11.27)$$

where the hyper-complex numbers obey Hamilton's rules

$$i^2 = j^2 = k^2 = ijk = -1. \quad (11.28)$$

Like imaginary numbers, the quaternion conjugate is defined as

$$\hat{q}^* = q_0 - iq_1 - jq_2 - kq_3 = (q_0, -q_1, -q_2, -q_3). \quad (11.29)$$

Unlike imaginary numbers, quaternion multiplication does not commute and obeys the rules:

$$ij = k, \quad jk = i, \quad ki = j, \quad ji = -k, \quad kj = -1, \quad ik = -j. \quad (11.30)$$

Although it would be a mistake to identify hyper-complex numbers with unit vectors in three-dimensional space (just as it would be a mistake to identify the imaginary number  $i$  with the  $y$  direction in a two-dimensional space), it is convenient to think of a quaternion as the sum of a scalar  $q_0$  and a vector  $\mathbf{q}$

$$\hat{q} = q_0 + \mathbf{q} = (q_0, \mathbf{q}). \quad (11.31)$$

The quaternion is said to be *pure* if the scalar part is zero.

By using the above definitions for multiplication of hyper-complex numbers, the product of two quaternions  $\hat{p}$  and  $\hat{q}$  can be shown to be

$$\hat{p}\hat{q} = q_0p_0 - \mathbf{q} \cdot \mathbf{p} + q_0\mathbf{p} + p_0\mathbf{q} + \mathbf{p} \times \mathbf{q} \quad (11.32)$$

where  $\mathbf{p}$  and  $\mathbf{q}$  are the vector part of the quaternions  $\hat{p}$  and  $\hat{q}$ , respectively.

As with matrices, the inverse  $\hat{q}^{-1}$  of a quaternion is another quaternion such that  $\hat{q}^{-1}\hat{q} = (1, 0, 0, 0)$ . It is easy to show that the inverse of  $\hat{q} = (q_0, q_1, q_2, q_3)$  is  $(q_0, -q_1, -q_2, -q_3)$  by carrying out the multiplication using (11.32).

Cayley discovered that if he normalized a quaternion so that  $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$  it could be used to describe pure rotations. The quaternion representation of rotation through an angle  $\theta$  about a normalized axis vector  $(n_x, n_y, n_z)$  can be written as

$$\hat{q} = \cos \frac{\theta}{2} + (in_x + jn_y + kn_z) \sin \frac{\theta}{2}. \quad (11.33)$$

We can use this quaternion to rotate a vector  $\mathbf{v}$  by constructing a pure quaternion  $\hat{v}$  using the vector as the quaternion's  $q_1$ ,  $q_2$ , and  $q_3$  components. The quaternion product

$$\hat{v}' = \hat{q}\hat{v}\hat{q}^{-1} \quad (11.34)$$

## 11.12 Quaternion Representation

257

**TABLE 11.3** Quaternion instance methods.

<b>org.opensourcephysics.display3d.numerics.Quaternion</b>	
add	Add the components of another quaternion to this quaternion.
angle	Gets the angle between this quaternion and another quaternion.
clone	Instantiates an exact copy of this quaternion.
conjugate	Changes the sign of the hyper-complex components $q_1$ , $q_2$ , and $q_3$ .
direct	Transforms the given point by the direct quaternion rotation.
getRotationMatrix	The matrix that performs an equivalent rotation.
inverse	Transforms the given point by the inverse quaternion rotation.
magnitude	Gets the magnitude of this quaternion.
magnitudeSquared	Gets the magnitude squared of this quaternion.
multiply	Multiplies this quaternion by another quaternion.
normalize	Normalizes this quaternion.
setCoordinates	Sets the quaternion coordinates $q_0$ , $q_1$ , $q_2$ , and $q_3$ .
setOrigin	Sets the origin about which the rotation will be transformed.
subtract	Subtracts the components of another quaternion from this quaternion.

produces a new pure quaternion  $\hat{v}'$  whose vector part is the rotated 3D vector. This expression is closely related to the axis-angle representation and is used in the `Quaternion` class to implement the `Transformation` interface.

The arithmetic described in this section has been implemented in the `Quaternion` class. This class stores a quaternion using instance variables  $q_0$ ,  $q_1$ ,  $q_2$ , and  $q_3$  and contains methods that perform quaternion multiplication and other operations. The following code fragment uses this class to orient a 3D element:

```

1 double [] n = new double []{ 0.5 , 0.5 , 1 }; // rotation axis
2 double theta = Math.PI/30; // rotation angle
3 double cos = Math.cos(theta/2) , sin = Math.sin(theta/2);
4 Transformation transformation =
5   new Quaternion(cos , sin*n[0] , sin*n[1] , sin*n[2]);
6 ellipsoid.setTransformation(transformation);

```

QRotationApp uses tests this code. It is available on the CD although it is not shown here because it is similar to Listing 11.7.

### 11.13 ■ RIGID BODY DYNAMICS

A solid object that does not deform is known as a *rigid body*. It has a natural coordinate system, known as the *body frame*, that has the important property that the body does not wobble if it is spun about one of these axes. The moments of inertia about the body frame axis (principal moments) are numbers that we label  $I_1$ ,  $I_2$ , and  $I_3$ . For symmetric bodies, such as boxes or ellipsoids, the body frame axes are the axes of symmetry of the object. It turns out that the differential equations that describe the rotation of a rigid body are much simpler in the body frame than in a reference frame that is fixed in space. The disadvantage is that the body frame is non-inertial and that all vectors must be transformed into this frame when solving the equations. A spinning top simulation, for example, must transform the constant force of gravity which is down in the space frame into a time-varying vector that can point up, down, or sideways as seen from the top's body frame. Fortunately, Elements have `toBodyFrame` and `toSpaceFrame` methods that makes the transformation between frames easy.

The rotational equation of motion in the body frame can be written as:

$$\mathbf{N} = \frac{d\mathbf{L}}{dt} + \boldsymbol{\omega} \times \mathbf{L} \quad (11.35a)$$

$$\mathbf{L} = \mathcal{I}\boldsymbol{\omega} \quad (11.35b)$$

where  $\mathbf{N}$  is the torque on the rigid body,  $\mathbf{L}$  is the angular momentum, and  $\boldsymbol{\omega}$  is the angular velocity. The principal moments of inertia are the diagonal elements of the inertia matrix  $\mathcal{I}$ . Equation (11.35) is Euler's equation for the motion of a rigid body. Using the three principal moments of inertia, this equation may be written in component form as

$$N_1 = I_1\dot{\omega}_1 + (I_3 - I_2)\omega_3\omega_2 \quad (11.36a)$$

$$N_2 = I_2\dot{\omega}_2 + (I_1 - I_3)\omega_1\omega_3 \quad (11.36b)$$

$$N_3 = I_3\dot{\omega}_3 + (I_2 - I_1)\omega_2\omega_1 \quad (11.36c)$$

and

$$L_1 = I_1\omega_1 \quad (11.37a)$$

$$L_2 = I_2\omega_2 \quad (11.37b)$$

$$L_3 = I_3\omega_3 \quad (11.37c)$$

where all vector components are in the body frame.

A simple application of Euler's equation for rigid bodies is the computation of the torque that must be applied to a shaft to keep an out-of-balance mass from wobbling. Assume that the angular velocity  $\boldsymbol{\omega}$  is constant in the space frame and lies along the z-axis. It is easy to rotate the mass-shaft body in the 3D scene by creating a group and transforming the group using a z-axis rotation matrix.

## 11.13 Rigid Body Dynamics

259

In order to compute the angular momentum using (11.37a), we must transform the angular velocity vector from the space frame  $(0, 0, \omega_z)$  to the body frame. The `toBodyFrame` method in the `Element` class in Listing 11.8 does just that. It shows the torque on a rectangular sheet rotating on a fixed shaft with uniform angular velocity  $\omega$ . The `SimulationControl` allows the mass to be tilted on the shaft to produce an out-of-balance configuration that would cause a real system to shake unless a torque is applied to the shaft. The program displays the angular momentum vector and the torque vector using color-coded arrows.

Listing 11.8 A mass rotating on a fixed shaft with uniform angular velocity  $\omega$ .

```

1 package org.opensourcephysics.manual.ch11;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display3d.simple3d.*;
4 import org.opensourcephysics.frames.Display3DFrame;
5 import org.opensourcephysics.numerics.*;

6
7 public class TorqueApp extends AbstractSimulation {
8     Display3DFrame frame = new Display3DFrame(" Rotation Test");
9     Element body = new ElementBox();           // shows rigid body
10    Element shaft = new ElementCylinder();      // shows shaft
11    Element arrowOmega = new ElementArrow();    // shows angular
12        velocity of shaft
13    Element arrowL = new ElementArrow();         // shows angular
14        momentum of body
15    Element arrowTorque = new ElementArrow();   // shows torque on
16        shaft
17    Group shaftGroup = new Group();            // contains shaft
18        and arrowOmega
19    Group bodyGroup = new Group();             // contains body,
20        arrowL, arrowTorque
21    double
22        theta = 0, omega = 0.1, dt = 0.1;
23    double
24        Ixx = 1.0, Iyy = 1.0, Izz = 2.0;       // principal moments
25        of inertia
26
27    public TorqueApp() {
28        frame.setDecorationType(VisualizationHints.DECORATION_AXES);
29        body.setSizeXYZ(1.0, 1.0, 0.1); // thin rectangle
30        shaft.setSizeXYZ(0.1, 0.1, 0.8);
31        arrowL.getStyle().setLineColor(java.awt.Color.MAGENTA);
32        arrowTorque.getStyle().setLineColor(java.awt.Color.CYAN);
33        bodyGroup.addElement(body);
34        bodyGroup.addElement(arrowTorque);
35        bodyGroup.addElement(arrowL);
36        shaftGroup.addElement(bodyGroup);
37        shaftGroup.addElement(arrowOmega);
38        shaftGroup.addElement(shift);
39    }
40}
```

```

33     frame.addElement(shaftGroup);
34 }
35
36 void computeVectors() {
37     double[] omega = body.toBodyFrame(new double[] {0, 0,
38                                         this.omega}); // convert omega to body
39                                         frame
40     double[] angularMomentum = new double[] {omega[0]*Ixx,
41                                         omega[1]*Iyy, omega[2]*Izz}; // L in body frame
42     double[] torque = VectorMath.cross3D(omega,
43                                         angularMomentum); // torque is computed
44                                         in body frame
45     arrowL.setSizeXYZ(angularMomentum);
46     arrowTorque.setSizeXYZ(torque);
47     // position torque arrow at tip of angular momentum
48     arrowTorque.setXYZ(angularMomentum);
49 }
50
51 public void initialize() {
52     omega = control.getDouble("omega");
53     arrowOmega.setSizeXYZ(0, 0, omega);
54     double tilt = control.getDouble("tilt");
55     bodyGroup.setTransformation(
56         Matrix3DTransformation.rotationX(tilt));
57     computeVectors();
58 }
59
60 public void reset() {
61     control.setValue("omega", "pi/4");
62     control.setValue("tilt", "pi/5");
63 }
64
65 protected void doStep() {
66     theta += omega*dt;
67     shaftGroup.setTransformation(
68         Matrix3DTransformation.rotationZ(theta));
69     computeVectors();
70 }
71
72 public static void main(String[] args) {
73     SimulationControl.createApp(new TorqueApp());
74 }
75 }
```

Notice how `TorqueApp` instantiates a shaft group and a body group in its constructor. The shaft group contains visual representations of the shaft, the angular velocity vector, and the body group. The body group contains representations of a thin rectangular sheet, the angular velocity, and the torque. The body group is rotated about the x-axis in the `initialize` method and the shaft group is rotated

## 11.14 JOGL

261

about the z-axis in the `doStep` method. Because the body group is within the shaft group, the body group also rotates. The `computeVectors` method displays the angular momentum and torque vectors by computing their values in the rotating (body) frame of reference. The torque is computed using a cross product of the body frame angular velocity and the body frame angular momentum according to the following kinematic formula:

$$\frac{d\mathbf{L}_{\text{space}}}{dt} = \frac{d\mathbf{L}_{\text{body}}}{dt} + \boldsymbol{\omega} \times \mathbf{L}_{\text{body}}. \quad (11.38)$$

Rigid body dynamics is a rich topic that is covered in most intermediate level mechanics textbooks. Unfortunately, its development is beyond the scope of this Guide. The best way to approach these types of problems is to write the equation of motion of the body in terms of quaternion components and their derivatives. The derivation of these equations can be found in *An Introduction to Computer Simulation Methods* by Gould, Tobochnik, and Christian. The `SpinningTopApp` program from this book is available on the CD. The application of quaternions to molecular dynamics simulations can be found in books by Rapaport or Allen.

## 11.14 ■ JOGL

Glenn Ford has developed a JOGL implementation of OSP 3D API and this implementation is located in the `jogl` package `org.opensourcephysics.-display3d.jogl`. This code and the examples in this section are available on the CD in the optional `osp_jogl.zip` archive. All that is required to use the `jogl` package is to install the operating system specific JOGL library and import the `jogl` package.

```
import org.opensourcephysics.display3d.jogl.*;
```

The JOGL library is available for Windows 2000 or later, OS X 10.3 or later, Solaris, and Linux. It can be downloaded from <https://jogl.dev.java.net/>. Libraries and installation instructions vary depending on the operating system because JOGL directly accesses video hardware. For example, on Windows the library is distributed in a zip file and you must copy the enclosed DLL files into the system folder (usually C:\windows or something similar) and the JOGL jar files into Java VM's class path (usually lib\ext) subdirectories in the JDK and the JRE).

Using the `jogl` package not only produces more realistic images, but it also allows programmers to customize the scene in ways that `simple3d` does not allow. The `jogl` implementation of the `DrawingPanel3D` interface has three methods that can be overridden to access underlying OpenGL library: `additionalGLInit`, `preCameraGL`, and `postCameraGL`. The `additionalGLInit` method is invoked just after the standard initialization of the OSP JOGL display, the `preCameraGL` method is invoked just before the camera rotates the scene, and the `postCameraGL` method is called just after the camera rotates the scene. The

additionalGLInit and postCameraGL methods are used to add lights to a scene in the SpotLightApp program shown in Listing 11.9.

Listing 11.9 A bouncing ball simulation with a spotlight that shines on a surface.

```

1 package org.opensourcephysics.manual.ch11;
2 import java.awt.*;
3 import net.java.games.jogl.*;
4 import org.opensourcephysics.controls.*;
5 import org.opensourcephysics.display3d.jogl.*;
6 import org.opensourcephysics.numerics.*;
7
8 public class SpotLightApp extends AbstractSimulation {
9     private static boolean
10        original_light = true, spot_light = true;
11     Element ball, floor;
12     double
13        velocity = 0, time = 0;
14     Vec3D lightPosition;
15     // Here it's a LightTestPanel defined at the bottom of this
16     // class
17
18     DrawingPanel3D panel;
19     DrawingFrame3D frame;
20
21     public SpotLightApp() {
22         // bouncing ball
23         ball = new ElementSphere();
24         ball.setXYZ(0, 0, 9.0);
25         ball.setSizeXYZ(2, 2, 2);
26         ball.getStyle().setFillColor(java.awt.Color.YELLOW);
27         ball.getStyle().setResolution(new Resolution(70, 70, 0));
28         // floor
29         floor = new ElementBox();
30         floor.setXYZ(0.0, 0.0, 0.0);
31         floor.setSizeXYZ(7.0, 7.0, 1.0);
32         floor.getStyle().setResolution(new Resolution(70, 70, 2));
33         // light will go above the ball and circle around
34         lightPosition = new Vec3D(1, 0, 10);
35         // Create the panel
36         panel = new LightTestPanel();
37         panel.getComponent().setBackground(Color.black);
38         panel.getVisualizationHints().setDecorationType(
39             VisualizationHints.DECORATION.NONE);
40         panel.getVisualizationHints().setAllowQuickRedraw(false);
41         // Add the objects to panel
42         panel.addElement(ball);
43         panel.addElement(floor);
44         frame = new DrawingFrame3D(panel);
45         frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);

```

## 11.14 JOGL

## 263

```

45     frame.setSize(300, 300);
46     frame.setVisible(true);
47 }
48
49 protected void doStep() {
50     double
51         z = ball.getZ(), dt = 0.05;
52     time += dt;
53     z += velocity*dt; // moves the ball
54     velocity -= 9.8*dt; // acceleration changes the velocity
55     if((z<=1.0)&&(velocity <0)) {
56         velocity = -velocity;
57     }
58     ball.setZ(z);
59     // make the light circle around the scene
60     lightPosition.x = Math.cos(time)*1.5;
61     lightPosition.y = Math.sin(time)*1.5;
62 }
63
64 public static void main(String[] args) {
65     AbstractSimulation s = new SpotLightApp();
66     s.startSimulation(); // starts the thread
67 }
68
69 private class LightTestPanel extends DrawingPanel3D {
70     public void additionalGLInit(GLDrawable drawable) {
71         initSpot(drawable.getGL());
72     }
73
74     public void postCameraGL(GLDrawable drawable) {
75         GL gl = drawable.getGL();
76         if(original_light) {
77             gl glEnable(GL.GL_LIGHT0);
78         } else {
79             gl glDisable(GL.GL_LIGHT0);
80         }
81         if(spot_light) {
82             gl glEnable(GL.GL_LIGHT2);
83             updateSpotProperties(gl);
84         } else {
85             gl glDisable(GL.GL_LIGHT2);
86         }
87     }
88
89     private void initSpot(GL gl) {
90         // white light
91         float ambient[] = {1.0f, 1.0f, 1.0f, 1.0f};
92         // properties of the light
93         gl glLightfv(GL.GL_LIGHT2, GL.GL_AMBIENT, ambient);

```

```

94     // Spot properties
95     // define the spot direction and cut-off
96     updateSpotProperties(gl);
97     // exponent propertie defines the concentration of the
98     // light
99     gl.g1Lightf(GL.GL_LIGHT2, GL.GL_SPOT_EXPONENT, 70.0f);
100    // light attenuation (default values used here : no
101    // attenuation with the distance)
102    gl.g1Lightf(GL.GL_LIGHT2, GL.GL_CONSTANT_ATTENUATION,
103                 0.0f);
104    gl.g1Lightf(GL.GL_LIGHT2, GL.GL_LINEAR_ATTENUATION,
105                 0.001f);
106    gl.g1Lightf(GL.GL_LIGHT2, GL.GL_QUADRATIC_ATTENUATION,
107                 0.03f);
108    gl.g1Enable(GL.GL_LIGHT2);
109 }
110
111 private void updateSpotProperties(GL gl) {
112     // set the light position
113     float position[] = {(float) lightPosition.x,
114                          (float) lightPosition.y, (float) lightPosition.z,
115                          1.0f};
116     gl.g1Lightfv(GL.GL_LIGHT2, GL.GL_POSITION, position);
117     // our light will always face down
118     float direction[] = {0, 0, -1};
119 }
```

Elements in the OSP jogl package can be customized by overriding the additionalGL method. This method is called just before the element is filled and edge-lines are drawn. The panel's preCameraGL method and the element's additionalGL method are used to a surface texture in the TextureApp program shown in Listing 11.10.

Listing 11.10 A sphere with a striped texture map.

```

1 package org.opensourcephysics.manual.ch11;
2 import net.java.games.jogl.GL;
3 import net.java.games.jogl.GLDrawable;
4 import org.opensourcephysics.display3d.jogl.*;
5
6 public class TextureApp {
7     DrawingPanel3D panel;
```

## 11.14 JOGL

265

```

8   DrawingFrame3D frame ;
9
10  public TextureApp() {
11      panel = new TextureTestPanel();
12      Element ball = new TexturedSphere(makeStripeImage1());
13      ball.setSizeXYZ(4, 4, 4);
14      ball.setXYZ(0, -3, 0);
15      ball.getStyle().setResolution(new Resolution(30, 30, 0));
16      panel.addElement(ball);
17      ball = new TexturedSphere(makeStripeImage2());
18      ball.setSizeXYZ(2, 2, 2);
19      ball.setXYZ(0, 3, 0);
20      ball.getStyle().setResolution(new Resolution(30, 30, 0));
21      panel.addElement(ball);
22      frame = new DrawingFrame3D(panel);
23      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
24      frame.setSize(300, 300);
25      frame.setVisible(true);
26  }
27
28  public static void main(String[] args) {
29      new TextureApp();
30  }
31
32  private class TextureTestPanel extends DrawingPanel3D {
33
34      private float xequalzero[] = {1.0f, 0.0f, 0.0f, 0.0f};
35      private float slanted[] = {1.0f, 1.0f, 1.0f, 0.0f};
36      private float currentCoeff[];
37      private int currentPlane;
38      private int currentGenMode;
39      // Set all of the proper flags for the textures we want to
40      // use
41
42      public void preCameraGL(GLDrawable drawable) {
43          GL gl = drawable.getGL();
44          gl.glEnable(GL.GL_DEPTH_TEST);
45          gl.glShadeModel(GL.GL_SMOOTH);
46          gl.glTexEnvf(GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
47                      GL.GL_MODULATE);
48          currentCoeff = xequalzero;
49          currentGenMode = GL.GL_OBJECT_LINEAR;
50          currentPlane = GL.GL_OBJECT_PLANE;
51          gl.glTexGeni(GL.GL_S, GL.GL_TEXTURE_GEN_MODE,
52                      currentGenMode);
53          gl.glTexGenfv(GL.GL_S, currentPlane, currentCoeff);
54          gl.glEnable(GL.GL_TEXTURE_GEN_S);
55          gl.glEnable(GL.GL_TEXTURE_1D);
56          gl.glEnable(GL.GL_CULL_FACE);

```

```

55     g1.gLEnable(GL.GL_LIGHTING);
56     g1.gLEnable(GL.GL_LIGHT0);
57     g1.gLEnable(GL.GL_AUTO_NORMAL);
58     g1.gLEnable(GL.GL_NORMALIZE);
59     g1.gLCullFace(GL.GL_BACK);
60     g1.gLMaterialf(GL.GL_FRONT, GL.GL_SHININESS, 64.0f);
61   }
62 }
63
64 private static final int stripeImageWidth = 32;
65 // Make a striped texture that's opaque
66
67 private byte[] makeStripeImage1() {
68   byte[] stripeImage = new byte[stripeImageWidth*4];
69   for(int j = 0;j<stripeImageWidth;j++) {
70     if(j<=4) {
71       stripeImage[j] = (byte) 255;
72     } else {
73       stripeImage[j] = (byte) 60;
74     }
75     if(j>4) {
76       stripeImage[j+stripeImageWidth] = (byte) 255;
77     } else {
78       stripeImage[j+stripeImageWidth] = (byte) 60;
79     }
80     stripeImage[j+2*stripeImageWidth] = (byte) 60;
81     stripeImage[j+3*stripeImageWidth] = (byte) 255;
82   }
83   return stripeImage;
84 }
85
86 // Make a translucent gradient-striped texture
87 private byte[] makeStripeImage2() {
88   byte[] stripeImage = new byte[stripeImageWidth*4];
89   for(int j = 0;j<stripeImage.length;j++) {
90     stripeImage[j] = (byte) (j*5);
91   }
92   return stripeImage;
93 }
94
95 private class TexturedSphere extends ElementSphere {
96   private byte stripeImage [] = new byte[stripeImageWidth*4];
97
98   public TexturedSphere(byte[] stripeImage) {
99     super();
100    this.stripeImage = stripeImage;
101  }
102
103 // Set the texture before rendering

```

## 11.15 Programs

267

```

104 public void additionalGL(GLDrawable drawable) {
105     GL gl = drawable.getGL();
106     gl.glPixelStorei(GL.GL_UNPACK_ALIGNMENT, 1);
107     gl.glTexParameterf(GL.GL_TEXTURE_1D, GL.GL_TEXTURE_WRAP_S,
108                         GL.GL_REPEAT);
109     gl.glTexParameterf(GL.GL_TEXTURE_1D,
110                         GL.GL_TEXTURE_MAG_FILTER,
111                         GL.GL_LINEAR);
112     gl.glTexParameterf(GL.GL_TEXTURE_1D,
113                         GL.GL_TEXTURE_MIN_FILTER,
114                         GL.GL_LINEAR);
115     gl.glTexImage1D(GL.GL_TEXTURE_1D, 0, 4, stripeImageWidth,
116                         0,
117                         GL.GL_RGBA, GL.GL_UNSIGNED_BYTE, stripeImage);
118 }
119 }
120 }
```

**11.15 ■ PROGRAMS**

The following examples are in the `org.opensourcephysics.manual.ch11` package. JOGL examples can be found in the `org.opensourcephysics.-manual.ch11_jogl` package.

**AlignmentApp**

`AlignmentApp` uses a Quaternion to align a 3D Element with a vector. See Section 11.10.

**BallInteractionApp**

`BallInteractionApp` uses the `InteractionTarget` API to enable changes in position. See Section 11.6.

**BounceApp**

`BounceApp` presents a 3D visualization of a bouncing ball as described in Section 11.3.

**Box3DApp**

`Box3DApp` creates a 3D box using the `simple3d` package and sets its properties. See Section 11.2.

**CameraApp**

`CameraApp` creates a 3D scene and sets the camera's properties. See Section 11.4.

**ElementApp**

ElementApp demonstrates the appearance and properties of 3D elements. See Section 11.5.

**FreeRotationApp**

FreeRotationApp models the torque free rotation of a spinning object. See Section 11.13.

**GroupApp**

GroupApp demonstrates how a group of objects can be positioned as a single object as described in Section 11.5.

**Interaction3DApp**

Interaction3DApp demonstrates how to add and handle actions in a Display3DFrame. See Section 11.6.

**LorenzApp**

LorenzApp models the Lorenz attractor by extending AbstractSimulation and implementing the doStep method. See Section 11.5.

**MatrixRotationApp**

MatrixRotationApp demonstrates how to use a Matrix3DTransformation to rotate a 3D element. See Section 11.10.

**QRotationApp**

QRotationApp demonstrates how to use the Quaternion class to rotate an Element. See Section 11.12.

**SHO3DApp**

SHO3DApp creates a 3d harmonic oscillator simulation by extending AbstractSimulation and implementing the doStep method.

**SpinningTopApp**

SpinningTopApp models the dynamics of a spinning top using quaternions. See Section 11.13.

**TorqueApp**

TorqueApp models Euler's equations to show the torque on an object that is spinning about a fixed axis. See Section 11.13.

## CHAPTER

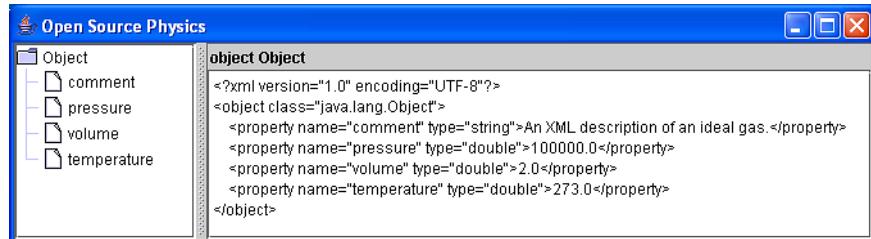
## 12

## XML Documents

©2005 by Doug Brown and Wolfgang Christian, July 23, 2005

Open Source Physics defines an xml framework that can be used to store, retrieve, and display structured documents.

### 12.1 ■ BASIC XML



**FIGURE 12.1** A graphical view of a basic xml document describing an ideal gas. When a node is selected, the corresponding xml `<object>` or `<property>` element is displayed in the right hand panel.

Extensible Markup Language (xml) is a significant development in information technology because it allows us to define a portable language for data that organizes and categorizes the data in a consistent and self-describing manner. It is used by OSP applications such as by Launcher (Chapter 15) to organize curricular material and Tracker (Chapter 16) to store video analysis data. If you are unfamiliar with the xml language, don't worry. Although xml can be an unforgiving language to write using a text editor, it is highly structured and therefore not very difficult to program. The OSP library provides all the tools that you need to create simple xml documents. We start by studying the XML representation of an ideal gas shown in Listing 12.1.

Listing 12.1 An ideal gas model described using xml.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <object class="java.lang.Object">
3      <property name="comment" type="string">An xml description of
          an ideal gas.</property>

```

```

4   <property name="pressure" type="double">100000.0</property>
5   <property name="volume" type="double">2.0</property>
6   <property name="temperature" type="double">273.0</property>
7 </object>
```

XML documents have a very simple structure. Like hypertext markup language (html) documents they are written in plain text and consist of one or more named *elements* that store data between their opening and closing tags. Listing 12.1 starts with a header that identifies it as an xml document. The document has a single root element that contains all xml content. The root of an Open Source Physics xml document is an `<object>` element that stores data in child `<property>` elements with *name* and *type* attributes. The *value* of the property is written between the opening and closing property tag.

Unlike html, anyone can define their own xml tags, their rules, and their meaning using a Document Type Definition (DTD). The Open Source Physics DTD is located in the resources package and defines only two element types, `<object>` and `<property>`. The `<object>` element stores the object's Java class as an attribute and the object's properties in child `<property>` elements.

Our xml framework is consistent with the control API described in Chapter 5. Programs instantiate a subinterface of Control named `XMLControl` and then set and get values from this control. The OSP implementation of `XMLControl` is the `XMLControlElement` class. Listing 12.2 uses the `XMLControlElement` to set values for an ideal gas model and to write these values to a file named `gas_data.xml`.

Listing 12.2 An `XMLControl` can be used to write data to a file.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.XMLControl;
3 import org.opensourcephysics.controls.XMLControlElement;
4
5 public class WriteXMLApp {
6     static String fileName = "gas_data.xml";
7
8     public static void main(String[] args) {
9         XMLControl xml = new XMLControlElement();
10        xml.setValue("comment", "An XML description of an ideal
11                      gas.");
12        xml.setValue("pressure", 1.0E5);
13        xml.setValue("volume", 2.0);
14        xml.setValue("temperature", 273.0);
15        xml.write(fileName);
16    }
}
```

Because the `gas_data.xml` file is a text file, it can be examined in any text editor. XML aware editors will often provide a tree view of the content as shown in Figure 12.1.

## 12.2 Object Properties

271

An XMLControl supports standard OSP control methods such as `getDouble` and `getString` to retrieve values of common data types. The `ReadXMLApp` program uses these methods to read the `gas_data.xml` data file.

Listing 12.3 An XMLControl can be used to read data from an xml file.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.XMLControl;
3 import org.opensourcephysics.controls.XMLControlElement;
4
5 public class ReadXMLApp {
6     public static void main(String[] args) {
7         XMLControl xml = new XMLControlElement("gas_data.xml");
8         System.out.println(xml.getString("comment"));
9         System.out.println(xml.getDouble("pressure"));
10        System.out.println(xml.getDouble("volume"));
11        System.out.println(xml.getDouble("temperature"));
12    }
13 }
```

Listing 12.4 shows how to display data in an xml property tree (see Figure 12.1) using an `XMLTreePanel`. When an object or property node is selected in the tree, the corresponding xml `<object>` or `<property>` element is displayed. Primitive and string property types are directly editable in the text field.

Listing 12.4 An `XMLTreePanel` is used to display an `XMLControl` in an xml property tree.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.OSPFrame;
4 import javax.swing.JFrame;
5
6 public class ShowXMLApp {
7     public static void main(String[] args) {
8         XMLControl xml = new XMLControlElement("gas_data.xml");
9         // display xml data in a tree view
10        JFrame frame = new OSPFrame(new XMLTreePanel(xml));
11        frame.setSize(650, 550);
12        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13        frame.setVisible(true);
14    }
15 }
```

## 12.2 ■ OBJECT PROPERTIES

Unlike basic OSP controls, xml controls can set and get objects in addition to primitive data types. Because child elements can be Java arrays, collections, and

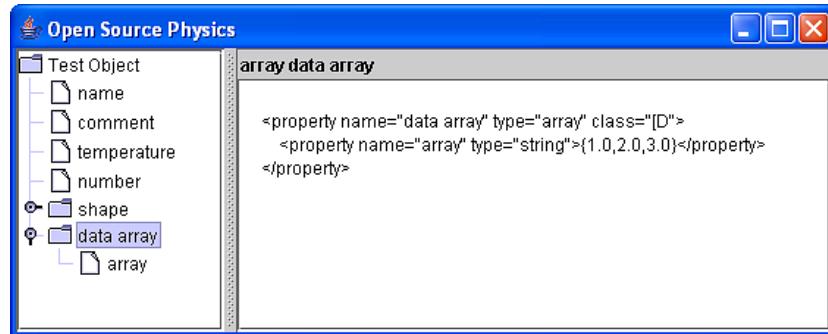


FIGURE 12.2 An xml property tree displays nested xml properties.

objects as well as primitive data types, property elements are often nested within an XML document. An `XMLControl` can, for example, store arrays and interactive shapes as shown in Listing 12.5. Note the various data types. The `InteractiveShape` object, for example, is a child element that has double, boolean, and color properties.

Listing 12.5 Objects in an XML document are nested within other objects. data.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.InteractiveShape;
4 import org.opensourcephysics.display.OSPFrame;
5 import javax.swing.JFrame;
6
7 public class WriteShowNestedXMLApp {
8     public static void main(String[] args) {
9         // create the XMLControl
10        XMLControl xml = new XMLControlElement();
11        xml.setValue("name", "Test Object");
12        xml.setValue("comment", "A test of XML.");
13        xml.setValue("temperature", 273.4);
14        xml.setValue("number", 200);
15        xml.setValue("shape",
16                    InteractiveShape.createRectangle(0.0, 1.0, 2.0,
17                                                 5.0));
18        xml.setValue("data array", new double[] {1, 2, 3});
19        // write the xml document
20        xml.write("nested_data.xml"); // save to file
21        // display the xml document in a tree view
22        JFrame frame = new OSPFrame(new XMLTreePanel(xml));
23        frame.setSize(650, 550);
24        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25        frame.setVisible(true);
}

```

## 12.2 Object Properties

273

26 }

Listing 12.5 writes an xml document to a file and displays the document in the xml tree shown in Figure 12.2. When a node contains an object, the object's properties are displayed in a subtree and the corresponding xml description is displayed.

An xml element's `name` property has a special significance for `Objects`. If the `name` property is set, then the value of that property will be displayed as the node name in the xml inspector.

```
1 xml.setValue("name","Test Object");
```

If the name property is not set, the inspector uses the class name when displaying the node in the tree.

Listing 12.6 shows how arrays and objects are retrieved from an `XMLControl`. Arrays and collections of primitive data types are instantiated within the `XMLControl` when the `getObject` is invoked. The program creates the objects and prints their properties in the system console after reading the `nested_data.xml` data file.

Listing 12.6 XML reads structured data from a file.

```
1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.XMLControl;
3 import org.opensourcephysics.controls.XMLControlElement;
4
5 public class ReadNestedXMLApp {
6     static String fileName = "nested_data.xml";
7
8     public static void main(String[] args) {
9         XMLControl xml = new XMLControlElement();
10        xml.read(fileName);
11        System.out.println(xml.getString("comment"));
12        System.out.println(xml.getDouble("temperature"));
13        System.out.println(xml.getInt("number"));
14        double[] array = (double[]) xml.getObject("data array");
15        for(int i = 0, n = array.length;i<n;i++) {
16            System.out.println(array[i]);
17        }
18        System.out.println(xml.getObject("shape"));
19    }
20}
```

Because the `getObject` method returns a generic Java `Object`, the value returned must be cast to the correct data type. The `getObject` method will instantiate any Java class that defines a no argument constructor or an object loader that implements the `XML.ObjectLoader` interface. This interface makes xml-based reading and writing possible and is described in the next section. Note that the

`InteractiveShape` class and many other OSP classes have predefined *object loaders* that implement this interface.

An `XMLControl` instance stores name-value pairs but what happens if we attempt to read a value that has not been stored? If there is no `boolean` then `getBoolean` returns `false`, if no `double` then `getDouble` returns `Double.NaN`, if no `int` then `getInt` returns `Integer.MIN_VALUE`, and if there is no `Object` then `getObject` returns `null`.

```

1 XMLControl xml = new XMLControlElement();
2 Color color = (Color) xml.getObject("background color");
3 // color will be null because a name-value pair has not been set

```

### 12.3 ■ OBJECT LOADERS

Objects save and load xml documents using an xml *object loader* that implements the `XML.ObjectLoader` interface shown in Listing 12.7.

Listing 12.7 The `XML.ObjectLoader` interface.

```

1 // See the XML class definition in the controls package for
2 // additional ObjectLoader documentation.
3 public interface ObjectLoader {
4     // Saves data from an object in an XMLControl.
5     public void saveObject(XMLControl control, Object obj);
6
7     // Creates an object using data in an XMLControl.
8     public Object createObject(XMLControl control);
9
10    // Loads an object with data from an XMLControl.
11    public Object loadObject(XMLControl control, Object obj);
12}

```

Implementations of the `XML.ObjectLoader` interface create, save, and load objects for a single Java class. Although loaders can be defined and implemented by any class, it is usually convenient and efficient to create them using an inner class. Listing 12.8 defines an `IdealGas` class that uses an anonymous inner class as a loader. The loader is created within the argument to the `XML.setLoader` method. Only one loader is needed for all `IdealGas` objects and this loader is accessed automatically from within the xml framework as needed.

Listing 12.8 A model of an ideal gas that defines an `XML.ObjectLoader`.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.*;
3
4 public class IdealGas {
5     static final double R = 8.31; // gas constant J/mole/K

```

## 12.3 Object Loaders

275

```

6   double p, v, t;           // pressure , volume , and
7   // temperature in MKS units
8
9   public IdealGas() {
10    this(101.3e3, 22.4e-3, 273);
11 }
12
13  public IdealGas(double pressure, double volume, double
14   temperature) {
15    p = pressure;
16    v = volume;
17    t = temperature;
18 }
19
20  public double getMoles() {
21    return p*v/R/t;
22 }
23
24  public String toString() {
25    return "ideal gas model: P=" + p + " V=" + v + " T=" + t;
26 }
27
28  static {
29    XML.setLoader(IdealGas.class, new XML.ObjectLoader() {
30      public void saveObject(XMLControl control, Object obj) {
31        IdealGas gas = (IdealGas) obj;
32        control.setValue("pressure", gas.p);
33        control.setValue("volume", gas.v);
34        control.setValue("temperature", gas.t);
35      }
36      public Object createObject(XMLControl control) {
37        return new IdealGas(); // creates a gas model at STP
38      }
39      public Object loadObject(XMLControl control, Object obj) {
40        IdealGas gas = (IdealGas) obj;
41        gas.p = control.getDouble("pressure");
42        gas.v = control.getDouble("volume");
43        gas.t = control.getDouble("temperature");
44        return gas;
45      }
46    });
47  }
48 }
```

The `IdealGas` explicitly registers its loader using the static `XML.setLoader(classtype, loader)` method. The `XML.setLoader` method adds the loader to the xml *loader registry*. Registering a loader for a given class allows objects of that class to be saved and loaded automatically. Another way for a class to register an xml loader is to define a public static `getLoader()` method

within the class itself that returns the loader. Most Open Source Physics classes use the `getLoader()` method because the loader is only created as needed. The xml framework first looks for a registered loader; if none, it looks for a static `getLoader()` method in the object class; if still none, then it uses a default loader. Classes with a registered loader can save object data in an xml document using the `saveObject` method as shown in Listing 12.9.

Listing 12.9 Loaders can be used to save object data.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.XMLControl;
3 import org.opensourcephysics.controls.XMLControlElement;
4
5 public class IdealGasSaveApp {
6     public static void main(String[] args) {
7         XMLControl xml = new XMLControlElement();
8         // create and save data for an ideal gas with the given state
9         xml.saveObject(new IdealGas(1.0E5, 2.0, 300));
10        xml.write("gas_object.xml"); // save to file
11    }
12}
```

Object loaders can also load xml data into an existing object. The `IdealGasLoadApp` program (Listing 12.10) creates an `IdealGas` in its default state at standard temperature and pressure. It then reads data into an `XMLControl` and loads the data into the gas object by invoking the `loadObject` method.

Listing 12.10 Loaders can be used to set properties of existing objects.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.*;
3
4 public class IdealGasLoadApp {
5     public static void main(String[] args) {
6         IdealGas gas = new IdealGas(); // ideal gas at STP
7         // creates an XML control and reads the xml data
8         XMLControl xml = new XMLControlElement("gas_object.xml");
9         // load gas object with data from xml control
10        xml.loadObject(gas);
11        // print a string representation of the gas
12        System.out.println(gas);
13    }
14}
```

Object loaders can be used to instantiate Java objects. The `IdealGasCreateApp` shown in Listing 12.11 first reads a data file into an `XMLControl`. Because the control's `loadObject` method is passed a null argument, the control invokes the loader's `createObject` method and passes the created object to the loader's `loadObject` method.

## 12.4 OSP Applications

277

Listing 12.11 Loaders can be used to create new objects.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.XMLControl;
3 import org.opensourcephysics.controls.XMLControlElement;
4
5 public class IdealGasCreateApp {
6     public static void main(String[] args) {
7         XMLControl xml = new XMLControlElement("gas_object.xml");
8         // creates object with stored data
9         IdealGas gas = (IdealGas) xml.loadObject(null);
10        System.out.println(gas);
11    }
12 }
```

Object loaders provide a convenient way to implement the `Cloneable` interface. An `XMLControlElement` is constructed using the current object. Invoking the control's `loadObject` method (Listing 12.12 with a null argument instantiates a new object using control's data.

Listing 12.12 Object loaders can be used to clone objects.

```

1 // implementation of Cloneable interface using XML
2 public Object clone(){
3     XMLControl control = new XMLControlElement(this);
4     return control.loadObject(null);
5 }
```

## 12.4 ■ OSP APPLICATIONS

XML is used throughout the Open Source Physics library to store a program's data, to create menus, to inspect objects, and to transfer data between programs. For example, many Open Source Physics controls have menu items to read, save, and inspect an object's state using xml. Because a program's state depends on data stored in the control and on data stored in the model, we define a class named `OSPAplication` that contains references to both these objects. The application's loader creates an xml tree that has the `OSPAplication` object as the root document. The first element is named *control* and contains the control's variable names and values. The second element is named *model* and contains the model's data. If an object such as the model does not have a loader, the object's class name is the only saved data.

Listing 12.13 defines a simple model that creates particles at random locations. Because these particles are interactive, they can be dragged within the display frame.

Listing 12.13 A model that creates particles at random locations.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.InteractiveShape;
4 import org.opensourcephysics.frames.DisplayFrame;
5
6 public class ParticleApp extends AbstractCalculation {
7     DisplayFrame frame = new DisplayFrame("x", "y", "Particles");
8
9     public void calculate() {
10         frame.clearDrawables(); // remove old circles
11         int n = control.getInt("number of particles");
12         double r = control.getDouble("radius");
13         for(int i = 0; i < n; i++) {
14             double x = -10+20*Math.random();
15             double y = -10+20*Math.random();
16             frame.addDrawable(InteractiveShape.createCircle(x, y, r));
17         }
18     }
19
20     public void reset() {
21         control.setValue("number of particles", 10);
22         control.setValue("radius", 0.5);
23     }
24
25     public static XML.ObjectLoader getLoader() {
26         return null;
27         // uncomment the next line to enable the loader
28         // return new ParticleAppLoader();
29     }
30
31     public static void main(String[] args) {
32         // creates the program and reads xml data using command-line
33         // arguments
34         CalculationControl.createApp(new ParticleApp(), args);
35         // command line arguments are optional
36         // use args[0] = "default_particles.xml" to read the example
37     }
}

```

Run `ParticleApp` and inspect the application's xml tree using the menu item under the control's file menu. Note that the control node contains its name-value pairs but that the model does not because the `getLoader` method in `ParticleApp` does not return a loader. We next discuss the loader that saves the model's data so that the program's entire configuration can later be recreated. Listing 12.14 shows how such a loader is written.

Listing 12.14 An `XML.ObjectLoader` for the `ParticleApp`.

```

1 package org.opensourcephysics.manual.ch12;

```

## 12.4 OSP Applications

279

```

2 import org.opensourcephysics.controls.XML;
3 import org.opensourcephysics.controls.XMLControl;
4
5 public class ParticleAppLoader implements XML.ObjectLoader {
6     public Object createObject(XMLControl control) {
7         ParticleApp model = new ParticleApp();
8         return model;
9     }
10
11    public void saveObject(XMLControl control, Object obj) {
12        ParticleApp model = (ParticleApp) obj;
13        control.setValue("frame", model.frame);
14    }
15
16    public Object loadObject(XMLControl control, Object obj) {
17        ParticleApp model = (ParticleApp) obj;
18        XMLControl childControl = control.getChildControl("frame");
19        if(childControl==null) {
20            return obj;
21        }
22        childControl.loadObject(model.frame);
23        model.calculate(); // calculate with the new values
24        model.frame.repaint();
25        model.frame.setVisible(true);
26        return obj;
27    }
28}

```

The ParticleApp loader shown defines methods to create, load, and save the model. Because the frame already defines an object loader that stores particle data, we only need to store and load the frame. Objects with xml loaders are saved in an object property using the `setValue` method. The following statement shows how the display frame's data are saved.

```
1 control.setValue("frame", model.frame);
```

Loading child xml data into an existing object is a two-step process. First the child XMLControl representing the named object is retrieved from the parent XMLControl. The child's `loadObject` method is then passed the object to be loaded. The passed object must, of course, have an xml loader.

```
1 XMLControl childControl = control.getChildControl("frame");
2 childControl.loadObject(model.frame);
```

Our simple graphical user interfaces such as `CalculationControl` and `AnimationControl` are not xml controls. They do, however, use xml controls internally to load and save data. First enable the use of the `ParticleAppLoader` by un-commenting the line containing the loader in the `ParticleApp` program's `getLoader` method. Run the program, open the inspector in the `Calculation-`

Control file menu, and expand the model’s xml tree. Observe how the program’s control and model are nested in the inspector. Select the `radius` parameter in the control node. Enter a new value in the inspector’s text field and note that the value displayed within the graphical user interface changes. Select a particle and change its x and y coordinates in the inspector. Note that the particle repositions itself on the screen. Close the inspector and save the application’s state in a file named `default_particles.xml`. Change the program’s state and then read the data file back into the program to restore the saved configuration.

XML data can be read into our graphical user interface controls using the `loadXML` method.

```
1 // defined for standard graphical user interface controls
2 control.loadXML("default_particles.xml");
```

We can also read xml data during program instantiation by passing xml data to the user interface control using the main method’s command-line arguments.

```
1 public static void main(String[] args) {
2     CalculationControl.createApp(new ParticleApp(), args);
3 }
```

Run `ParticleApp` with the “`default_particles.xml`” command line parameter and note that the saved data determines the program’s default conditions. The most recently accessed xml data file will be loaded into a program when the reset button is pressed in any of the standard OSP controls. The ability to set a program’s default parameters with an xml data file is designed for curriculum authors wishing to distribute a program with multiple initial conditions using the Launcher as described in Chapter 15.

## 12.5 ■ CLIPBOARD DATA TRANSFER

The ability to store and create objects using xml allows users to easily pass data between applications using the clipboard. The user copies an xml description onto the clipboard and pastes this description into another program. The following code fragment shows how the system clipboard is used.

```
1 protected void copyAction(XMLControlElement control) {
2     StringSelection data = new StringSelection(control.toXML());
3     Clipboard clipboard =
4         Toolkit.getDefaultToolkit().getSystemClipboard();
5     clipboard.setContents(data, null);
}
```

The mechanism for cutting and pasting clipboard contents to and from a `DrawingFrame` is, in fact, already in place. To examine how this mechanism works, run `ParticleApp` and `ScratchPadApp` simultaneously. The `ScratchPadApp` is shown in Listing 12.15.

## 12.6 Programs

281

Listing 12.15 A program with a DrawingFrame that has been enabled to accept clipboard data.

```

1 package org.opensourcephysics.manual.ch12;
2 import org.opensourcephysics.display.*;
3 import javax.swing.JFrame;
4
5 public class ScratchPadApp {
6     public static void main(String[] args) {
7         PlottingPanel panel = new PlottingPanel("x", "y", "Scratch
8             Pad");
9         panel.setPreferredMinMax(-10, 10, -10, 10);
10        DrawingFrame frame = new DrawingFrame(panel);
11        frame.setEnabledPaste(true);
12        frame.setEnabledReplace(true);
13        frame.setVisible(true);
14        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15    }
}

```

Create a random distribution of particles using ParticleApp and select the copy item under the edit menu. This action copies the particle xml data to the clipboard. Select the paste item under the ScratchPadApp file menu to paste the particles into this second program. Repeat.

## 12.6 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch12` package.

### **IdealGasCreateApp**

`IdealGasCreateApp` creates a new ideal gas model using xml data.

### **IdealGasLoadApp**

`IdealGasLoadApp` loads an ideal gas model using xml data.

### **IdealGasSaveApp**

`IdealGasSaveApp` saves an ideal gas model using xml data.

### **ParticleApp**

`ParticleApp` demonstrates how transfer xml data between programs.

### **ReadNestedXMLApp**

`ReadNestedXMLApp` reads objects into an xml document from an xml file.

### **ReadXMLApp**

ReadXMLApp reads primitive data types into an xml document from an xml file.

### **ScratchPadApp**

ScratchPadApp demonstrates how to cut and paste drawables using the clipboard.

### **ShowXMLApp**

ShowXMLApp reads an xml document and displays the document in an xml tree view.

### **WriteShowNestedXMLApp**

WriteShowNestedXMLApp writes an xml document and displays the document in an xml tree view.

### **WriteXMLApp**

WriteXMLApp writes an xml document to a file.

# CHAPTER

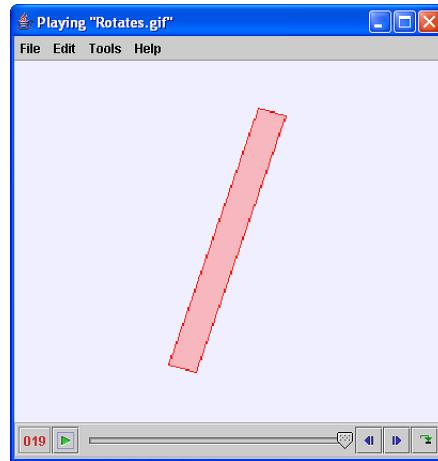
# 13

## Video

©2005 by Doug Brown and Wolfgang Christian, June 20, 2005

The Open Source Physics media API is defined in the `org.opensourcephysics.media.core` package and contains classes for playing, recording and analyzing digital videos. Concrete implementations of this API are available for *animated gif* images and *QuickTime* video formats. The *Tracker* application described in Chapter 16 uses the API described in this chapter to build a full-featured cross-platform video analysis program.

### 13.1 ■ OVERVIEW



**FIGURE 13.1** A VideoPanel can play animated gifs and QuickTime video formats.

Because videos of real-world phenomena or of simulations of phenomena are now common, video tools are playing an increasing important role in physics-education software. In order to meet this need, the Open Source Physics library contains an API for recording and playing of video. This API is defined in the `org.opensourcephysics.media.core` package. Implementations of this API for *animated gif* images and for *QuickTime* movies are available in the gif and the

quicktime sub-packages of the media package, respectively. Because the media API is often used with the Apple QuickTime plug-in and this plug-in may be unavailable on some computers, the media framework is distributed as an optional code library `osp_media.zip` from the OSP website.

### 13.2 ■ ANIMATED GIFS

Creating a simple animated gif is straightforward using the gif media package as shown in Listing 13.1. This example will run on any platform because the animated gif format is supported in the standard Java library.

Listing 13.1 A `GifVideoRecorder` creates *animated gif* images from a sequence of images.

```

1 package org.opensourcephysics.manual.ch13;
2 import java.io.*;
3 import java.awt.image.*;
4 import org.opensourcephysics.display.*;
5 import org.opensourcephysics.media.gif.*;

6
7 public class GifRecorderApp {
8     public static void main(String[] args) {
9         String fileName = "Rotates.gif";
10        // create a drawing panel, frame, and content
11        DrawingPanel panel = new DrawingPanel();
12        panel.setBuffered(
13            true); // buffered panel automatically generates an image
14        DrawableShape rectangle = DrawableShape.createRectangle(0,
15            0, 1.5,
16            15);
17        panel.addDrawable(rectangle); // add rectangle to panel
18        DrawingFrame frame = new DrawingFrame("Creating
19            "+fileName+"",
20            panel);
21        frame.setVisible(true);
22        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
23        // create a gif video recorder
24        GifVideoRecorder recorder = new GifVideoRecorder();
25        try {
26            // create the new video and set the frame duration
27            recorder.createVideo();
28            recorder setFrameDuration(100); // 10 fps
29            // get gif encoder to set optional repeat in browser
30            // (default is 1, 0 repeats continuously)
31            recorder.getGifEncoder().setRepeat(0);
32            // gif encoder can also set transparent color or quality
33            recorder.getGifEncoder().setTransparent(panel.getBackground());
34            // add animation frames to the video

```

## 13.2 Animated Gifs

285

```

32     for(int i = 0;i<20;i++) {
33         rectangle.setTheta(i*Math.PI/10);
34         BufferedImage image = panel.render(); // generates an
35             image because panel is buffered
36         recorder.addFrame(
37             image); // use the displayed image as the video frame
38         try {
39             Thread.currentThread().sleep(100);
40         } catch(InterruptedException ex1) {}
41         // save the completed video file
42         recorder.saveVideo(fileName);
43     } catch(IOException ex) {}
44     panel.setMessage("Animated gif saved.");
45 }
46 }
```

The GifVideoRecorder class is instantiated and its properties are set to create an animated gif with a 100-millisecond frame duration that plays continuously in a video browser. Video frames are images and one way to create these images it to invoke the render method in a DrawingPanel. Notice that the DrawingPanel must be buffered to use this technique. If the buffered option is not set the DrawingPanel does not generate an image and the render method returns null. Display3D panels, on the other hand, are always buffered and the render method will always return an image.

Playing an animated gif is even easier as shown in Listing 13.2. The GifVideo constructor takes a file name and the images in this file are read into an array. These images are then played by the VideoPanel at the frame rate encoded into the animated gif file. VideoPanel is a subclass of InteractivePanel with a video player control as shown in Figure 13.1. A video panel always draws the video as a background behind other drawable objects.

Listing 13.2 GifVideoApp plays *animated gif* images.

```

1 package org.opensourcephysics.manual.ch13;
2 import java.io.*;
3 import org.opensourcephysics.display.*;
4 import org.opensourcephysics.media.core.*;
5 import org.opensourcephysics.media.gif.*;
6
7 public class GifPlayerApp {
8     public static void main(String[] args) {
9         String fileName = "Rotates.gif";
10        DrawingPanel panel = new VideoPanel();
11        DrawingFrame frame = new DrawingFrame(panel);
12        frame.setVisible(true); // will also work with a hidden frame
13        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
14        try { // create and draw a gif video
15            GifVideo animatedGif = new GifVideo(fileName);
```

```

16    panel.addDrawable(animatedGif);
17    frame.setTitle("Playing "+fileName+"");
18 } catch(IOException ex) {}
19 }
20 }
```

### 13.3 ■ PLAYING QUICKTIME VIDEO



**FIGURE 13.2** The QTPlayerApp program displays QuickTime video formats.

The Apple QuickTime player plays the most popular video formats and Apple distributes a QuickTime for Java library with this player that allows us to access the native API. On Windows we recommend that you install QuickTime 7.0 or later (after installing Java) because it automatically installs the necessary library.<sup>1</sup> The latest version of QuickTime may be downloaded from

<http://www.apple.com/quicktime/download/>.

The shared video API allows QuickTime videos to be recorded and played simply by replacing `GifVideoRecorder` and `GifVideo` with `QTVideoRecorder` and `QTVideo` in Listings 13.1 and 13.2. The `QTPlayerApp`, shown in Listing 13.3, adds menu items for loading QuickTime videos with a chooser and for

<sup>1</sup>QuickTime 6.4 or 6.5 can also be used if you select a custom and install all options.

## 13.3 Playing QuickTime Video

287

verifying proper installation of Apple's QuickTime library `QTJava.zip`. If the `verifyQuickTime` method does not find `QTJava.zip` in the Java VM's external library folder (path shown on dialog), search for the file and copy it into this folder<sup>2</sup>. Note that when you update the Java VM the `QTJava.zip` library must be recopied into the new lib/ext folder.

Listing 13.3 `QTVideo` plays video files that are supported by QuickTime.

```

1 package org.opensourcephysics.manual.ch13;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.media.core.*;
4 import org.opensourcephysics.media.quicktime.*;
5 import javax.swing.*;
6 import java.awt.event.*;
7 import java.io.File;
8
9 public class QTPlayerApp {
10     VideoPanel panel = new VideoPanel();
11     DrawingFrame frame = new DrawingFrame("QT Video Player",
12                                         panel);
13
14     QTPlayerApp() {
15         frame.setVisible(true); // will also work with a hidden frame
16         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17         addMenuItems();
18         loadVideo();
19     }
20
21     public void addMenuItems() {
22         JMenu menu = frame.getMenu("File"); // gets menu from toolbar
23         menu.addSeparator();
24         JMenuItem videoItem = new JMenuItem("Load Video");
25         videoItem.addActionListener(new ActionListener() {
26             public void actionPerformed(ActionEvent e) {
27                 loadVideo();
28             }
29         });
30         menu.add(videoItem);
31         JMenuItem verifyItem = new JMenuItem("Verify QuickTime");
32         verifyItem.addActionListener(new ActionListener() {
33             public void actionPerformed(ActionEvent e) {
34                 verifyQuickTime();
35             }
36         });
37         menu.add(verifyItem);
}

```

<sup>2</sup>On Windows computers, the QuickTime installer usually places the `QTJava.zip` library into the `C:/windows/system32` directory. A recent beta of QuickTime 7 placed this library into a subdirectory in `C:/Program Files` folder. *Caveat emptor.*

```

38
39 public void loadVideo() {
40     panel.clear(); // removes all drawables including the old
41     video
42     try {
43         QTVideo video = new QTVideo(); // displays file chooser
44         panel.addDrawable(video);
45         frame.setTitle("Playing "+video.toString());
46     } catch(java.io.IOException ex) {
47         System.out.println("Error creating QTVideo:
48             "+ex.toString());
49     }
50
51     public void verifyQuickTime() {
52         File dir = new File(System.getProperty("java.ext.dirs"));
53         File file = new File(dir, "QTJava.zip");
54         String s = file.exists()
55             ? "QTJava.zip found" : "QTJava.zip not found";
56         JOptionPane.showMessageDialog(frame,
57             s+" in "+dir.getAbsolutePath(), "Verify QuickTime",
58             JOptionPane.INFORMATION_MESSAGE);
59     }
60
61     public static void main(String[] args) {
62         new QTPlayerApp();
63     }
}

```

If the QTVideo class is instantiated without an argument it requests a file name using a chooser dialog. A wide variety of file types can be selected from the chooser including *mpeg* developed by the motion picture experts group, *avi* developed by Microsoft, and *mov* developed by Apple Computer.

### 13.4 ■ CONTROLLING VIDEO DISPLAY

After a video has been instantiated, its properties are set using accessor methods.

```

1 video.setLooping(true);      // turns on looping
2 video.setRate(0.4);          // play() will be in slow motion
3 video.setAngle(Math.PI/6);   // rotate counterclockwise
4 video.setRelativeAspect(2); // stretch the video horizontally
5 video.setWidth(1.3);        // set width in world units to scale
    the video

```

Video frames can be modified before they are displayed by adding filters to the video filter stack. The GhostFilter, for example, overlays images to produce a “live” motion diagram.

## 13.4 Controlling Video Display

289



**FIGURE 13.3** The VideoPropertiesApp program sets video display properties and rotates the video clip.

```
1 video.getFilterStack().addFilter(new GhostFilter());
```

Search the core media package for files matching `*Filter` to find other implementations of the video Filter interface such as DeinterlaceFilter, BrightnessFilter, and GrayScaleFilter.

A VideoPanel can draw videos and other objects in either *image-space* (pixel-space) or *world-space*. When drawing in image-space, the image reference frame (that is, the image itself) is fixed. When drawing in world-space, the world reference frame is fixed. The image reference frame defines positions in pixel units relative to the upper left corner of a video image—that is, the upper left corner of a  $320 \times 240$  video is at  $(0.0, 0.0)$  and the lower right corner is at  $(320.0, 240.0)$ . The DrawingSpacesApp available on the CD demonstrates the relationship between image-space and world-space.

A VideoPanel contains a VideoPlayer object with play control buttons, a slider, and a frame readout. A player's display properties can be set as follows:

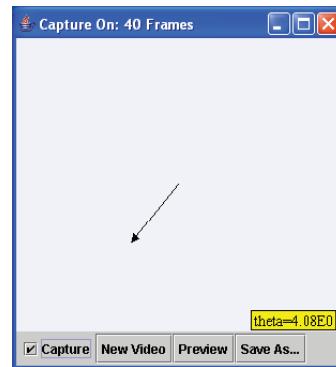
```
1 VideoPanel vidPanel = new VideoPanel(video);
2 VideoPlayer player = vidPanel.getPlayer();
3 player.setInspectorButtonVisible(true);
4 player.setReadoutType("step"); // sets readout type
```

The VideoPlayer class contains a VideoClip object that breaks a video into segments called *clips*. A clip has a start frame and an end frame that can be set. You can also skip frames within a clip when the player's step button is pressed.

```
1 VideoClip clip = player.getVideoClip();
2 clip.setStartFrameNumber(3); // also indirectly sets video start
   frame
3 clip.setStepSize(4);           // also indirectly sets video end
   frame
```

The `VideoPropertiesApp` program, available on the CD and shown in Figure 13.3, uses the code snippets above.

### 13.5 ■ VIDEO CAPTURE TOOL



**FIGURE 13.4** The `VideoCaptureTool` can be added to any buffered drawing panel. The tool captures frames when the panel's `render` method is invoked.

The `VideoCaptureTool` in the `tools` package (Figure 13.4) provides a user interface for creating and saving animation videos in both gif image and QuickTime movie formats. `CaptureAnimationApp`, shown in Listing 13.4, illustrates its use.

Listing 13.4 `CaptureAnimationApp` uses a `VideoCaptureTool` to automatically capture video frames from a drawing panel.

```

1 package org.opensourcephysics.manual.ch13;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.controls.*;
4 import org.opensourcephysics.tools.*;

5
6 public class CaptureAnimationApp extends AbstractSimulation {
7     // create a drawing panel and frame and add an arrow to animate
8     DrawingPanel panel = new DrawingPanel();
9     DrawingFrame frame = new DrawingFrame(panel);
10    Arrow arrow = new Arrow(0, 0, 5, 0);
11    double
12        theta = 0, dtheta = Math.PI/30;
13
14    public CaptureAnimationApp() {
15        panel.addDrawable(arrow);
16        panel.setBuffered(true); // video capture requires buffering

```

## 13.6 Image and Video Analysis

291

```

17    frame.setAnimated(true);
18    frame.setVisible(true);
19    frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
20 }
21
22 protected void doStep() {
23     theta += dtheta;
24     arrow.setXlength(5*Math.cos(theta));
25     arrow.setYlength(5*Math.sin(theta));
26     panel.setMessage("theta="+decimalFormat.format(theta));
27 }
28
29 // creates and shows the video capture tool
30 public void showVideoCaptureTool() {
31     if(panel.getVideoCaptureTool()==null) {
32         panel.setVideoCaptureTool(new VideoCaptureTool());
33     }
34     panel.getVideoCaptureTool().setVisible(true);
35 }
36
37 public static void main(String[] args) {
38     (SimulationControl.createApp(new CaptureAnimationApp(),
39         args)).addButton("showVideoCaptureTool", "Capture");
40 }
41 }
```

Here a DrawingPanel is used to draw a simple animation using an animation control. When a VideoCaptureTool is created and assigned to a buffered DrawingPanel, the VideoCaptureTool will add a frame to the video whenever the panel's render method is invoked and the Capture check-box is selected. The animation must be visible on the screen or the render method will skip creating a new image. This is done for efficiency as there is usually no reason to draw a window if it is iconified or hidden. Captured videos are saved with a standard Save As dialog.

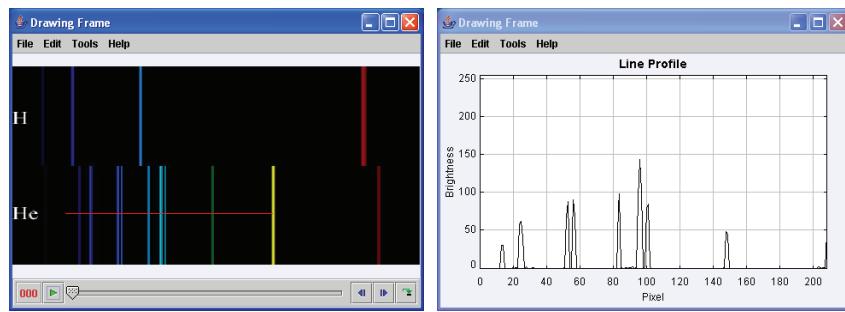
**13.6 ■ IMAGE AND VIDEO ANALYSIS**

A VideoPanel can display common still-image formats. The LineProfileApp shown in Listing 13.5 opens a jpg file showing hydrogen and helium spectra. A cross-section of this image is then displayed using a TLineProfile.

Listing 13.5 LineProfileApp reads brightness values across a row of pixels in an image..

```

1 package org.opensourcephysics.manual.ch13;
2 import java.io.*;
3 import java.awt.*;
4 import javax.swing.*;
```



(a) Spectra.

(b) Line profile.

**FIGURE 13.5** The LineProfileApp program shows an image of hydrogen and helium spectra and a cross section of the image intensity.

```

5 import org.opensourcephysics.display.*;
6 import org.opensourcephysics.media.core.*;
7
8 public class LineProfileApp {
9     VideoPanel vidPanel;
10    Dataset dataset = new Dataset();
11    PlottingPanel plotPanel = new PlottingPanel("Pixel",
12        "Brightness",
13        "Line Profile");
14    TLineProfile profile = new TLineProfile(100, 150, 310, 150);
15
16    public LineProfileApp() {
17        Video video = null;
18        try {
19            video = new ImageVideo("videos/Spectra.jpg");
20        } catch(IOException ex) {}
21        // create the video panel that will also plot the data
22        vidPanel = new VideoPanel(video) {
23            // overrides paintEverything to plot data
24            protected void paintEverything(Graphics g) {
25                super.paintEverything(g);
26                plotData();
27            }
28        };
29        vidPanel.setShowCoordinates(false);
30        vidPanel.setDrawingInImageSpace(true);
31        // add the line profile tool
32        profile.setColor(Color.red);
33        vidPanel.addDrawable(profile);
34        // create the drawing frame

```

## 13.6 Image and Video Analysis

293

```

34 DrawingFrame frame = new DrawingFrame( vidPanel );
35 frame . setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
36 frame . setVisible( true );
37 // create the plotting panel for the profile
38 createPlots();
39 }
40
41 private void plotData() {
42     dataset . clear();
43     int[] data = profile . getProfile();
44     for( int n = 0; n < data . length; n++ ) {
45         dataset . append( n, data [n] );
46     }
47     plotPanel . repaint();
48 }
49
50 private void createPlots() {
51     // make a plot panel to display data
52     plotPanel . setAutoscaleX( true );
53     plotPanel . setPreferredMinMaxY( 0, 255 );
54     DrawingFrame drawingFrame = new DrawingFrame( plotPanel );
55     // make a dataset and set its display properties
56     dataset . setConnected( true );
57     dataset . setMarkerShape( Dataset.NO_MARKER );
58     plotPanel . addDrawable( dataset );
59     drawingFrame . setVisible( true );
60 }
61
62 public static void main( String[] args ) {
63     new LineProfileApp();
64 }
65 }
```

`TLineProfile` is one of many specialized interactive objects created for the *Tracker* video analysis program. It reads image pixels along a path specified by begin and end points. These pixel values are obtained using the `getProfile` method.

The line profile is plotted whenever the `TLineProfile` is moved. One way to do this is to dig into the OSP library and notice that a `DrawingPanel` draws its list of drawable objects by invoking the `paintEverything` method. Because the panel is redrawn after an object is moved we can obtain an up-to-date line profile by overriding the `paintEverything` method and plotting the data after the superclass implementation is invoked. This is done in `LineProfileApp` by creating an anonymous inner class that extends `VideoPanel`.

The `VideoAnalysisApp` program uses other *Tracker* objects to create a small video analysis program that draws graphs of x vs t and y vs t for points that are marked by clicking on the video. It is not show here but available on CD. Although you can adapt this code if your application requires video analysis, a more robust

(and easier) solution may be distribute your program together with Tracker in a Launcher package as described in Chapters 15 and 16.

### 13.7 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch13` package.

#### **CaptureAnimationApp**

`CaptureAnimationApp` uses a `VideoCaptureTool` to automatically capture video frames from a drawing panel as described in Section 13.5.

#### **DrawingSpacesApp**

`DrawingSpacesApp` draws a video in image space and world space as described in Sectipn13.4.

#### **GifPlayerApp**

`GifPlayerApp` described in Section 13.2 plays *animated gif* images.

#### **GifRecorderApp**

`GifRecorderApp` described in Section 13.2 records an *animated gif*.

#### **LineProfileApp**

`LineProfileApp` shows an image of hydrogen and helium spectra and a cross section of the image intensity as described in Section 13.6.

#### **QTPlayerApp**

`QTPlayerApp` described in Section 13.3 displays QuickTime video formats.

#### **VideoAnalysisApp**

`VideoAnalysisApp` draws graphs of x vs t and y vs t for points that are marked by clicking on the video as described in Section 13.6.

#### **VideoPropertiesApp**

`VideoPropertiesApp` sets video display properties and rotates the video clip as described in Section 13.4.

# CHAPTER 14

## Utilities

©2005 by Wolfgang Christian, 20 July 2005

Almost every development project has a small collection of utilities and tricks that save time by providing quick and easy access to common programming tasks. This chapter describes utilities that we have found useful in various OSP projects.

### 14.1 ■ RESOURCE LOADER



**FIGURE 14.1** An image resource can be loaded from almost anywhere.

A program often need access to *resources* such as text, images, or sounds and these resources can be located almost anywhere. Although Java provides a variety of tools for reading files, the API depends on the type of file and on the location of the file. During development a resource file may be in a directory on a local hard drive but this same resource may be placed on a web server or packaged in a jar file after deployment. Unfortunately, the resource must be read differently from each of these locations. The `ResourceLoader` class in the tools package solves this problem by providing a standard API for reading common file types and by searching multiple locations in a predictable search order. We can use the `ResourceLoader` to obtain to obtain the image shown in Figure 14.1 by invoking the static `getResource` method.

```

1 String name = "earth.gif";
2 Resource res= ResourceLoader.getResource(name);
3 Image image= res.getImage();
```

We can use this same syntax it to obtain a text file:

```

1 String name = "orbit_help.html";
2 Resource res= ResourceLoader.getResource(name);
3 String txt= res.getString();
```

Or we can obtain a reference to the resource that can be used later:

```

1 String name = "http://www.opensourcephysics.org/help.html";
2 Resource res= ResourceLoader.getResource(name);
3 URL page= res.getURL();
```

Listing 14.1 shows a program that uses a `ResourceLoader` to read and print the the `data.txt` file located in the directory from which the program was executed.

Listing 14.1 `GetTextApp` loads a text file.

```

1 package org.opensourcephysics.manual.ch14;
2 import org.opensourcephysics.tools.*;
3
4 public class GetTextApp {
5     public static void main(String[] args) {
6         // get string from resource
7         String name = "data.txt";
8         Resource res = ResourceLoader.getResource(name);
9         if(res==null) {
10             System.out.println("Resource not found. Name="+name);
11             return;
12         }
13         System.out.println("path="+res.getAbsolutePath());
14         System.out.println(res.getString());
15     }
16 }
```

When the `getResource` method is invoked from a Java application it searches for the resource in the following order: in directories and subdirectories, on the web, and inside jar archives. The search starts in the directory from which the program was executed. If the resource is not found, it then assumes the given name refers to a url and searches the web. If the resource is again not found it searches within the jar file from which the program was launched. The code remains the same if the `getResource` method is invoked from within an applet. The search begins in directories starting at the the applet's `codebase` and then searches within the jar file(s) specified in the applets `archive` tag. The resource name

## 14.1 Resource Loader

297

can include a relative path to another directory such as `./images/earth.gif` or even to an external site via a URL.

The `getResource` method has a second signature that makes it easy to place resources in directories relative to a class definition. A program can store its resources in the same package (directory) as the byte code (class file) that uses the resource and then pass an appropriate `class` to the loader. The `GetImageApp` program shown in Listing 14.2 uses this syntax to read the `earth.jpg` image from the `ch14` code package.

**CAUTION:** Java compilers and Java development environments may not copy resources from a project's code directory (`src`) to a project's output directory (`bin` or `classes`). Most development environments have a setting that determines which files are copied and you must enable this setting for the resource file types.

Listing 14.2 `GetImageApp` program loads an image from the same location as the `GetImageApp` class.

```

1 package org.opensourcephysics.manual.ch14;
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.frames.*;
4 import org.opensourcephysics.tools.*;
5
6 public class GetImageApp {
7     public static void main(String[] args) {
8         Resource res = ResourceLoader.getResource("earth.jpg",
9                                         GetImageApp.class);
10        MeasuredImage mi = new MeasuredImage(res.getBufferedImage(),
11                                              -1,
12                                              1, -1, 1);
13        DisplayFrame frame = new DisplayFrame("Earth");
14        frame.addDrawable(mi);
15        frame.setVisible(true);
16        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
17    }
}

```

The loader can cache resources in memory so that it can they load quickly in response to subsequent requests. Caching is disabled by default and is enabled as follows:

```
1 ResourceLoader.setCacheEnabled(true);
```

The loader defines utility methods such as `getImage` and `getString` that obtain a resource of a given type using a single method.

```

1 // get image from web
2 String s = "http://www.cabrillo.edu/~dbrown/images/doug48.gif";
3 Image image= ResourceLoader.getImage(s); //gets resource and
   converts it to an image

```

**TABLE 14.1** Resource accessor methods.

<b>org.opensourcephysics.tools.Resource</b>	
getAbsolutePath	The location from where the resource was loaded.
getAudioClip	The audio resource.
getBufferedImage	The image resource converted into a BufferedImage.
getFile	The file associated with this resource.
getIcon	The image resource converted into an ImageIcon.
getImage	The image resource.
getString	The text resource.

Consult the resource loader code and the documentation to learn about additional features such as the ability to add search paths.

Text files such a html and *rich text format* (rft) files are resources that can be displayed in a JEditorPane. We have combined this Swing component with a OSP resource loader to create the TextFrame class. Listing 14.3 uses this frame to display an html page that is stored in the ch14 package.

Listing 14.3 TextFrameApp loads and displays an html page containing the GNU GPL license.

```

1 package org.opensourcephysics.manual.ch14;
2 import org.opensourcephysics.display.*;
3
4 public class TextFrameApp {
5     public static void main(String[] args) {
6         TextFrame frame = new TextFrame("gpl.html",
7             TextFrameApp.class);
8         frame.setVisible(true);
9         frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
10    }
}

```

## 14.2 ■ CUSTOM MENUS

Subclasses of OSPFrame such as ControlFrame and DrawingFrame instantiate menu bars in order provide easy access to common tasks such as saving files. A JMenu bar contains various menu's and these menus are identified by names such as File, Edit, and Help which are listed on the bar. Pressing a name shows a drop-down list of menu items some of which lead to additional submenus. The default menu bar may not be suitable for your application and can be replaced. An alternative to replacement is to add or remove selected items from a menu bar that already exists.

## 14.2 Custom Menus

299

The `OSPFrame` class provides utility methods to delete menus and menu items from a frame using text-labels to identify the object.

```

1 DisplayFrame frame = new DisplayFrame("Empty Frame");
2 frame.removeMenu("Tools"); // deletes an entire
   menu
3 frame.removeMenuItem("File", "Inspect"); // deletes an item from
   a menu

```

The `OSPFrame` class provides methods to retrieve a menu from the menu bar for customization.

```

1 JMenu menu = frame.getMenu("Help");

```

If the requested menu does not exist, a new menu with the given label is created and added to the menu bar. The `JMenu` object that is returned can now be modified by adding menu items and separators. Display panels also have a popup menu and this menu can also be customized. A reference to the popup menu is obtained from the display panel and a menu item is added. Listing 14.4 creates two menu items and adds them to the menu bar and the popup menu. Each item has an action listener that invokes the `showHelp` method.

Listing 14.4 `CustomHelpApp` adds a menu item to the Help menu.

```

1 package org.opensourcephysics.manual.ch14;
2 import org.opensourcephysics.display.*;
3 import java.awt.event.*;
4 import javax.swing.*;
5
6 public class CustomHelpApp {
7     DrawingFrame frame = new DrawingFrame(new DrawingPanel());
8
9     public CustomHelpApp() {
10        // custom help added to Help menu
11        JMenu menu = frame.getMenu("Help");
12        JMenuItem helpItem = new JMenuItem("Custom Help");
13        helpItem.addActionListener(new ActionListener() {
14            public void actionPerformed(ActionEvent e) {
15                showHelp();
16            }
17        });
18        menu.addSeparator();
19        menu.add(helpItem);
20        // custom help added to popup menu
21        JPopupMenu popup = frame.getDrawingPanel().getPopupMenu();
22        helpItem = new JMenuItem("Custom Help");
23        helpItem.addActionListener(new ActionListener() {
24            public void actionPerformed(ActionEvent e) {
25                showHelp();
26            }
27        });
28    }
29
30    void showHelp() {
31        JOptionPane.showMessageDialog(frame, "Custom Help");
32    }
33}

```

```

27    });
28    popup.add(helpItem);
29    frame.setVisible(true); // shows the DrawingFrame
30    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31 }
32
33 public void showHelp() {
34     JOptionPane.showMessageDialog(frame, "This space for rent.", 
35         "Program Help", JOptionPane.INFORMATION_MESSAGE);
36 }
37
38 public static void main(String[] args) {
39     new CustomHelpApp();
40 }
41 }
```

Menus can also be defined and created using xml. An xml document containing *launch nodes* is created as described in Chapter 12 or using LaunchBuilder as described in Chapter 15. The xml file name is then passed to the `parseXMLMenu` method in an `OSPFFrame` as shown in Listing 14.5. The example adds a Documents menu that contains menu items to launch (execute) the `CustomHelpApp` and `GetImageApp` programs from this chapter's code package.

Listing 14.5 XMLMenuApp adds a menu to a frame's menu bar.

```

1 package org.opensourcephysics.manual.ch14;
2 import org.opensourcephysics.display.*;
3
4 public class XMLMenuApp {
5     public XMLMenuApp() {
6         DrawingFrame frame = new DrawingFrame("XML Menus", null);
7         // "custom_menu.xml" is an xml file that was created using
8         // LaunchBuilder.
9         // parseXMLMenu uses the ResourceLoader so that we can do
10        // the following.
11         frame.parseXMLMenu("custom_menu.xml", XMLMenuApp.class);
12         frame.setSize(300, 150);
13         frame.setVisible(true);
14         frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
15     }
16
17     public static void main(String[] args) {
18         new XMLMenuApp();
19     }
20 }
```

### 14.3 ■ CHOOSER

Java provides an easy-to-use file chooser that enable users to browse a disk to open and save files. Users often navigate to a working directory and expect the program to remember this location during subsequent file operations. This behavior is easy to achieve if the same chooser is used over and over because a chooser stores information from session to session.

The `OSPFrame` class instantiates a static file chooser that can be used to access the local file system.

```
1 JFileChooser chooser = OSPFrame.getChooser();
```

When the `getChooser` method is first invoked the chooser opens in a directory specified by the value of the static `chooserDir` variable. We initialize this variable to point to the application's starting directory.

```
1 chooserDir = System.getProperty("user.dir", null);
```

No other action can be taken until the chooser is closed because the chooser is a modal dialog box. If the chooser is closed and `getChooser` is again invoked, the same chooser object is returned. Listing 14.6 uses a chooser to load a resource into a `TextFrame`.

Listing 14.6 ChooserApp opens (reads) a resource using a chooser and displays the file in a `TextFrame`.

```
1 package org.opensourcephysics.manual.ch14;
2 import org.opensourcephysics.controls.*;
3 import org.opensourcephysics.display.*;
4 import javax.swing.JFileChooser;
5
6 public class ChooserApp {
7     TextFrame frame = new TextFrame(null);
8
9     public void choose() {
10         frame.setVisible(true);
11         JFileChooser chooser = OSPFrame.getChooser();
12         if(chooser.showOpenDialog(null)==JFileChooser.APPROVE_OPTION)
13         {
14             OSPFrame.chooserDir =
15                 chooser.getCurrentDirectory().toString();
16             String fileName =
17                 chooser.getSelectedFile().getAbsolutePath();
18             frame.loadResource(fileName, null);
19             OSPFrame.chooserDir =
20                 chooser.getCurrentDirectory().toString();
21         }
22     }
23 }
```

```

20 public static void main(String[] args) {
21     OSPControl.createApp(new ChooserApp()).addButton("choose",
22         "Choose");
23 }
24 }
```

Although the static chooser returned by `getChooser` will open the directory where it was last used, it is sometimes necessary to create custom choosers and these choosers should also open in the directory that was last used. One way to achieve this is to store the current directory in the `chooserDir` variable when a chooser is closed.

```

1 // instantiate a chooser so that it opens in OSPFrame.chooserDir.
2 JFileChooser chooser =new JFileChooser(new
3     File(OSPFrame.chooserDir));
4 // do the following when a chooser is closed so as to set
5     OSPFrame.chooserDir=chooser.getCurrentDirectory().toString();
```

Multiple choosers will not remain “synchronized” but new choosers will at least start in a common directory.

#### 14.4 ■ STATIC METHODS

Open Source Physics simulations typically display drawing frames when an Initialize button is pressed, disable menus when a Run button is pressed, and clear data when a Reset button is pressed. The `GUIUtils` class in the display package defines static methods that perform these and other routine housekeeping chores (see Table 14.2). These methods are based on a static method in the `Frame` class that returns an array containing all `Frames` created by the application.

```
1 Frame[] frames = Frame.getFrames();
```

Methods in the `GUIUtils` class process elements in the array performing the requested action. In order to control the scope of these actions, some method check to see if a property is set. The `renderAnimatedFrames` method, for example, is called from within an animation thread but certain frame may have their `animated` property set to false and these frames will not be rendered to improve performance. Likewise, frames with their `autoclear` property set to false will not loose their data when an Initialize button is pressed.

The `GUIUtils` class also contains methods to create and save images based on the `EpsGraphics2D` package by James Mutton. `EpsGraphics2D` is suitable for creating high quality encapsulated postscript (EPS) graphics for use in documents and papers, and can be used just like a standard `Graphics2D` object.

Many Java programs use `Graphics2D` to draw stuff on the screen, and while it is easy to save the output as a png or jpeg file, it is a little harder to export it as an EPS for including in a document or paper.

**TABLE 14.2** Graphical user interface utility methods are defined in the `GUIUtils` class.

<b>org.opensourcephysics.display.GUIUtils</b>	
<code>clearDrawingFrameData</code>	Clears the data in animated DrawingFrames and repaints the frame's content.
<code>closeAndDisposeOSPFrames</code>	Disposes all OSP frames except the given frame.
<code>enableMenubars</code>	Enables and disables the menu bars in DrawingFrames.
<code>renderAnimatedFrames</code>	Renders all OSPFrames whose animated property is true..
<code>repaintAnimatedFrames</code>	Repaints all OSPFrames whose animated property is true..
<code>showDrawingAndTableFrames</code>	Shows all drawing and table frames.

The `EpsGraphics2D` class makes the whole process extremely easy, because you can use it as if it's a `Graphics2D` object to create eps, jpeg, and png images. If you choose eps, the images that drawn can be resized without leading to any of the jagged edges you may see when resizing pixel-based images, such as jpeg and png files.

The `saveImage` method in the `GUIUtils` class is designed to print OSP components. A component such as drawing panel is passed to the `saveImage` method along with an output file and an output file format. There is an optional third parameter (can be null) that is used by the dialog box. The entire process is extremely simple. Invoking the following method brings up a dialog box that request a file name. The user enters a file name an eps image of the drawing panel's contents is created.

```
GUIUtils.saveImage(drawingPanel, "eps", DrawingFrame.this);
```

## 14.5 ■ INSPECTORS

Open Source Physics components, such as `DrawingPanel` and `ControlFrame`, often have inspectors that provide information about the internal state of an object to the user. Because default inspectors are designed for technical users who are debugging programs or teaching computational physics, they may be inappropriate for a general audience. There are, however, mechanisms for disabling these default inspectors and replacing them with other components. Replacement inspectors are `JFrames` and can therefore be defined using standard Swing or EJS components as shown in Listing 14.7.

Listing 14.7 Default inspectors can be replaced with custom components.

```
package org.opensourcephysics.manual.ch14;
```

```
2 import org.opensourcephysics.display.*;
3 import org.opensourcephysics.ejs.control.GroupControl;
4 import java.awt.Color;
5 import javax.swing.*;
6
7 public class CustomInspectorApp {
8     DrawingPanel panel = new DrawingPanel();
9     DrawingFrame frame = new DrawingFrame(panel);
10    GroupControl control; // reference to the inspector
11
12    public CustomInspectorApp() {
13        panel.setPopupMenu(null); // disable the popup on right click
14        JDialog inspector = getCustomInspector();
15        panel.setCustomInspector(inspector); // inspector for right
16            click
17        frame.setCustomInspector(inspector); // inspector for file
18            menu
19        frame.setVisible(true); // shows the
20            DrawingFrame
21        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22    }
23
24    public void setColor() {
25        if(control.getBoolean("isRed")) {
26            panel.setBackground(Color.RED);
27        } else {
28            panel.setBackground(Color.BLUE);
29        }
30    }
31
32    JDialog getCustomInspector() {
33        control = new GroupControl(this);
34        JDialog dialog = (JDialog) control
35            .add("Dialog",
36                "name=inspector; title=Inspector;
37                    visible=visible; location=300,300;
38                    size=100,50")
39            .getComponent();
40        control.add(
41            "Panel",
42            "name=controlPanel; position=center; parent=inspector");
43        control.add(
44            "CheckBox",
45            "parent=controlPanel; variable=isRed; text=Red;
46                selected=false; action=setColor");
47        return dialog;
48    }
49
50    public static void main(String[] args) {
```

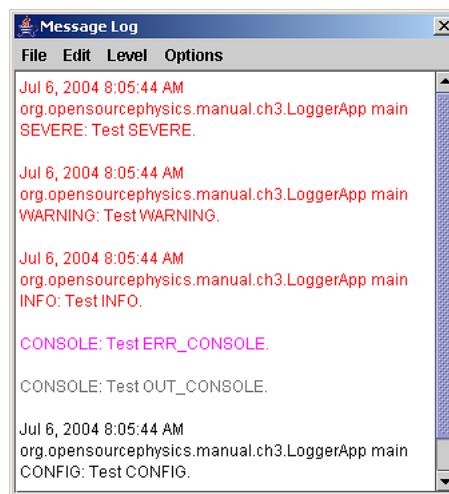
## 14.6 Logging

305

```

45 } new CustomInspectorApp();
46 }
47 }
```

## 14.6 ■ LOGGING



**FIGURE 14.2** An OSPLog displays logger and system console messages in a modal dialog box.

Java 1.4 and above provide a very flexible *logging* API that enables a program to record system information, informational messages, and debugging output. This API is far superior to the old standby `System.out.println`. The `OSPLog` class uses the logging API to create a frame that displays logger output as well as system console output such as error messages.<sup>1</sup> Listing 14.8 shows how to post messages to the OSP logger using various logging levels. The output is shown in Figure 14.2. The `OSPLog` displays the logger's output in a window that allows a user to decide the level of detail using a menu bar. The user may also save the output to a disk file for later analysis.

**Listing 14.8** A `OSPLog` test program showing the various log levels.

```

1 package org.opensourcephysics.manual.ch14;
2 import org.opensourcephysics.controls.ConsoleLevel;
```

<sup>1</sup>The OSP log cannot capture the system console from an applet because the typical applet security policy does not allow a program to redirect the console stream to another device. Other logger messages continue to be logged in applet mode.

```

3| import org.opensourcephysics.controls.OSPLog;
4|
5| public class LoggerApp {
6|     public static void main(String[] args) {
7|         OSPLog.getOSPLog().setVisible(true); // displays the output
8|             window
9|         OSPLog.setLevel(ConsoleLevel.ALL);
10|        OSPLog.severe("Test SEVERE.");
11|        OSPLog.warning("Test WARNING.");
12|        OSPLog.info("Test INFO.");
13|        System.err.println("Test ERR_CONSOLE.");
14|        System.out.println("Test OUT_CONSOLE.");
15|        OSPLog.config("Test CONFIG.");
16|        OSPLog.fine("Test FINE.");
17|        OSPLog.finer("Test FINER.");
18|        OSPLog.finest("Test FINEST.");
19|        // Caution: program will keep running after Dialog window is
20|            closed
21|        // cannot set JFrame to EXIT_ON_CLOSE because OSPLog is a
22|            dialog.
23|
24}

```

Logging does not consume any significant amount of time processing messages below the current level. Allowed levels range from SEVERE to FINEST in the order shown in Listing 14.8. The default level shows messages above Level.OUT\_CONSOLE so that console and error messages are displayed. A common debugging strategy is to log messages at FINE, FINER, and FINEST levels and then examine the log at the appropriate level of detail in order to determine at what point a program or algorithm failed.

## 14.7 ■ PROGRAMS

The following examples are in the `org.opensourcephysics.manual.ch14` package.

### **ChooserApp**

`ChooserApp` uses a file chooser to load a resource as described in Section 14.3.

### **CustomHelpApp**

`CustomHelpApp` demonstrates how to add a menu item to the Help menu as described in Section 14.2.

**CustomInspectorApp**

CustomInspectorApp demonstrates how to create a custom inspector for a drawing panel as described in Section 14.5.

**GetImageApp**

GetImageApp tests the ResourceLoader class by reading an image as described in Section 14.1.

**GetTextApp**

GetTextApp reads the data.txt file from the root of the classpath as described in Section 14.1.

**LoggerApp**

LoggerApp demonstrates the use of the OSPLog class as described in Section 14.6.

**RemoveMenuApp**

RemoveMenuApp removes a menu and a menu item from a DisplayFrame menu bar.

**TextFrame**

TextFrame loads an html frame from the ch14 package directory as described in Section 14.1.

**XMLMenuApp**

XMLMenuApp adds a menu to a frame's menu bar using data in an xml document as described in Section 14.2.

## CHAPTER

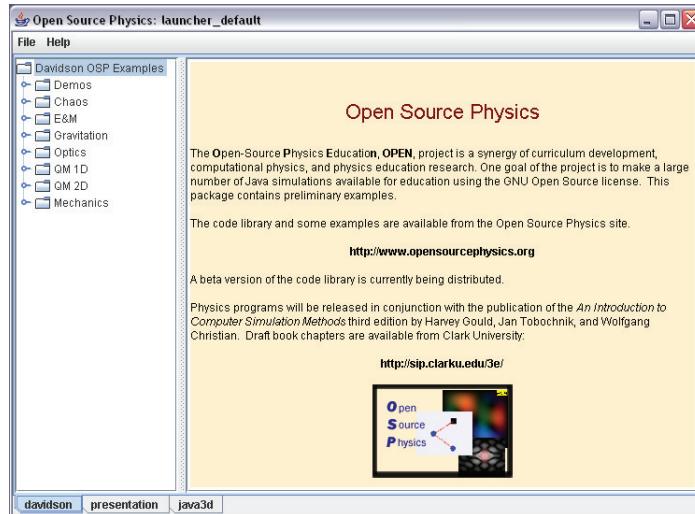
## 15

## Authoring Curricular Material

©2005 by Mario Belloni and Wolfgang Christian, August 2005

We describe Launcher and LaunchBuilder and show how these programs are used to author, organize, and run Java-based curricular material.

### 15.1 ■ OVERVIEW



**FIGURE 15.1** Launcher used to display the organizational structure of the ready-to-run Java-based curricular material from the `osp-guide.jar` file.

As you have already seen in this *Guide*, the Open Source Physics project, with its wide variety of packages, libraries, programs, and files, is a large body of work useful for the study of computational physics. Many instructors, however, do not teach (or do research in) computational physics. In order for these instructors to use OSP material in their courses (or in their educational, experimental, or theoretical research), the various physical models already available must be easily accessible, modifiable, and distributable. The paradigm for authoring, organizing,

and running curricular material described in this Chapter uses the `Launcher` and `LaunchBuilder` programs to accomplish this goal.

The `Launcher` program displays curricular units by using a tree structure to organize the material according to topic, course, etc., as shown in Figure 15.1. Each unit can include html pages, launchable programs, and parameters stored in xml files. Delivering curricular material in `Launcher` packages has several advantages. First, the material can be made self contained and therefore easily distributable (for more on distribution mechanism see Chapter 18). Second, the material is only dependent on having a Java VM on a local machine and not on the type of operating system or browser. This is important as there are at least three to four standard browsers on each operating system and testing curricular material on these ever-changing configurations has become increasingly problematic and time consuming.

The `LaunchBuilder` program creates and organizes the curricular units in `Launcher` packages. Although `Launcher` and `LaunchBuilder` were developed primarily for OSP-based curricular materials, they can be used to launch *any* Java program packaged in a jar file. As shown in Section 15.3, `LaunchBuilder` is an easy-to-understand and easy-to-use editor for the creating and organizing curricular material.

In this chapter, after we describe how to create and organize curricular material, we present examples from three short topical units: classical mechanics (orbits), electromagnetism (radiation from point charges), and quantum mechanics (time evolution of superpositions of states).

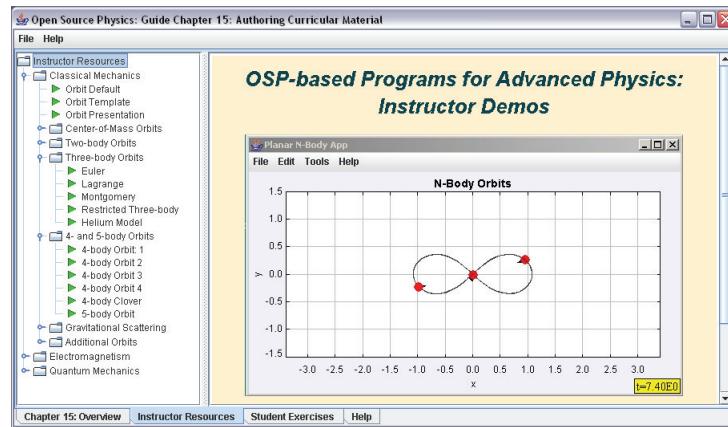
## 15.2 ■ LAUNCHER

`Launcher` enables users to access ready-to-run Java-based curricular material developed by the OSP project and other authors. For a complete listing of the curricular materials distributed with this *Guide*, see Table 1.1 of Chapter 1.

Run the `Launcher` example file, `osp_demo.jar`, by executing it from the command line or double-clicking it in a GUI file browser. `Launcher` automatically reads xml data from a file within the jar file and loads the specified programs and resources. A collection of curricular materials is then displayed in a *TabSet* as shown in Figure 15.2 for the `osp_demo.jar`.

A *TabSet* organizes material into one or more tabs, each of which displays an *Explorer* pane and a *Description* pane within `Launcher`. The *Explorer* pane (left) allows users to easily navigate the tree-based curricular material and launch the associated programs. Selecting a tree node (single click) displays its associated html or text description in the *Description* pane (right). Double-clicking a folder node expands or collapses the contents of that node, while double-clicking a launchable node (depicted with a green arrow) will launch a particular program.

In order to use `Launcher`, a developer creates a Java archive (a jar file) that designates `Launcher` as the target using a *manifest* similar to the one shown in Listing 15.1. The manifest is packaged inside the jar file along with compiled



**FIGURE 15.2** An example of how Launcher is used to organize the ready-to-run Java-based curricular material which appears in `osp_demo.jar`.

Java classes and resources (html pages, xml files, images, and sounds). As the `osp_demo.jar` example shows, all of the resources for a Launcher unit can be packaged inside a single jar file for easy distribution.

**Listing 15.1** A Java manifest that executes the main method in Launcher.

```

1 Manifest-Version: 1.0
2 Built-By: W. Christian
3 Main-Class: org.opensourcephysics.tools.Launcher

```

Curriculum authors need not, however, concern themselves with these details. The core OSP library is packaged and distributed in the `osp.jar` file and this archive already contains a manifest that executes Launcher.

Launcher reads resources using a text-based xml configuration file and these resources may be located externally in other jar files, zip files, or directories. Thus, it is easy to author and distribute Launcher-based curricular material without recompiling Java code or creating jar files. Although Launcher configuration files can be edited using any text editor, they are most easily authored using LaunchBuilder as described in the next Section.

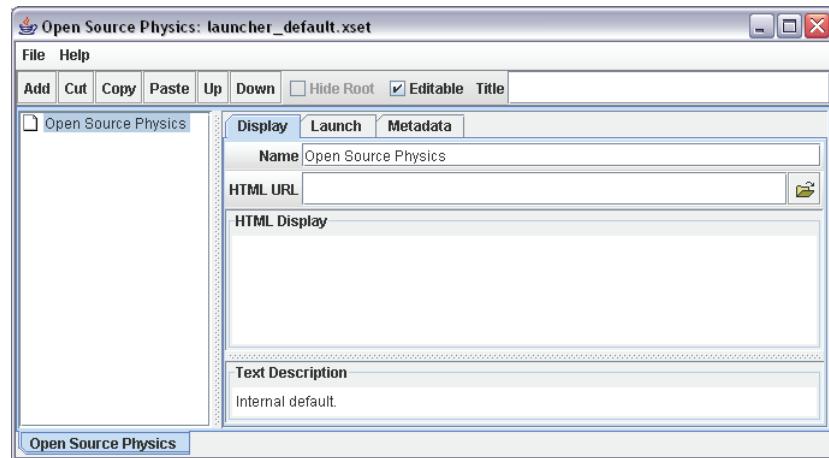
### 15.3 ■ LAUNCHBUILDER

The `osp_demo.jar` file contains over 50 examples of ready-to-run physics curricular material. In order to learn how to author your own material, we will begin with the simpler `osp.jar` file. Run the `osp.jar` file and select the *Edit* menu item under Launcher's *File* menu to invoke LaunchBuilder as shown in Figure 15.3. LaunchBuilder shows the same tree structure as Launcher, but a tool-

bar has been added and the *Description* pane has been replaced with an *Editor* pane with *Display*, *Launch* and *Metadata* tabs.

### Editing Launcher Tabs and Trees

The left-hand side of the LaunchBuilder workspace contains the *Explorer* pane with an xml tree. Click on the root node of the tree then click the “Add” button to add a node. Each node can be associated with a html document by entering a URL in the URL HTML text field found in the *Display* tab of the *Editor* pane. If an html document is not available, you may enter an alternate text narrative in the Text Description field near the bottom of the *Display* tab.



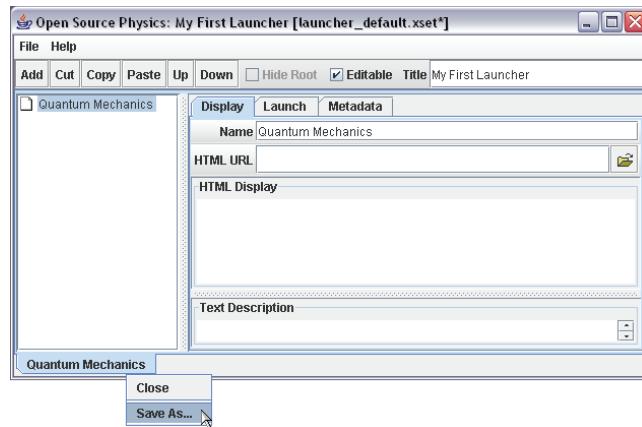
**FIGURE 15.3** Selecting the *Edit* menu item under Launcher’s *File* menu invokes LaunchBuilder. Shown is the basic LaunchBuilder workspace that appears when opening the osp.jar file.

As shown in Figure 15.3, in osp.jar there is a single Launcher tab (displaying the text “Open Source Physics”) with a single root node named “Open Source Physics.”

Select the root and change its name to “Quantum Mechanics” by entering the name in the *Display* tab Name text field (this text field, like all LaunchBuilder text fields, turns yellow when editing; hit the enter key or select another field to apply the change). Note that the LauncherBuilder tab label also changes to reflect the new root name.

Use the Title text field on the toolbar to give your *TabSet* a title (“My First Launcher”). To see how your *TabSet* will look in Launcher, select the *Preview* item from the *File* menu. Close your preview to return to LaunchBuilder.

To save this tab root as an ‘independent’ Launcher node (an xml file), right-click the tab (control-click on Mac) and select the *Save As...* item from the popup



**FIGURE 15.4** The `LaunchBuilder` showing the *Save As...* popup menu for saving tabs as xml files.

menu as shown in Figure 15.4. In the *Save As...* dialog that appears, assign the xml file the name `quantum_mechanics.xml` and save. You should notice that the file `quantum_mechanics.xml` now appears in the same directory as the `osp.jar` file.<sup>1</sup>

Independent nodes represent independent content trees that can be referenced by file name and opened as tab roots (*Open...*) or imported as branches within larger trees (*Import...*). This permits very modular content organization. Independent nodes are identified by a file icon. The color of the icon indicates the type and status of the xml file: yellow is a changed file, white is a writable file, magenta is a jar file, and red is a read-only file.

Now invoke the *Save As...* item under *File* menu as shown in Figure 15.5. This brings up the file-saver dialog, shown in Figure 15.6, that displays a tree showing the *TabSet* files to be saved. All file names are relative to the base path displayed below the tree. To change a name or base path, click in a field or, for the *TabSet* name and base path, click the *Choose...* button to bring up a file chooser. For our current example, simply accept the default base path and the default *TabSet* file name `launcher_default.xset`.

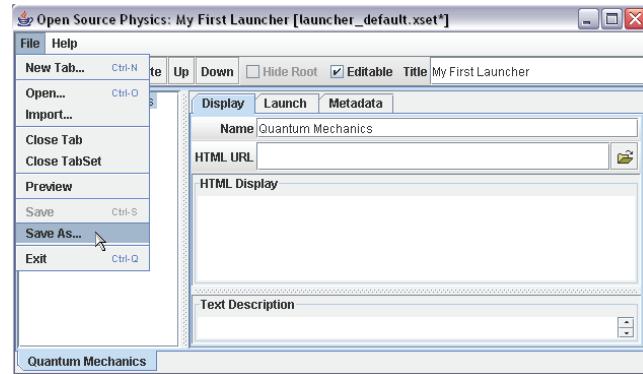
The saver tree allows you to save all the files defined in the current *TabSet* or to combine multiple files into a single ‘self-contained’ file by collapsing a tree node. If you collapse the *TabSet* node then all data will be saved in a single *TabSet* file. File names are not saved.

The saver tree icons also give you information about the files you are saving. Green icons indicate new files, while yellow icons indicate that an existing file

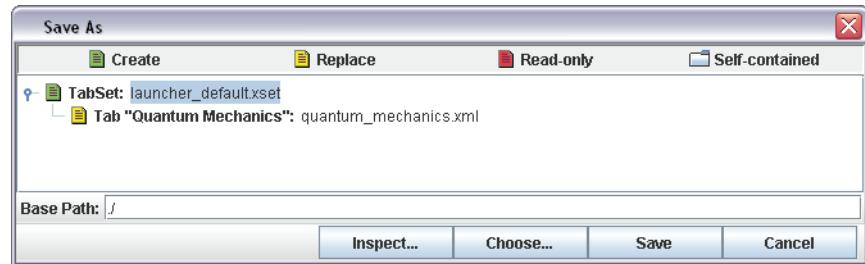
<sup>1</sup>A convenient method for naming a `Launcher` tab involves naming the root and then hiding it. To see how this works, add a child node to the root by clicking the *Add* button and assign the name “QM” to it. Now hide the root node by checking the *Hide Root* box on the toolbar. The “QM” node now appears as the root but the tab name is still “Quantum Mechanics.”

## 15.3 LaunchBuilder

313



**FIGURE 15.5** The *File* menu for LaunchBuilder showing the *Save As...* menu item.



**FIGURE 15.6** The resulting file-saver dialog box from selecting the *Save As...* menu item.

will be overwritten. Red icons identify read-only files for which you must choose a different name or base path before saving.

Close Launcher and LaunchBuilder and re-open `osp.jar`. The configuration files, `launcher_default.xset` and `quantum_mechanics.xml`, that were just created where the jar file is located, will now load as the default, overriding the internal xml files. On startup, both Launcher and LaunchBuilder search for the default file, `launcher_default.xset`, first in the directory containing the jar, then within the jar itself. This search order allows teachers to adapt pre-packaged content to their curricular needs by providing edited external files that override the internal ones. Since the Launcher will look for these files, moving or deleting them will cause the Launcher to load the internal xml files again.

LaunchBuilder can be invoked from within Launcher only if the *TabSet* is saved with the *Editable* option checked on the toolbar. Unchecking this option protects curricular material from unintended modification. LaunchBuilder may also be executed from the command line as shown below, by creating a separate jar file with a manifest that defines LaunchBuilder to be the main class, or by

using Java Web Start.<sup>2</sup>

```
java -classpath osp.jar
org.opensourcephysics.tools.LaunchBuilder
```

Although Launcher and LaunchBuilder accept a command line parameter that can specify any *TabSet* file, it is often useful to give a *TabSet* the default name `launcher_default.xset` as in the example above. *TabSets* with names other than the default can be opened by selecting the *Open...* item from the *File* menu.

### Editing Launcher Nodes

The right-hand side of the LaunchBuilder workspace contains the *Editor* pane which provides three tabs for setting node properties:

**Display tab** contains fields for specifying the name, html page and text description of the selected node. This allows authors to name nodes or folders and specify the html page that will be shown when a launch node or folder is selected. A text description can also be specified.

**Launch tab** contains fields for specifying a jar file containing Java applications, a launchable application within that jar, and an optional xml file with parameters to be loaded when the application is launched. This allows authors to associate with a particular node: a jar file, a program within that jar file, and an xml file with pre-set parameters to load when the node is launched. If an xml file is not specified, the chosen program will use its default parameters when it opens.

**Metadata tab** tab contains fields for specifying the code and/or curriculum author and other metadata useful for web-based search engines and resource collections. This allows authors to embed metadata tags within the xml files associated with the curricular material they are creating.

With your newly created *TabSet* open in Launcher, select the *Edit* menu item to again enter the LaunchBuilder workspace. Select the *Display* tab in the *Editor* pane. Add a child node to the “Quantum Mechanics” node and name it “Harmonic Oscillator” using the Name text field. You must either hit the return button on your keyboard or select another text field for this change to take effect.

Now select the “Harmonic Oscillator” node and add a child named “Two State” to it. Note that “Harmonic Oscillator” has become a folder with “Two State” as its sole content, and that the frame title now displays an asterisk to indicate that the *TabSet* has changed.

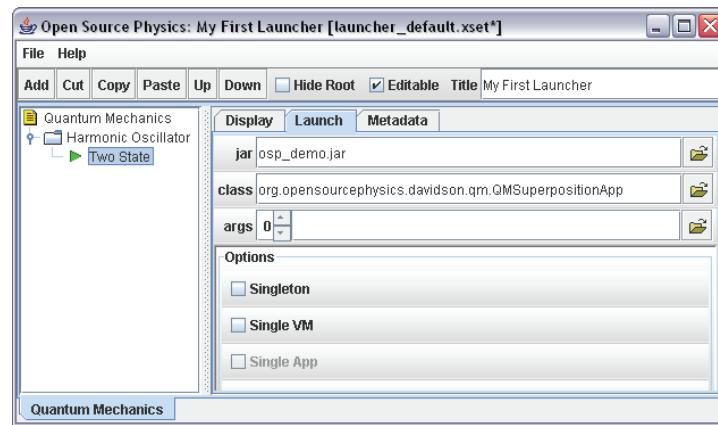
<sup>2</sup>A Java applet is embedded in an html page and depends on a browser to be executed. A Java application is distributed over the web from an html page using *Web Start* and is then independent of the web. The downloaded program has some security restrictions (just like an applet) but is independent of the browser and executes like other programs. In fact, a *Web Start* dialog asks the user if they wish to install an icon on the desktop to access the installed program.

## 15.3 LaunchBuilder

315

To move a node up or down within a given level of the tree, select the node and click the “Up” or “Down” button. Duplicate a node by first copying it to the clipboard (select node and click the “Copy” button) and then pasting it as a child of a different node (select the parent node and click the “Paste” button). Remove or move a node using the “Cut” button.

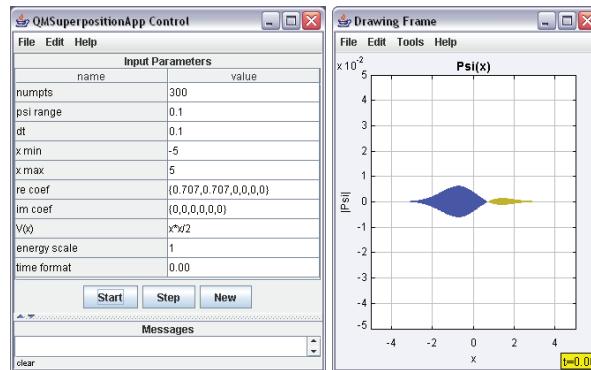
Select the “Two State” node and associate an html document with it by entering a URL in the HTML URL field. If no html document is available, enter a text description in the Text Description field.



**FIGURE 15.7** Using LaunchBuilder to create curricular modules. In “Harmonic Oscillator,” a node, “Two State,” has been added (automatically changing “Harmonic Oscillator” into a folder). With the *Launch* tab selected, Launcher is configured to open *osp\_demo.jar* and access the program *QMSuperpositionApp* when “Two State” is double-clicked.

Next we specify a jar file containing the Java applications. These applications need not be OSP programs although we will assume that is the case in the remainder of this Chapter. With the node “Two State” selected (highlighted), select the *Launch* tab in the *Editor* pane. Note the *jar* and *class* text fields which are used for specifying a jar file name and a launch class name, respectively. Also note that the *class* text field is initially disabled since no jar file has been specified. Use the browse button to the right of the *jar* text field to select the *osp\_demo.jar* file. The *class* text field is now enabled and you can now browse the contents of the jar file to select any launchable class (class with a main method). Select the *org.opensourcephysics.davidson.qm.QMSuperpositionApp* class. The “Two State” node now displays a green arrow icon to indicate that it executes a Java program. Double-click the node to execute the program. You must also click the “Initialize” button (turning it into a “Start” button) in the control frame, as shown in Figure 15.8. Save your work.

The jar text field allows authors to specify more than one jar file (as shown in Figure 15.7 which uses the `osp_demo.jar`), and child nodes inherit jar files from their parents. In other words, if a jar file is specified at the root of the Launcher configuration, that jar file is added to the classpath for all nodes in the tree. Hence, if we were to populate a folder or collection with programs from the same jar file, it would make sense to add the jar at the root of the collection.



**FIGURE 15.8** The result of launching the node “Two State.”

Save all files, exit LaunchBuilder, and re-open Launcher. Click on the “Two State” node to run the program. We now examine the xml files that were created.

### Launcher and XML Files

Launcher recognizes both xml and xset file extensions. Both extensions are used to designate a file as an xml document. All Launcher xml files use the standard OSP format (see Chapter 12). This format uses an `<object>` tag with “class” attribute to save a Java object and `<property>` tags with “name” and “type” attributes to save the object’s properties. Since a `<property>` can be of type `<object>`, object and property tags are nested.

The xset extension signals that the document contains complete *TabSet* data. The `launcher_default.xset` document describing “My First Launcher” is shown in Listing 15.2. To view this file within Launcher, open the saver dialog (*File | Save As...*), select the *TabSet* node and click the *Inspect...* button. An xml inspector will then open that shows the xml file organized into a tree structure on the left with the corresponding xml text on the right.

**Listing 15.2** An xset document clears the Launcher and loads new material.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <object class="org.opensourcephysics.tools.Launcher$LaunchSet">
3   <property name="title" type="string">My First
      Launcher </property>

```

## 15.3 LaunchBuilder

317

```

4   <property name="editor_enabled" type="boolean">true</property>
5   <property name="width" type="int">650</property>
6   <property name="height" type="int">300</property>
7   <property name="divider" type="int">160</property>
8   <property name="launch_nodes"
9     type="collection" class="java.util.ArrayList">
10    <property name="item" type="string">
11      quantum_mechanics.xml
12    </property>
13  </property>
14</object>
```

Inspect the *TabSet* xml, shown in Listing 15.2. Note that it specifies a collection of launch nodes to be loaded into tabs. Here, there is a single launch node, and the data saved is simply the file name of the independent Launcher file that defines the “Quantum Mechanics” tab. Also note the other properties: “title,” “editor\_enabled,” “width,” “height,” and “divider.” When Launcher opens a *TabSet* it first closes any previous tabs, then loads the new tabs, resizes and titles the frame, and sets the divider location based on these values.

Documents with an xml extension describe an ‘independent’ launch node that can be opened in a tab or attached to an existing tree in LaunchBuilder. Inspect the *quantum\_mechanics.xml* file, shown in Listing 15.3. You should recognize the names and other properties of the nodes as well as their parent/child relationships.

With the xml inspector open, again select the *TabSet* node and note how the xml changes when you collapse it, so it is self-contained. All of the data that was previously in the *quantum\_mechanics.xml* file is now in the *TabSet* file.

**Listing 15.3** An xml document describes an xml tree that can be loaded into a Launcher.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <object class="org.opensourcephysics.tools.LaunchNode">
3   <property name="name" type="string">Quantum
4     Mechanics</property>
5   <property name="description" type="string"></property>
6   <property name="child_nodes" type="collection"
7     class="java.util.ArrayList">
8     <property name="item" type="object">
9       <object class="org.opensourcephysics.tools.LaunchNode">
10        <property name="name" type="string">Harmonic
11          Oscillator</property>
12        <property name="child_nodes" type="collection"
13          class="java.util.ArrayList">
14          <property name="item" type="object">
15            <object class="org.opensourcephysics.tools.LaunchNode">
16              <property name="name" type="string">Two State </property>
                <property name="launch_class" type="string">
                  org.opensourcephysics.davidson.qm.QMSuperpositionApp
```

```

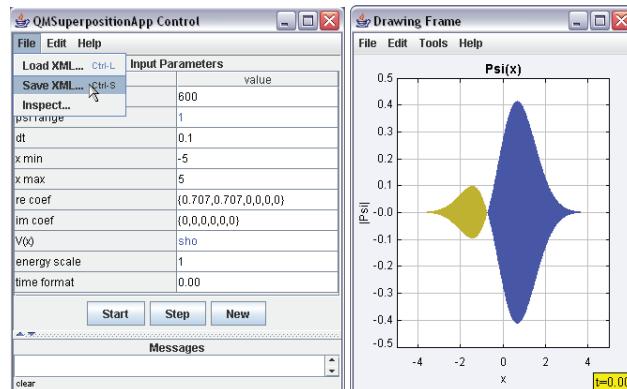
17      </property>
18      <property name="classpath"
19          type="string">osp-demo.jar </property>
20  </object>
21  </property>
22  </property>
23  </property>
24  </property>
25 </object>

```

### Saving and Loading Application Data

The *Launch* tab in the *LaunchBuilder* workspace also provides an *args* field for specifying one or more (usually using the [0] value) command line arguments passed to the main method of the launch class. Generally, the *args[0]* parameter is used to specify the name of an xml document containing initialization parameters. This allows a single Java program to be used with many different parameter sets.

The easiest way to create such an xml file is by using of a program's control frame. Open *Launcher* and double-click the "Two State" node. Make some changes to the text fields of the control frame. Under the control frame's *File* menu select the *Save XML...* item as shown in Figure 15.9.



**FIGURE 15.9** The node "Two State," has been launched yielding the following program control (left) and view (right). The default configuration has been altered and is in the process of being saved as an xml file as shown by the *Save XML...* dialog.

Save your xml file as `test.xml`. This xml file can be loaded two ways. First, use the *Load XML...* item under the control frame's *File* menu to load the file directly. Second, use the *args[0]* text field (as shown in Figure 15.7) in

## 15.3 LaunchBuilder

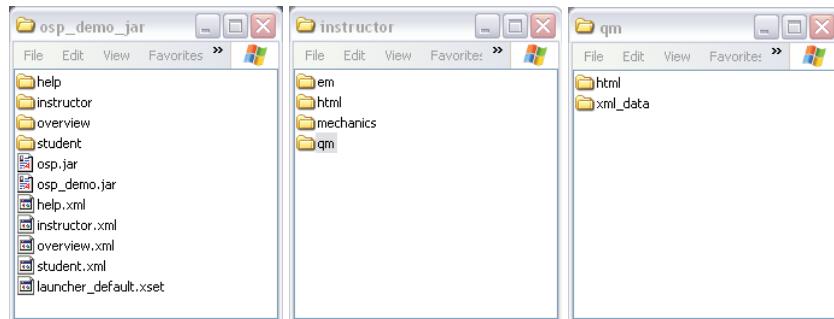
319

the *Launch* tab of the LaunchBuilder workspace to specify that the parameters saved in the `test.xml` file will automatically load the next time the node “Two State” is selected in Launcher. You must, of course, save this change.

In addition, there are several options that control the runtime environment for the launch class. The Singleton option allows only one running instance for a given node. The Single VM option instantiates the application in the same Java VM that is executing Launcher rather than a separate VM. (Note that because of security restrictions, single VM mode must be selected if Launcher is used to distribute curricular material from a server as an applet or using Java Web Start technology.) The Single App option allows only one application to run at a time—a running application is terminated when the user double-clicks a node to launch another. The Show Log option displays a console-like log window containing error and information messages when the application starts or runs. This option is useful for debugging and for some types of output data. A log level can be specified to log different levels of detail from ‘finest’ to ‘severe.’ *Launch* nodes inherit runtime options from their parent, so selecting an option at the root will apply to all nodes.

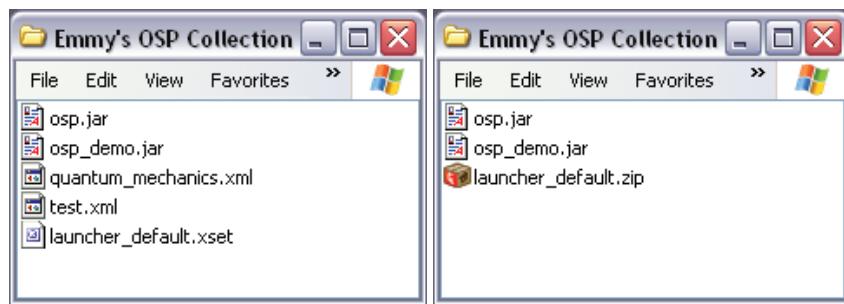
### Organizing and Distributing Material

If you have followed the creation of curricular materials using LaunchBuilder described thus far, you should have three files in the same directory as your jar files: `launcher_default.xset`, `quantum mechanics.xml`, and `test.xml`. These xml files supersede the internal xml files contained within the jar allowing teachers to author their own material with LaunchBuilder. While this is convenient, quickly the issue of organization becomes important. At the moment there are only three xml files to worry about. However, in time there may be tens or hundreds of xml files for *TabSets*, xml files for saved program data, and html files for descriptions. How should we organize then?



**FIGURE 15.10** The internal file structure of the `osp_demo.jar` file. On the left, the root directory structure is shown; in the center, the subdirectories of the `instructor` directory showing folders organized according to topic; on the right, the contents of the `qm` directory revealing `xml_data` and `html` subdirectories.

The simplest directory structure organizes the html and xml data files in separate subdirectories, while leaving the xset file, `launcher_default.xset`, and `TabSets` at the root where the jar file is located. Such a directory structure is shown in Figure 15.10 for the internal file structure of the `osp_demo.jar` file. Notice that at the root there are the jar files which were used to create the `TabSets` and xml data, and to associate html pages with the created material. Also note the xset at the root and the xml files that store `TabSet` information to produce the results shown in Figure 15.2. Within the `qm` directory are the `xml_data` and `html` directories associated with the quantum mechanics curricular material.



**FIGURE 15.11** Two directory structures that yield the same result when double-clicking the `osp.jar` file. The different structures emphasize the use of a single zip file, `launcher_default.zip`, containing the xml and xset files.

Figure 15.10 shows a rather extensive directory structure, which often makes distribution difficult. One way to simplify distribution is to compress all xml and xset files into a single zip file called `launcher_default.zip`. If these files reside in subdirectories or if there are html description files, these should be compressed in `launcher_default.zip` as well. The `osp.jar` and `osp_demo.jar` files look for this filename and then reads the files contained in it. In order to be read by the jar file, these files should be compressed directly and not placed in a folder prior to compressing. Figure 15.11 shows the resulting files as a result of compressing xml and xset files. Compressing the files and folders shown in Figure 15.10 would yield the same resulting structure with a different `launcher_default.zip` file. In order to see the pre-packaged directory structure of `osp_demo.jar`, which is similar to that shown in Figure 15.10, expand (unzip) this jar file and look at its contents.

## 15.4 ■ CURRICULUM DEVELOPMENT OVERVIEW

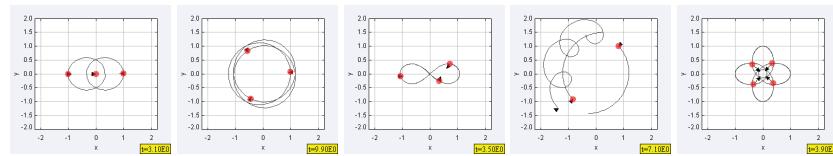
To accompany this Chapter we have created curricular units for classical mechanics (orbits), electromagnetism (radiation from point charges), and quantum

## 15.5 Classical Mechanics: Orbits

321

mechanics (time evolution of superpositions of states). This material can be accessed by double-clicking the file `osp_demo.jar`. We briefly begin each Section with a review of the theory behind the phenomena we are simulating, then briefly describe the programs used and the curricular unit. A fuller description of the programs (including all of the editable parameters) and the curricular material can be found in the “Overview” and “Help” tabs in the Launcher that opens when accessing `osp_demo.jar`.

## 15.5 ■ CLASSICAL MECHANICS: ORBITS



**FIGURE 15.12** Classical orbit examples from the `osp_demo.jar` file.

One of the standard topics in classical mechanics, at any level, is that of Kepler’s laws of planetary motion. This topic rests on Newton’s law of universal gravitation,

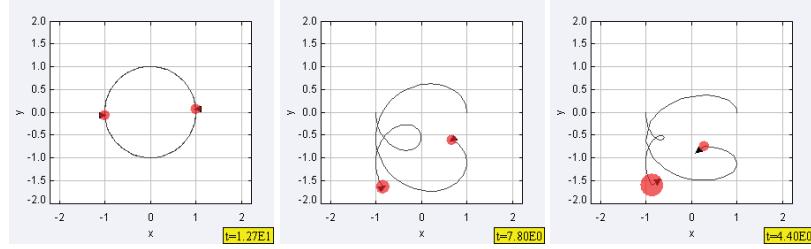
$$\mathbf{F}_{12} = -Gm_1m_2 \frac{\mathbf{r}_{12}}{r_{12}^3}, \quad (15.1)$$

for the gravitational force on mass  $m_1$  due to mass  $m_2$ , given a separation  $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2| = \sqrt{r_1^2 + r_2^2 - 2\mathbf{r}_1 \cdot \mathbf{r}_2}$ , and where  $G = 6.67 \times 10^{-11} \text{ N m}^2/\text{kg}^2$ . The force is attractive and lies along the line separating the two masses.

Kepler’s laws can be simply expressed as (1) planets move in ellipses with the Sun at one focus (2) planets sweep out equal areas in equal times as it orbits the Sun, and (3) the square of the period of a planet’s orbit is proportional to the cube of the semimajor axis,  $a$ , of its orbit:  $T^2 = (\frac{4\pi^2}{GM}) a^3$ .

Introductory treatments of Kepler’s laws focus on the special cases of two-body systems where the central object is much more massive than the orbiting body, which itself usually orbits in circular motion. More advanced treatments include the general study of elliptical orbits and occasionally cases where the objects are nearly the same mass. Less frequently, systems with more than two objects are considered, and for good reason. These systems are, in general, not exactly solvable. However, as we shall see, there are special cases in which the three-, four-, and five-body problems can be solved as shown in Figure 15.12.

The simplest Kepler problem involves just two objects with masses  $m_1$  and  $m_2$ . These objects are attracted to each other via Eq. (15.1) and there is usually some motion of both objects as shown in Figure 15.13. Each object is given a



**FIGURE 15.13** Classical orbit examples from the `osp_demo.jar` file which show three different binary systems' orbits.

position vector relative to an origin, but it is the *separation vector* or *relative coordinate*,  $\mathbf{r} = \mathbf{r}_{12} = \mathbf{r}_1 - \mathbf{r}_2$  that appears in the gravitational force,  $\mathbf{F} = -Gm_1m_2\frac{\mathbf{r}}{r^3}$ , and the gravitational potential energy,  $U(r) = -\frac{Gm_1m_2}{r}$ , of the system.

We can naively write the Lagrangian for the system:

$$\mathcal{L} = \frac{1}{2}m_1\dot{\mathbf{r}}_1^2 + \frac{1}{2}m_2\dot{\mathbf{r}}_2^2 - U(r), \quad (15.2)$$

which is given in terms of three coordinates, although there are just two degrees of freedom. We use *center-of-mass coordinates*

$$\mathbf{R} = \frac{m_1\mathbf{r}_1 + m_2\mathbf{r}_2}{m_1 + m_2}, \quad (15.3)$$

and the total mass of the system,  $M = m_1 + m_2$ , to write  $\mathbf{r}_1 = \mathbf{R} + \frac{m_2}{M}\mathbf{r}$  and  $\mathbf{r}_2 = \mathbf{R} - \frac{m_1}{M}\mathbf{r}$ , a substitution which allows the Lagrangian to obtain the form:

$$\mathcal{L} = \frac{1}{2}M\dot{\mathbf{R}}^2 + \frac{1}{2}\mu\dot{\mathbf{r}}^2 - U(r), \quad (15.4)$$

where  $\mu = \frac{m_1m_2}{m_1+m_2}$  is the reduced mass and  $U(r) = -\frac{G\mu M}{r}$ . Eq. 15.4 suggests that the system behaves as if there were two fictitious particles: one of mass  $M$  that moves with the center of mass and another of mass  $\mu$  that moves with a speed of the relative position as shown in Figure 15.14.

We can use this idea to re-write the Lagrangian as  $\mathcal{L} = \mathcal{L}_{\text{rel}} + \mathcal{L}_{\text{cm}}$ , where

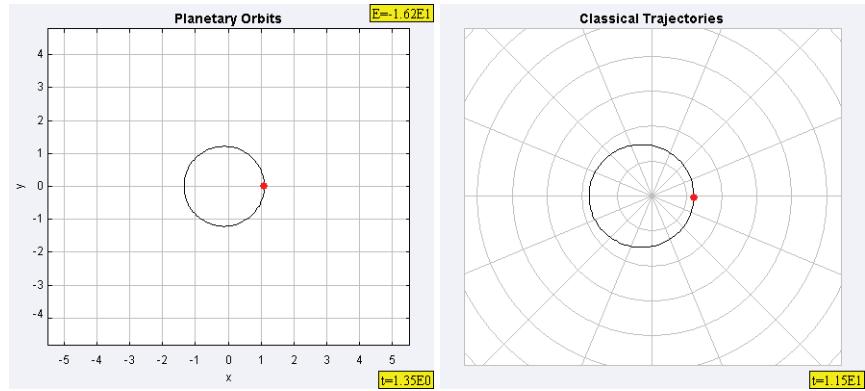
$$\mathcal{L}_{\text{cm}} = \frac{1}{2}M\dot{\mathbf{R}}^2 \quad \text{and} \quad \mathcal{L}_{\text{rel}} = \frac{1}{2}\mu\dot{\mathbf{r}}^2 - U(r). \quad (15.5)$$

For the center-of-mass motion,  $\ddot{\mathbf{R}} = 0$ , which means that  $\dot{\mathbf{R}} = \text{constant}$  and for the equation of motion of the relative coordinate, we have

$$\mu\ddot{\mathbf{r}} = -\nabla U(r). \quad (15.6)$$

## 15.5 Classical Mechanics: Orbits

323



**FIGURE 15.14** Classical orbit examples from the `osp_demo.jar` file which show center-of-mass motion in both Cartesian and polar coordinates.

If we choose a frame of reference in which  $\dot{\mathbf{R}} = 0$ , such as shown in Figure 15.14, the entire Lagrangian is just:

$$\mathcal{L} = \frac{1}{2}\mu\dot{\mathbf{r}}^2 - U(r) , \quad (15.7)$$

and the resulting motion will lie on a plane that we can identify as the  $x$ - $y$  plane. We can now write the Lagrangian in polar coordinates as:

$$\mathcal{L} = \frac{1}{2}\mu\dot{r}^2 + \frac{1}{2}\mu r^2\dot{\phi}^2 - U(r) . \quad (15.8)$$

For the  $\phi$  coordinate we find the equation of motion,  $\mu r^2\ddot{\phi} = 0$ , or that  $\mu r^2\dot{\phi} = L = \text{constant}$ . Here,  $L$  is the  $z$  component of the angular momentum and it is a constant.

The radial equation of motion for the  $r$  coordinate gives:

$$\mu\ddot{r} = \frac{L^2}{\mu r^3} - \frac{G\mu M}{r^2} . \quad (15.9)$$

The first term on the right-hand side is called the centrifugal force,  $F_{cf} = \mu r\dot{\phi}^2 = \frac{L^2}{\mu r^3}$ , and the second term on the right-hand side is the gravitational force.

To show that these orbits are in general elliptical, we are interested in the *trajectory* plot,  $r(\phi)$ , for an arbitrary orbit. We solve for the time parameter in order to make substitutions that eventually get rid of  $t$ . We begin by using the substitution,  $u = \frac{1}{r}$ , which after taking time derivatives and simplifying, yields the differential equation:

$$\frac{d^2u}{d\phi^2} + u = \frac{Gm^2M}{L^2} , \quad (15.10)$$

which has the solution that  $u = A \cos(\phi - \phi_0) + \frac{Gm^2M}{L^2}$  or in terms of  $r$  as:

$$r(\phi) = \frac{1}{A \cos(\phi - \phi_0) + \frac{Gm^2M}{L^2}}. \quad (15.11)$$

If we set  $\phi_0 = 0$  and rewrite, we find that

$$r(\phi) = \frac{1}{A [\cos(\phi) + \frac{Gm^2M}{AL^2}]}, \quad (15.12)$$

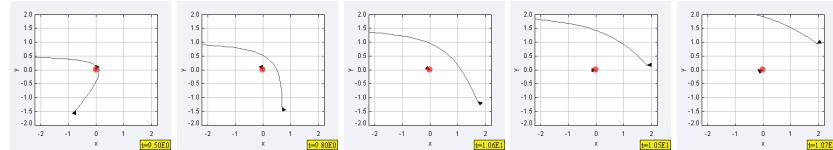
where we easily find the maximum (apogee) and minimum (perigee) value of this expression, and also find that the eccentricity is  $e = \frac{AL^2}{Gm^2M}$ .

In addition to closed orbits, we may also consider scattering (unbound) orbits. Particles incident on a cross sectional area  $d\sigma$  will scatter an angle  $d\Omega$ . The differential cross-section is defined as:  $\mathcal{D}(\theta) = \frac{d\sigma}{d\Omega}$ . In terms of the impact parameter,  $b$ , we have that  $d\sigma = db b d\phi$  and  $d\Omega = \sin(\theta) d\theta d\phi$  and therefore the differential cross-section becomes

$$\mathcal{D}(\theta) = \frac{d\sigma}{d\Omega} = \frac{b}{\sin(\theta)} \left| \frac{db}{d\theta} \right|. \quad (15.13)$$

We find for the entire cross-section:

$$\sigma = \int \frac{d\sigma}{d\Omega} d\Omega = \int \mathcal{D}(\theta) d\Omega. \quad (15.14)$$



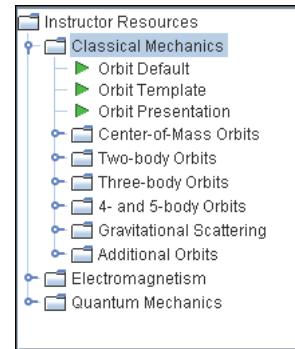
**FIGURE 15.15** Classical gravitational scattering examples from the `osp_demo.jar` file which show ever-increasing impact parameters.

For gravitational scattering, we use the fact that the radial position of the scatterer,  $r$ , is related to angle by Eq. (15.11). We begin by evaluating  $\frac{1}{r}$  as given by Eq. (15.11) and solve for the parameter  $A$  using the energy at the particle's extrema (usually  $r = -\infty$ ) in its unbounded orbit. In doing so, after much algebra and trigonometry, we find that the impact parameter,  $b$ , can be written in terms of  $\Theta$ , the scattering angle (the total angle scattered), as

$$b = \frac{GmM}{2E} \cot\left(\frac{\Theta}{2}\right), \quad (15.15)$$

where  $E$  is the energy of the scattered object. Examples of gravitational scattering and the relationship between  $b$  and  $\Theta$  given by Eq. (15.15) are shown in Figure 15.15.

### Programs and Curricular Materials



**FIGURE 15.16** The *Explorer* pane of the *Launcher* from the `osp_demo.jar` showing the classical mechanics curricular materials.

The classical mechanics materials use the following three OSP programs which are found in the “Davidson package” (`org.opensourcephysics.davidson`):

`gravitation.PlanetApp` shows a planet orbiting a massive central star in Cartesian coordinates. The initial position ( $x, y$ ) and velocity ( $v_x, v_y$ ) of the orbiting mass can be set.

`gravitation.ClassicalApp` shows a planet orbiting a massive central star in polar coordinates. The initial position ( $r, \phi$ ) and velocity ( $v_r, v_\phi$ ) of the orbiting mass can be set.

`nbody.OrbitApp` shows two or more objects that are gravitational attracted to each other. Any number of objects can be added with any mass, initial position, and initial velocity.

The curricular material is organized into six units:

**Center-of-Mass Orbits** show variations on a planet orbiting a much more massive central star in Cartesian and polar coordinates. The examples show one circular and two elliptical orbits that begin at the same position, yet have different initial velocities. It is the difference in initial velocity that accounts for the difference in orbital trajectory around the central star.

**Two-body Orbits** show a variety of two-body orbits where the masses of the two objects are comparable (unlike above). One set of examples varies one of the object’s mass, while the other varies the initial position of one of the masses.

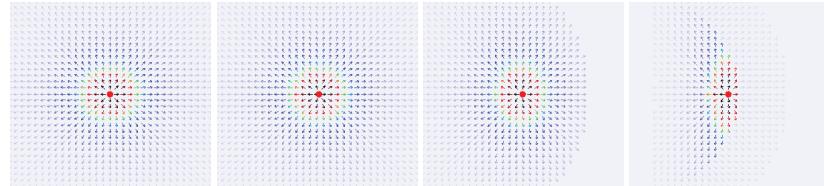
**Three-body Orbits** show four three-body orbits. The *Euler* example shows a special solution determined by Euler in which all three masses always lie on a straight line. The *Lagrange* example shows a special solution determined by Lagrange in which the objects maintain a geometric relationship to each other as vertices of a polygon. In this case, the three objects are always at the vertices of an isosceles triangle (whose sides can change length). The *Montgomery* example shows a stable orbit of three objects that orbit each other in a figure-eight pattern with all three objects tracing out the exact same orbital trajectory. The *Restricted Three-body* example shows an example of this class of exactly-solvable problems. Two objects with the same mass are in orbit around each other. A third, much less massive, object also orbits, following one object around its orbit.

**4- and 5-body Orbits** show a variety of unstable orbits with four and five masses all with similar masses with all three objects tracing out the exact same orbital trajectory. The 4-body *Clover* example is one of Lagrange type where the four objects are always at the vertices of a square (whose sides can change length).

**Gravitational Scattering** shows a variety of examples of unbounded scattering of an object in the gravitational field of a more massive object. The impact parameter varies in each simulation.

**Additional Orbits** shows center-of-mass orbits of a wider variety than that of the Center-of-Mass Orbits unit.

## 15.6 ■ ELECTROMAGNETISM: RADIATION



**FIGURE 15.17** Electromagnetic radiation examples from the `osp_demo.jar` file which show the electric field vectors (in the  $xy$  plane) of a positively charged particle at increasing constant velocities.

Maxwell's equations are written in MKS units as:

$$\nabla \cdot \mathbf{E} = \rho/\epsilon_0 , \quad \nabla \cdot \mathbf{B} = 0 , \quad (15.16)$$

## 15.6 Electromagnetism: Radiation

327

$$\nabla \times \mathbf{E} + \frac{\partial \mathbf{B}}{\partial t} = 0, \quad \nabla \times \mathbf{B} - \epsilon_0 \mu_0 \frac{\partial \mathbf{E}}{\partial t} = \mu_0 \mathbf{J}, \quad (15.17)$$

where we choose to write these equations with all of the fields on one side and all the true sources on the other. For a variety of reason, the calculation of time-dependent fields is easier in the potential formalism. In this formalism,  $\mathbf{E} = -\nabla\phi - \frac{\partial \mathbf{A}}{\partial t}$  and  $\mathbf{B} = \nabla \times \mathbf{A}$ , where  $\phi$  is the electric (scalar) potential and  $\mathbf{A}$  is the magnetic (vector) potential. Making these two substitutions in Eq. (15.17), using the double-curl rule, and simplifying, we are left with only two equations:

$$-\nabla^2\phi - \frac{\partial}{\partial t}(\nabla \cdot \mathbf{A}) = \rho/\epsilon_0, \quad (15.18)$$

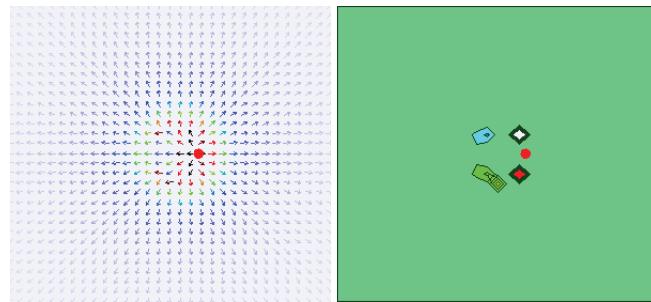
and

$$-\nabla^2\mathbf{A} + \nabla(\nabla \cdot \mathbf{A}) + \epsilon_0 \mu_0 \nabla \left( \frac{\partial \phi}{\partial t} \right) + \epsilon_0 \mu_0 \frac{\partial^2 \mathbf{A}}{\partial t^2} = \mu_0 \mathbf{J}. \quad (15.19)$$

If we choose the Lorentz gauge,  $\epsilon_0 \mu_0 \frac{\partial \phi}{\partial t} + \nabla \cdot \mathbf{A} = 0$ , Eqs. (15.18) and (15.19) reduce nicely to

$$\left[ \epsilon_0 \mu_0 \frac{\partial^2}{\partial t^2} - \nabla^2 \right] \phi = \rho/\epsilon_0 \quad \text{and} \quad \left[ \epsilon_0 \mu_0 \frac{\partial^2}{\partial t^2} - \nabla^2 \right] \mathbf{A} = \mu_0 \mathbf{J}, \quad (15.20)$$

which are wave equations for electromagnetic waves, with speed  $c = \frac{1}{\sqrt{\epsilon_0 \mu_0}} = 3 \times 10^8$  m/s.



**FIGURE 15.18** Electromagnetic radiation example from the `osp_demo.jar` file which shows the electric field vectors and magnetic field lines of a positively charged particle changing velocity.

We find that for static cases, the equations in Eq. 15.20 reduce to:

$$\nabla^2\phi = -\rho/\epsilon_0 \quad \text{and} \quad \nabla^2\mathbf{A} = -\mu_0 \mathbf{J}, \quad (15.21)$$

which have the solutions:

$$\phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \int \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} dV' \quad \text{and} \quad \mathbf{A}(\mathbf{r}) = \frac{\mu_0}{4\pi} \int \frac{\mathbf{J}(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} dV'. \quad (15.22)$$

In the static case, the potentials and fields do not change. However, when charge distributions and current distributions change, the potentials and fields must also change. Where we measure the potentials and fields,  $\mathbf{r}$ , there is, however, a time delay of the signal that the charge and current distributions located at  $\mathbf{r}'$  have changed. This signal travels at the speed of light, and hence the relationship between the time associated with the charge and current distributions (the cause),  $t'$ , and the time when we measure the potentials and fields (the effect),  $t$ , is:

$$t' = t - \frac{|\mathbf{r} - \mathbf{r}'|}{c}, \quad (15.23)$$

where  $t'$  is called the retarded time. This is the time at which the signal left and  $t$  is the time at which the signal arrives. The difference is due to the constant velocity,  $c$ , that the signal travels at between  $\mathbf{r}'$  and  $\mathbf{r}$ . For time-dependent charge and current distributions we must include the retardation effects in Eq. (15.22) which yields:

$$\phi(\mathbf{r}, t) = \frac{1}{4\pi\epsilon_0} \int \frac{\rho(\mathbf{r}', t')}{|\mathbf{r} - \mathbf{r}'|} dV' \quad \text{and} \quad \mathbf{A}(\mathbf{r}, t) = \frac{\mu_0}{4\pi} \int \frac{\mathbf{J}(\mathbf{r}', t')}{|\mathbf{r} - \mathbf{r}'|} dV'. \quad (15.24)$$

provided one uses the retarded time for  $t'$ .

Using  $\mathbf{E} = -\nabla\phi - \frac{\partial\mathbf{A}}{\partial t}$  and  $\mathbf{B} = \nabla \times \mathbf{A}$  one can, in principle, calculate the electric and magnetic fields for any arbitrary charge and current distribution. These explicit calculations were first worked out by Jefimenko in 1966 and yield:

$$\mathbf{E}(\mathbf{r}, t) = \frac{1}{4\pi\epsilon_0} \int \left[ \frac{(\mathbf{r} - \mathbf{r}')\rho(\mathbf{r}', t')}{|\mathbf{r} - \mathbf{r}'|^3} + \frac{(\mathbf{r} - \mathbf{r}')\dot{\rho}(\mathbf{r}', t')}{c|\mathbf{r} - \mathbf{r}'|^2} - \frac{\dot{\mathbf{J}}(\mathbf{r}', t')}{c^2|\mathbf{r} - \mathbf{r}'|} \right] dV', \quad (15.25)$$

and

$$\mathbf{B}(\mathbf{r}, t) = \frac{\mu_0}{4\pi} \int \left[ \frac{\mathbf{J}(\mathbf{r}', t') \times (\mathbf{r} - \mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|^3} + \frac{\dot{\mathbf{J}}(\mathbf{r}', t') \times (\mathbf{r} - \mathbf{r}')}{c|\mathbf{r} - \mathbf{r}'|^2} \right] dV', \quad (15.26)$$

which reduce to the usual expressions for the electrostatic and magnetostatic fields when  $\dot{\rho} = \dot{\mathbf{J}} = 0$ .

Radiation due to a point charge,  $q$ , is most directly determined by using the retarded potentials of Eq. (15.24) to yield

$$\phi(\mathbf{r}, t) = \frac{1}{4\pi\epsilon_0} \left[ \frac{qc}{|\mathbf{r} - \mathbf{r}'|c - (\mathbf{r} - \mathbf{r}') \cdot \mathbf{v}} \right], \quad (15.27)$$

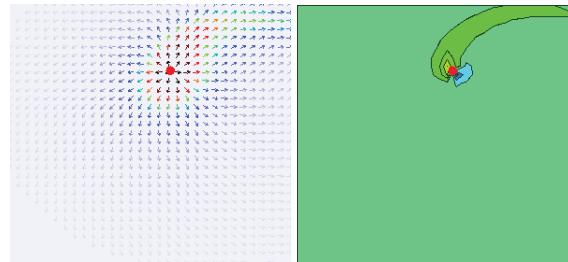
and

$$\mathbf{A}(\mathbf{r}, t) = \frac{\mathbf{v}}{c^2} \phi(\mathbf{r}, t), \quad (15.28)$$

## 15.6 Electromagnetism: Radiation

329

which are the Liénard-Wiechert potentials for moving point charges. Examples of what the electric field vectors and the magnetic field lines in the  $xy$  plane that result from moving point charges can be seen in Figures 15.17-15.19.



**FIGURE 15.19** Electromagnetic radiation examples from the `osp_demo.jar` file which show the electric field vectors and magnetic field lines (in the  $xy$  plane) of a positively charged particle in circular motion.

If we consider stationary charges and steady currents (charges moving at a constant velocity) we find that they cause static fields. We may expect from Eqs. (15.25) and (15.26) that accelerating charges and time-varying currents will create changing electric and magnetic fields, and therefore the possibility of electromagnetic radiation.

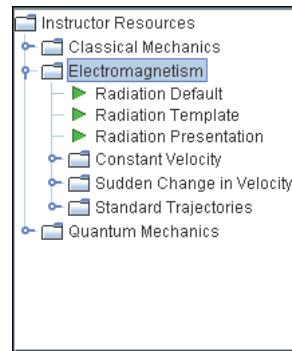
To determine whether or not we will have radiation from a particular moving charge, we look at the Poynting vector,  $\mathbf{S} = \frac{1}{\mu_0}(\mathbf{E} \times \mathbf{B})$ . When we integrate the Poynting vector over a surface, we get the power delivered through that area:

$$P(r) = \oint \mathbf{S} \cdot \mathbf{n} da . \quad (15.29)$$

In order to have radiation, there must be some power delivered to infinity, and therefore Eq. (15.29) must not vanish as  $r \rightarrow \infty$ . Since the surface area goes like  $r^2$ , for radiating systems the Poynting vector must go like  $\frac{1}{r^2}$  otherwise the power would vanish at large  $r$ . The electric and magnetic fields, therefore, must each fall off like  $\frac{1}{r}$  to have radiation. Since static electric and magnetic fields go like  $\frac{1}{r^2}$  at best, the Poynting vector goes like  $\frac{1}{r^3}$ , and hence point charges associated with these kinds of fields do not radiate. To study radiation, we look for electric and magnetic fields that go like  $\frac{1}{r}$ , which we have seen from Eqs. (15.25) and (15.26).

Qualitatively, one can think about the electric and magnetic fields that are created by point charges in the following way: there are the fields that stay *attached* to the point charge and there are fields that are *thrown off* by the point charge. The fields that are thrown off and make it to infinity, are characterized as radiation. Look at Figures 15.17-15.19 again, which ones show radiation?

The electromagnetism materials can be found use the following OSP program in the “Davidson package.”



**FIGURE 15.20** The *Explorer* pane of the *Launcher* from the `osp_demo.jar` showing the electromagnetism curricular materials.

`electrodynamics.RadiationApp` which shows a positively charged particle's electric field vectors and magnetic field lines (in the  $xy$  plane) calculated from the Liénard-Wiechert potentials of Eqs. (15.27) and (15.28). The trajectory of the particle ( $f[x] = x(t)$ ,  $f[y] = y(t)$ ) can be set, where  $c = 1$ .

The curricular material is organized into three units.

**Constant Velocity** shows variations on a positively charged particle moving with constant velocity. The electric field vectors and magnetic field lines in the  $xy$  plane are shown for a particle moving at constant velocities varying from  $v = 0$  to  $v = 0.99c$ .

**Sudden Change in Velocity** shows the effect of a sudden change in velocity on the electric field vectors and magnetic field lines in the  $xy$  plane associated with a positively charged particle.

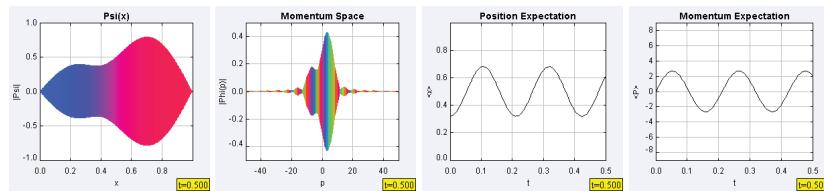
**Standard Trajectories** shows the five standard trajectories of a moving point charge that give rise to radiation. *Bremsstrahlung* shows a moving charge experiencing “braking,” a constant deceleration, giving rise to radiation. *SHO x* and *SHO y* show a particle moving in simple harmonic motion in either the  $x$  or  $y$  direction. There are two examples of constant velocity motion of the particle in one direction, and a constant acceleration motion in the other direction. *Wiggler x* and *Wiggler y* show a particle moving in simple harmonic motion in either the  $x$  or  $y$  direction and a constant velocity in the other direction. A particle following these trajectories (subjected to alternating magnetic fields) is called a wiggler and such high-intensity radiation from a wiggler can be harnessed in a free electron laser. Finally, there is *Synchrotron* radiation in which the positively charged particle is forced to move in circular motion.

### 15.7 ■ QUANTUM MECHANICS: SUPERPOSITIONS

Time-dependent quantum-mechanical wave functions are inherently complex (having real and imaginary components) due to the time evolution governed by the Schrödinger equation, which in one-dimensional position space is

$$\left[ -\frac{\hbar^2}{2m} \frac{\partial^2}{\partial x^2} + V(x) \right] \psi(x, t) = i\hbar \frac{\partial}{\partial t} \psi(x, t). \quad (15.30)$$

The standard way to visualize the wave functions that solve Eq. (15.30) is to either consider just the real part or consider the probability density, approaches that discard all phase information.



**FIGURE 15.21** Quantum mechanics examples from the `osp_demo.jar` file which show the time evolution of the equal-mix two-state ( $n_1 = 1, n_2 = 2$ ) superposition in the infinite square well by displaying the wave function in position and momentum space (shown in color-as-phase representation) and the expectation values of  $\hat{x}$  and  $\hat{p}$  versus time.

As an example, consider the standard problem of a particle of mass  $m$  confined to a region of length  $L$  by an infinite square well potential defined by

$$V(x) = \begin{cases} 0 & \text{for } 0 < x < L \\ \infty & \text{otherwise} \end{cases} \quad (15.31)$$

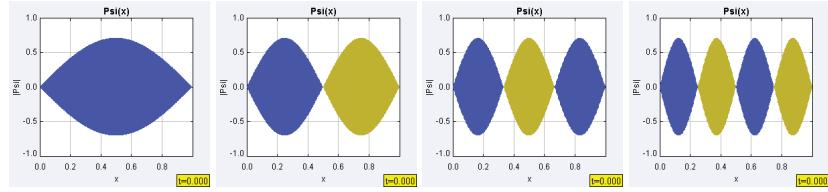
which has the position-space energy eigenstates (which are non-zero only within the well):

$$\psi_n(x) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right), \quad (15.32)$$

with energy eigenvalues,  $E_n = \frac{\hbar^2 \pi^2 n^2}{2mL^2}$ . The first four position-space energy eigenstates are shown in Figure 15.22.

The momentum-space energy eigenstates can be determined from the position-space energy eigenstates via the standard Fourier transform,

$$\phi(p) = \frac{1}{\sqrt{2\pi\hbar}} \int_{-\infty}^{+\infty} \psi(x) e^{-ipx/\hbar} dx, \quad (15.33)$$

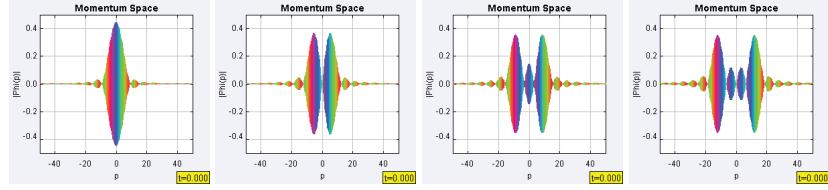


**FIGURE 15.22** Quantum mechanics examples from the `osp_demo.jar` file which show the first four energy eigenstates in position space (shown in color-as-phase representation) of the infinite square well.

which yields

$$\phi_n(p) = (-i)\sqrt{\frac{L}{\pi\hbar}}e^{-ipL/2\hbar} \left[ e^{+in\pi/2} \frac{\sin[(pL/\hbar - n\pi)/2]}{(pL/\hbar - n\pi)} - e^{-in\pi/2} \frac{\sin[(pL/\hbar + n\pi)/2]}{(pL/\hbar + n\pi)} \right]. \quad (15.34)$$

The first four momentum-space energy eigenstates are shown in Figure 15.23.



**FIGURE 15.23** Quantum mechanics examples from the `osp_demo.jar` file which shows the first four energy eigenstates, in momentum space, of the infinite square well.

In general, the time evolution of wave functions can be written as

$$\psi(x, t) = e^{-i\hat{\mathcal{H}}t/\hbar}\psi(x, 0), \quad (15.35)$$

where  $\hat{\mathcal{H}}$  is the Hamiltonian of the system. For the position- and momentum-space energy eigenstates of the infinite square well, the time dependence of these states are

$$\psi_n(x, t) = e^{-iE_n t/\hbar}\psi_n(x) \quad \text{and} \quad \phi_n(p, t) = e^{-iE_n t/\hbar}\phi_n(p), \quad (15.36)$$

where  $\psi_n(x)$  and  $\phi_n(p)$  are given in Eq (15.32) and Eq (15.34), respectively.

We can consider states with non-trivial time evolution by considering equal-mix two-state superpositions, which is one of the simplest examples of non-trivial

## 15.7 Quantum Mechanics: Superpositions

333

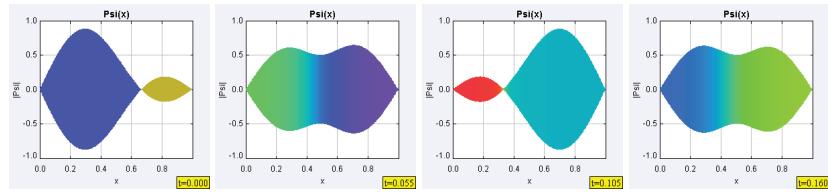
time-dependent states.<sup>3</sup> These position- and momentum-space wave functions are just

$$\Psi_{n_1 n_2}(x, t) = \frac{1}{\sqrt{2}} [\psi_{n_1}(x, t) + \psi_{n_2}(x, t)] , \quad (15.37)$$

and

$$\Phi_{n_1 n_2}(p, t) = \frac{1}{\sqrt{2}} [\phi_{n_1}(p, t) + \phi_{n_2}(p, t)] . \quad (15.38)$$

The first equal-mix two-state superposition ( $n_1 = 1, n_2 = 2$ ) of the infinite square well is shown in position space at four different times in Figure 15.24.



**FIGURE 15.24** Quantum mechanics examples from the `osp_demo.jar` file which shows the first equal-mix two-state superposition ( $n_1 = 1, n_2 = 2$ ) of the infinite square well at four different times.

We can examine the time dependence of more general states by adopting a general description for a time-dependent superposition:

$$\Psi(x, t) = \sum_{n=1}^{\infty} c_n e^{-iE_n t/\hbar} \psi_n(x) , \quad (15.39)$$

where the expansion coefficients satisfy  $\sum_n |c_n|^2 = 1$ . As an example, consider an initially localized state, chosen to be of Gaussian shape, and of the form

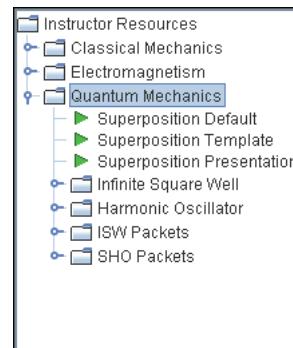
$$\Psi_G(x, 0) = \frac{1}{\sqrt{b\sqrt{\pi}}} e^{-(x-x_0)^2/2b^2} e^{ip_0(x-x_0)/\hbar} . \quad (15.40)$$

The expansion coefficients,  $c_n$ , for this superposition are determined by the integral

$$c_n = \int_0^L [\psi_n^*(x)] [\Psi_G(x, 0)] dx . \quad (15.41)$$

<sup>3</sup>One of the earliest pedagogical visualizations of the time dependence of such a two-state system is by C. Dean, “Simple Schrödinger Wave Functions Which Simulate Classical Radiating Systems,” *Am. J. Phys.* **27**, 161–163 (1959).

Such a time-dependent state in the infinite square well experiences so-called quantum-mechanical revivals, in which an initially localized wave packet reforms a definite time later. The quantum-mechanical revivals for this system are exact and any wave packet returns to its initial state after a time  $T_{\text{rev}}$ . At half this time,  $t = T_{\text{rev}}/2$ , the wave packet also reforms (same shape, width, etc.), but at a location mirrored about the center of the well with ‘mirror’ (opposite) momentum as well. At various fractional multiples of the revival time,  $pT_{\text{rev}}/q$ , the wave packet can also reform as several small copies (sometimes called ‘mini-packets’ or ‘clones’) of the original wave packet.



**FIGURE 15.25** The *Explorer* pane of the *Launcher* from the `osp_demo.jar` showing the quantum mechanics curricular materials.

The curricular material uses the `QMSuperpositionApp` to show wave functions in color-as-phase representation ( $\hbar = 2m = 1$ ). In color-as-phase representation the amplitude of the wave function is depicted as the distance from the bottom to the top of the wave function at a given point and time. The phase is depicted as the color of the wave function with a map between phase angle and color. In quantum-mechanical time evolution, the phase evolves in time *clockwise* in the complex plane.

The position-space energy eigenstates and their energies are calculated either numerically for any user-defined potential energy function,  $V(x)$ , or calculated analytically for the special cases of the infinite square well, the periodic infinite well, and the simple harmonic oscillator.

The power of `QMSuperpositionApp` is that it allows the author or user to specify a set of expansion coefficients for Eq. (15.39) in a comma-delimited list. The program then calculates the sum over states, and then automatically evolves the superposition state in time. The use of analytic solutions for each eigenstate allows the simulation to run for a long period of time and yet never accumulate numerical error since the wave function is calculated anew at each time step according to an analytic formula.

Depending on the analysis one wishes to perform, one of the following programs based on `QMSuperpositionApp` (which itself only shows the position-space wave function) in the “qm” sub-package within the “Davidson package” can be chosen:

`QMSuperpositionProbabilityApp` adds a view of the position-space probability density.

`QMSuperpositionExpectationXApp` adds a view of the expectation value of  $\hat{x}$ ,  $\langle \hat{x} \rangle$ .

`QMSuperpositionExpectationPApp` adds a view of the expectation value of  $\hat{p}$ ,  $\langle \hat{p} \rangle$ .

`QMSuperpositionCarpetApp` adds a view of the position-space quantum carpet.

`QMSuperpositionMomentumCarpetApp` which adds a view of the momentum-space carpet.

`QMSuperpositionFFTApp` which adds a view of the momentum-space wave function.

The curricular material is organized into three units.

**Infinite Square Well** shows the first five energy eigenstates and six two-state superpositions for a particle in an infinite square well. The eigenstates are shown in position and momentum space. The two-state superpositions are time dependent and are shown to evolve in time. The superpositions are shown in position and momentum space. In addition, one can choose to look at the expectation value of  $\hat{x}$  and  $\hat{p}$ .

**Harmonic Oscillator** shows the first five energy eigenstates and six two-state superpositions for a particle in a harmonic oscillator. The eigenstates are shown in position and momentum space. The two-state superpositions are time dependent and are shown to evolve in time. The superpositions are shown in position and momentum space. In addition, one can choose to look at the expectation value of  $\hat{x}$  and  $\hat{p}$ .

**Infinite Square Well Packets** shows four initially localized Gaussian wave packets in an infinite square well. The energy scale, which determines the time scale for the animation, is set  $2/\pi$  which forces  $T_{\text{rev}} = 1$ . This is a convenient time scale for the study of revivals and fractional revivals which occur at  $T_{\text{rev}}$  and fractions of  $T_{\text{rev}}$ , respectively. The time evolution of these states are shown in position and momentum space. In addition, one can choose to look at the expectation value of  $\hat{x}$  and  $\hat{p}$  and also the position-space quantum carpet.

**Harmonic Oscillator Packets** shows six different initially localized Gaussian wave packets (varying initial widths and momenta) in three different harmonic oscillator wells ( $\omega = 0.5, 1, 2$ ).

## CHAPTER

## 16

## Tracker

©2005 by Doug Brown, June 28 2005

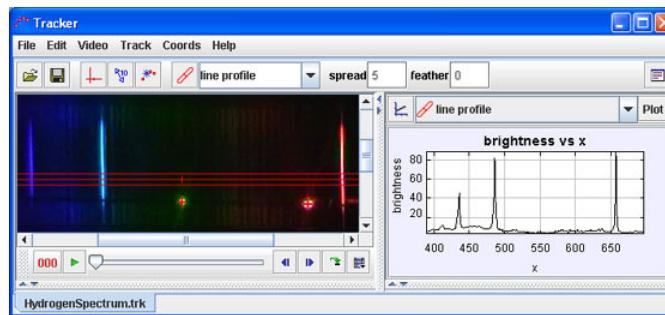
Tracker is a video analysis application built using the core OSP library and the optional media framework defined in the `org.opensourcephysics.media` package. Tracker source code is available in the `org.opensourcephysics.-cabrillo.tracker` package.

### 16.1 ■ OVERVIEW

One of the goals of the Open Source Physics project is to develop a Java library that can be used both to teach computational physics and to develop robust high-quality educational software. The Tracker video analysis application described here and the EJS modeling application described in Chapter 17 are examples of large programs that illustrate and achieve this design goal.

The development of these programs has enabled the OSP project to improve the core library in many ways. The API has been improved and made more consistent, bugs have been found and squashed, and we have learned what tools are useful to the developer community. But most importantly, the development of Tracker and EJS has resulted in great tools for the physics education community.

### 16.2 ■ VIDEO ANALYSIS



**FIGURE 16.1** Tracker analysis of a Hydrogen spectrum.

The value of video analysis in physics education is well established. The video format is familiar to students, contains a wealth of spatial and temporal data (built-in clock), and provides a bridge between direct observations and abstract representations of many physical phenomena. Tracking the positions and colors of features in a video image, and then transforming the resulting data into real-world values and graphical overlays, offers many possibilities for building and testing physical models both conceptually and analytically.

Moreover, there are effective and interactive educational video applications that require no feature tracking at all. For example, special effects filters can generate live motion diagrams of moving objects, and theoretical model animations can be overlaid directly on videos of interference patterns. In addition, saving an animation as a video is a useful way to archive, transmit or display it easily in another application. Often, recording a video of a processor-intensive animation is the most practical way to display it at a reasonable speed.

Combining video analysis with model animations illustrates many features of the experimental method: the videos provide opportunities for objective observation, measurement, and model building, and the model animations allow theoretical predictions and comparisons with observed behavior. Tracker has been developed to facilitate this combination by extending the OSP drawing, control and tools frameworks and thus seamlessly integrating the powerful OSP animation library into video analysis.

As its name implies, Tracker analyzes videos by tracking features of interest in the video image. The examples discussed below illustrate the types of tracks and analysis that are possible. All examples can be found on Tracker's web start page at

<http://www.cabrillo.edu/~dbrown/tracker/webstart/>.

## 16.3 ■ EXAMPLES

### Ball Toss

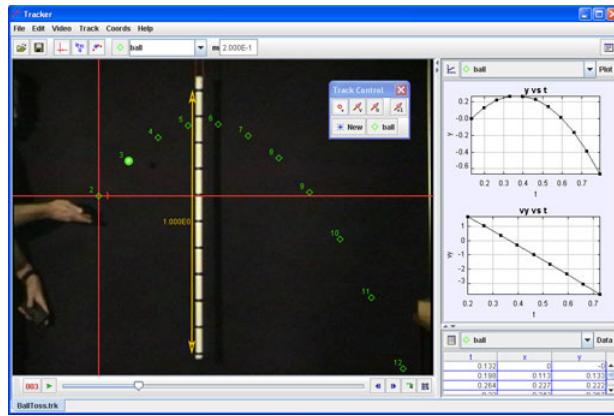
Figure 16.2 shows a tossed ball that has been tracked as a *Point Mass*. The track consists of a series of steps (green diamonds), each corresponding to the ball's position at a single time. In this example, all of the steps are visible simultaneously since "trails" have been turned on. The step associated with the current video frame is highlighted with a bold circle.

A *Coordinate System*, defined by an origin, angle and scale, transforms the ball's raw image positions into world coordinates. The red *Axes* shows the current origin and angle and the yellow *Tape Measure* shows the current scale. A *Plot* view and *Table* view of the ball's world data are displayed on the right.

In Figure 16.3 the Ball Toss video has been magnified and the ball's velocity and acceleration vectors have been displayed with buttons on the *Track Control*. The vectors can be (a) dragged with the mouse for side-by-side comparison or tip-to-tail addition, and (b) multiplied by the mass to display momentum and net

## 16.3 Examples

339

**FIGURE 16.2** Ball Toss.

force.

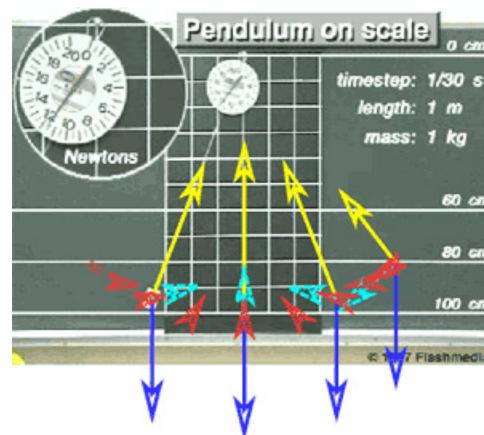
There is no limit to how many point masses can be tracked. In addition, a Center of Mass can track a system of masses such as colliding or exploding objects.

**FIGURE 16.3** Velocity and acceleration vectors.**Pendulum**

Figure 16.4 uses blue and yellow *Vectors* to track the weight and tension forces acting on a simple pendulum and a cyan *Vector Sum* that shows the time-dependent net force. For comparison, the motion of the pendulum has also been tracked and its acceleration displayed as red arrows. This example shows how Tracker enables students to extend force diagrams beyond static images and to verify their validity visually, thus reinforcing the conceptual net force-acceleration connection. The numerical data associated with these tracks is, of course, also available for analysis.

**Hydrogen Spectrum**

Figure 16.1 uses a *Line Profile* track to measure the brightness of the pixels in a video image of a Hydrogen spectrum. The pixel positions are converted to wavelength and the resulting spectral intensity data are plotted on the right. A *Calibration*

**FIGURE 16.4** Pendulum forces and acceleration.

*bration Points* track uses red and green laser spots in the spectrum to easily and accurately calibrate the wavelength scale. Students can interactively explore the spectrum in detail by dragging the line profile with the mouse.

### Motion Diagram

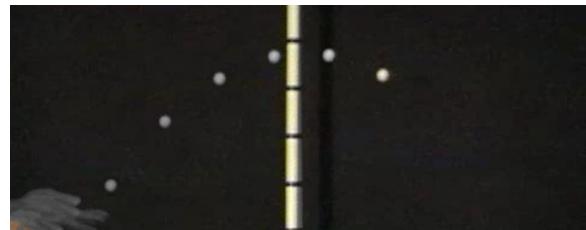
**FIGURE 16.5** Live motion diagram.

Figure 16.5 illustrates how Tracker can create a live motion diagram simply by applying a “ghost” *Filter* to a video. There are no tracks at all here! Live motion diagrams can be powerful conceptual teaching tools, particularly in interactive lectures.

## 16.4 ■ INSTALLING AND USING TRACKER

Tracker, like all Open Source Physics programs, requires Java 1.42 or later. In addition, you must install QuickTime.<sup>1</sup> On Windows we recommend QuickTime 7.0 or later (after you install Java) because it automatically installs QuickTime for Java. QuickTime 6.4 or 6.5 can be used if you select custom and install all options. The latest version of QuickTime may be downloaded from

[http://www.apple.com/quicktime/download/.](http://www.apple.com/quicktime/download/)

You do not need to purchase QuickTime Pro to use Tracker. For more help refer to the “Installation” help topic on the Tracker website.

[http://www.cabrillo.edu/~dbrown/tracker/.](http://www.cabrillo.edu/~dbrown/tracker/)

Tracker itself may be launched directly from the web using Java *WebStart* or downloaded to a desktop. If you download, double-click the Tracker.jar file to launch Tracker.

Tracker-based curricular material may be organized using the OSP Launcher. Launch nodes are associated with the Tracker program, Tracker xml data, and an html page. Download and run the TrackerHelp.jar file to see how this is implemented. See Saving Tracker Documents on the Tracker website for information on how to create Tracker xml files.

Once Tracker is launched, a typical analysis session follows the steps listed below. Each of these is described in detail in its corresponding chapter section.

1. A video file is opened or imported, video clip properties are set and video filters are applied if desired.
2. Coordinate system properties are set using the axes, tape measure and/or one or more calibration points.
3. Video features are tracked by creating appropriate tracks and marking the features on each frame of interest.
4. Track data are viewed and analyzed in a plot or table view. Data may be copied to the clipboard for pasting into other applications.
5. The entire document (video, coordinate system and tracks) is saved in a Tracker xml data file.

## 16.5 ■ USER INTERFACE

Every Tracker tab, like the “Untitled” tab shown in Figure 16.6, displays a Tracker document with the following components:

<sup>1</sup>Tracker can open jpeg images and animated gifs and these file formats do not require QuickTime.



**FIGURE 16.6** Tracker user interface.

- *Menu Bar* with menus that control most program commands and settings.
- *Toolbar* that offers quick access to frequently used commands, controls and track data.
- Main *Video View* that displays the video and tracks.
- *Video Player* that controls the video and tracks and provides access to video clip settings.
- *Split Panes* that provide access to additional views.

### Menus

The *File*, *Edit* and *Help* menus are shown in Figure 16.7 and described below. The *Video*, *Track*, and *Coords* menus are discussed in the corresponding chapter sections. The Window menu is not shown.

#### File menu:

- *New* creates a new untitled tab with a blank Tracker document.
- *Open* opens a video or Tracker xml file in a new tab.
- *Import* imports a video or selected elements of an xml file into the current tab.
- *Export* saves selected elements of the current tab in an xml file.
- *Save* saves the entire current tab in an xml file.
- *Record* records the current video clip and tracks as a QuickTime movie or animated gif.

## 16.5 User Interface

343

**FIGURE 16.7** Tracker Menus.

Edit menu:

- *Copy* copies the selected track to the clipboard. If no track is selected, the entire tab is copied.
- *Clear* deletes all tracks in the current tab.
- *Mat Size* sets the minimum size of the drawing area. By default, the mat size is that of the video.
- *Preferences* displays the preferences dialog as described in Configuring Tracker.
- *Properties* displays the current tab in xml format (i.e., exactly as it would be saved).

Help menu:

- *Message Log* displays a message log useful for trouble-shooting.
- *About QuickTime* displays the QuickTime version and verifies correct QuickTime operation.
- *About Java* displays the version of the Java VM being used.

**Toolbar**

The toolbar always includes the components shown in Figure 16.8 and listed below. When a track is selected, additional data fields may be displayed.

Toolbar components shown from left to right in Figure 16.8:

1. *Open* button opens a video or Tracker xml file in a new tab.

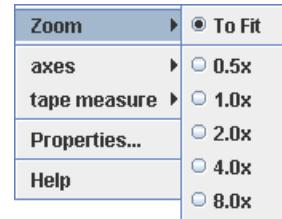
**FIGURE 16.8** Tracker toolbar.

2. *Save* button saves the current tab in an xml file.
3. *Axes* button shows and hides the axes.
4. *Tape Measure* button shows and hides the tape measure.
5. *Track Control* button shows and hides the track control.
6. *Footprint* button (shown disabled) sets the color and footprint of the selected track.
7. *Selected Track* dropdown selects tracks and displays/edits the name of the selected track.
8. *Description* button shows and hides track and document descriptions.

### Video View Popup

The video view has a fixed, stable video image. All tracks, and the axes and tape measure, are marked and/or edited in this view.

To facilitate accurate marking, a popup menu (see Figure 16.9) allows you to Zoom (magnify) the video image up to 8x. The popup is displayed by right-clicking the mouse (control-click on Mac); the zoomed image is then centered on the click point. The popup menu also provides quick access to track menus and Tracker help.

**FIGURE 16.9** Video popup menu.

### Video Player

You can drag the video player by the left end to convert it to a floating window as shown in Figure 16.10. Closing the floating window restores the player to its normal position.

Video player components (from left to right):

1. *Readout* displays the current video time, frame or step.



**FIGURE 16.10** Video player control.

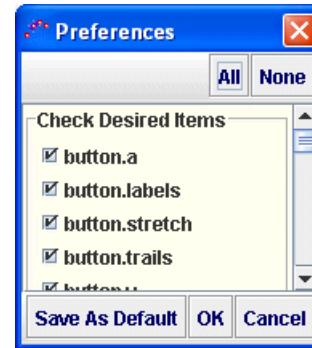
2. *Play/Pause* button plays, pauses and resets the video.
3. *Slider* enables you to move quickly to a desired frame.
4. *Step Forward* and *Step Back* buttons step the video one frame at a time.
5. *Loop* button toggles looping (continuous play).
6. *Clip Settings* button displays the Clip Inspector (see Clip Inspector).

### Split Panes

Split panes allow you to open additional views (see Section 16.10) by moving or clicking a thin divider. By default, the right divider opens a Plot View and the bottom divider opens a World View. Each of these also contains a split pane for a total of four views. Views can also be opened by selecting them in the *Window* menu.

### Configuration

Tracker's user interface can be greatly simplified by hiding unwanted features using the Preferences dialog shown in Figure 16.11. This is particularly important when introducing students to Tracker for the first time. As they gain familiarity with the program, additional features can be included as needed.



**FIGURE 16.11** Configuration preferences dialog box.

To display the preferences dialog, choose the *Edit|Preferences* menu item. Selected features will be enabled; unselected ones will be hidden.

There are two ways to save your preferences:

1. Click the Save As Default button. This will immediately save the currently checked items in a default configuration file named "default\_config.trk". New Tracker documents automatically load this file to configure themselves. (Note: the Save As Default button is only available when the config.save item is selected in the preferences. If it is hidden and you wish to enable it, select the config.save item and click the OK button, then reopen the preferences dialog.)
2. Select the config.saveWithData item in the preferences. The preferences will then be included in the xml file whenever the Tracker document is saved. Opening or importing this file will then restore the saved preferences.

## 16.6 ■ VIDEOS

Tracker can open QuickTime movies, animated gif files, jpeg and gif images, and image sequences. A Tracker document can have only one video open at a time.

Videos are characterized by their pixel dimensions, frame count and time intervals between frames (note that there is no requirement for a constant time interval between frames). All videos are displayed in 24-bit RGB format regardless of their original color depth.

It is not necessary to open a video to use Tracker. When there is no video, or when the video is hidden, tracks are drawn on a white background.

### Video Clips

A Video Clip defines a set of equally spaced steps in a video using three parameters: a start frame, a step size and a step count. The start frame is the frame number of the first step, the step size is the frame increment between successive steps, and the step count is the number of steps in the clip. For example, a clip with start frame 3, step size 2 and step count 5 would have step numbers 0, 1, 2, 3 and 4 that map to video frame numbers 3, 5, 7, 9 and 11, respectively. Newly opened videos have default start frame 0, step size 1 and step count equal to the video frame count.

Note: point mass tracks work best when their steps are marked on every frame in the clip, so it is useful to set the clip properties before marking. Clip settings may be changed at any time, but previously unmarked frames will then require marking.

A clip is defined for every video and even for no video. For single-frame and null videos the clip settings apply to tracks but every step maps to the same video image.

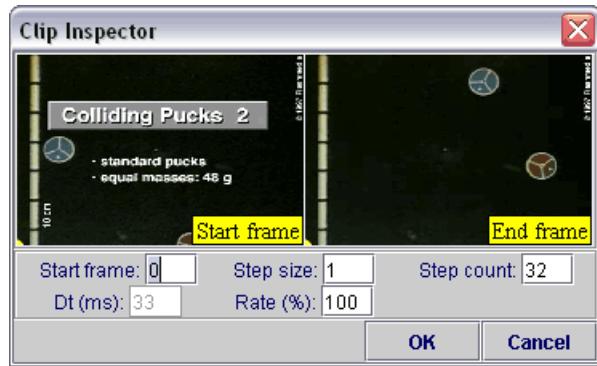
### Clip Inspector

The Clip Inspector (see Figure 16.12) shows the current video clip settings along with thumbnail images of the start and end frames. In addition, there are fields for setting the mean time  $t$  between video frames (important for high-speed or time-

## 16.6 Videos

347

lapse videos) and the play rate as a percent of normal playback speed. To display the clip inspector, click the Clip Settings button on the player.



**FIGURE 16.12** Video clip inspector.

### Playing Videos

The player plays only those frames that are video clip steps. By default, all steps are played with no frames dropped even though this often results in slower than normal play. If this feature is turned off a video will attempt to play at the rate specified in the clip inspector but may drop frames.

The readout can display time in seconds (measured from the clip's start frame), step number, or video frame number. To select a readout type, click the readout and choose from the dropdown list.

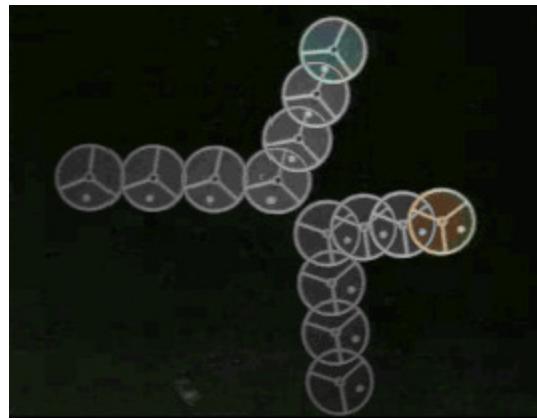
### Video Filters

It is often useful to process a video image in order to deinterlace, suppress noise, or enhance features of interest. Tracker provides such image processing through the use of video *Filters*. Multiple filters may be applied to a video, in which case the filtered image from each becomes the source image for the next.

Most filters apply to each video image independently, but some, like “ghost”, combine multiple images and are apparent only when stepping or playing the video as shown in Figure 16.13.

Tracker currently provides the following filters:

- Baseline (subtracts a background image from each video frame)
- Brightness/contrast
- Deinterlace (selects odd or even fields of interlaced video frames)
- Ghost (leaves trails of bright objects against a dark background)

**FIGURE 16.13** Ghost filter.

- Dark ghost (leaves trails of dark objects against a bright background)
- Gray scale
- Negative
- Sum (adds or averages video frames)
- Threshold (turns pixels above the threshold brightness white, below threshold black)

Most filters have settable properties that are accessible through their properties dialog. The properties dialog is displayed when a filter is first created and when choosing a filter's Properties menu item.

### **Video Menu**

The Video menu (see Figure 16.14) allows a user to Import a video into the current tab, Close an existing video, toggle the Visible and Play All Steps properties, and create and configure filters. Multiple filters are listed and applied in the order in which they are created. Existing filters can be temporarily Disabled or permanently Deleted. Choose Video | Filters | Clear to delete all filters.

### **Recording Videos**

Tracker can record the current video clip with filters and visible track overlays as a QuickTime movie or animated gif. The movie or gif will have the dimensions of the current viewport in the main video view. Note: QuickTime movie file sizes can be large since they are not compressed. Animated gif files are always compressed, but have only 256 colors.

## 16.7 Coordinate System

349

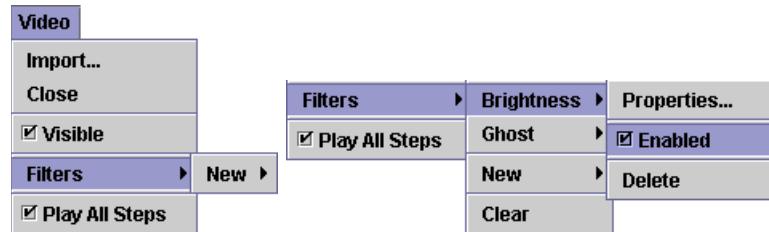


FIGURE 16.14 Video Menus.

To record a video, choose the desired format from the File|Record menu shown in Figure 16.15. Assign a file name to the new movie or gif and click the Record button in the chooser dialog.



FIGURE 16.15 Video Menus.

Tracker will reset the video to the first frame in the clip and ask if you wish to include it in the video. Click Add to record the frame and step forward, Skip to skip it and step forward, or End to end the recording without adding the current frame.

## 16.7 ■ COORDINATE SYSTEM

Marking a point in a video image uniquely defines its image position. But since a video image is a camera view of the real world, a marked point also has a world position relative to some laboratory reference frame. Transforming image positions into world coordinates is the job of the Coordinate System.

A point's image position is its pixel position on the video image, measured from the top left corner. The positive x-axis points to the right and the positive y-axis points down. Image units are like pixels except that they are doubles, not

integers. This means that the center of the top left pixel has coordinates (0.5, 0.5); the image position (0.0, 0.0) is the top left corner of the top left pixel! The lower right corner of a 640x480 pixel image is at (640.0, 480.0).

The world position is the scaled position of the point relative to a specified world reference frame. The reference frame origin may be anywhere on or off the image and the positive x-axis may point in any direction. The positive y-axis is 90 degrees counterclockwise from the positive x-axis.

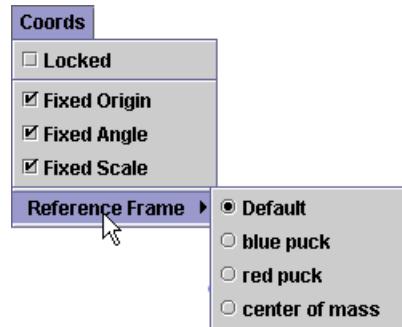
The coordinate system encapsulates the relationship between image and world coordinates by defining for each frame of the video:

1. *Origin*: the image coordinates of the world origin.
2. *Angle*: the counterclockwise angle in radians from the image x-axis to the world x-axis.
3. *Scale*: the world scale in image units (pixels) per world unit.

With these definitions, the coordinate system constructs a set of AffineTransforms, powerful Java objects for manipulating points and shapes in 2D space. Locking the coordinate system prevents any changes to the transforms.

### Coords Menu

The Coords menu (see Figure 16.16) has items for Locking the coordinate system, setting the Fixed scale, origin and angle properties of the default coordinate system, and choosing a Reference Frame. A fixed coordinate system property is the same in every video frame (i.e., the camera view does not change), so setting it in one frame sets it in all. For videos that pan, tilt or zoom, un-selecting an item allows (and requires) the corresponding property to be set independently for every frame in the clip.



**FIGURE 16.16** Coords menu has items to control the coordinate system.

The Reference Frame submenu enables you to select reference frames in which the origin moves along with a point mass or center of mass track (angle and scale

## 16.7 Coordinate System

**351**

continue to be those of the default coordinate system). Center of mass reference frames are particularly useful when studying collisions.

**Axes**

**FIGURE 16.17** The origin and angle of the axes can be set using the mouse or using a text field.

The Axes provide a convenient way to display and set the origin and angle of the coordinate system. The positive x-axis is indicated by a tick mark near the origin. To display the axes, click the axes button on the toolbar.

To set the origin, select and drag or nudge it to the desired location in the main video view as shown in Figure 16.17. To set the angle, click anywhere on the positive x-axis and drag or nudge it to rotate the axes about the origin. Hold down the shift key to restrict angles to 5 degree increments. The angle is displayed in an angle field on the toolbar. A desired angle may be entered directly in this field.

**Tape Measure**

**FIGURE 16.18** The tape measure.

The Tape Measure shown in Figure 16.18 has a readout that displays the distance between its ends in world units. To display the tape, click the tape button on the toolbar. Drag the shaft of the tape to move it or either end to change its length or orientation.

To set the scale (calibrate the video) using the tape measure, first set the ends of the tape at positions that are a known distance apart. Then double-click the tape readout and enter the known distance.

**Offset Origin**

An Offset Origin provides a way to set the position of an origin that is not on the video image. The offset origin has a fixed offset in world units relative to the origin so that when it is dragged the origin moves with it.



**FIGURE 16.19** The coordinate axes origin can be offset by dragging or using a text box.

To create an offset origin, choose the Track | New | Offset Origin menu item and click on a feature in the video image that has known world coordinates. The offset origin is initially assigned the current world coordinates of the clicked image feature, and the x- and y-components of the offset are displayed on the toolbar. To change the offset, enter the desired values in the toolbar fields. To move an offset origin, select and drag or nudge it. (See Figure 16.19.)

Note that changing the offset value moves the coordinate system origin so that the position of the offset origin remains unchanged. Conversely, moving the offset origin moves the coordinate system origin so that the offset (world coordinates) remains unchanged.

### Calibration Points

*Calibration Points* shown in Figure 16.20 are similar to the offset origin except that there are two points with known world coordinates. When either of the calibration points is dragged, all three coordinate system properties—scale, origin and angle—change in order to maintain these world coordinates. Calibration points are the easiest way to set coordinate system properties when two features with known world coordinates are visible in all video frames.



**FIGURE 16.20** Calibration points set the video frame's scale.

To create a set of calibration points, choose the Track | New | Calibration Points menu item and click on two features in the video image that have known world coordinates. The calibration points are initially assigned the current world coordinates of the clicked image features and their x- and y-components are displayed on the toolbar.

To change the world coordinates of either point, enter the desired values in the toolbar fields. Changing either point's world coordinates changes the scale, origin and angle so that the image positions of both points remain unchanged.

To move a calibration point, select and drag or nudge it. Moving a calibration point changes the scale, origin and angle so that the world coordinates of both

## 16.8 Tracks

353

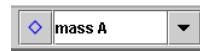
points and the image position of the unselected point remain unchanged.

Note: Calibration points are very powerful. It is strongly recommended to display the axes and tape measure, then create and "play" with some calibration points to see how they work.

### 16.8 ■ TRACKS

A *Track* represents a single video feature that evolves over time. All interactive elements in Tracker, including the axes, tape measure, offset origin and calibration points discussed above, are tracks.

The image position or shape of the feature in a single video frame is known as a Step; thus, a track is a series of steps. Each step in turn consists of an array of points that can be selected and moved with the mouse or keyboard. Some steps, like those for point mass tracks, have only a single point, but others, like vector steps, have two end points plus a center handle point.



**FIGURE 16.21** Track name, color, and footprint.

Every track is identified by its Name, Color, Footprint (visible shape) and Description. Newly created tracks are assigned default values for the first three properties that depend on the type of track. For example, a point mass might initially be called "mass A" and be drawn as a blue diamond. These are displayed as a toolbar item as shown in Figure 16.21 when the track is selected.

#### Track Control

The *Track Control* is a compact floating window for creating tracks and controlling the properties of existing tracks. To display the track control, click the track control button on the toolbar. The Track menu duplicates some of the track control functionality and provides access to axes and tape measure properties.



**FIGURE 16.22** Track control and track menu.

Track control buttons (from left to right):

1. *Trails* button shows and hides all trails.
2. *Labels* button shows and hides all labels.
3. *Positions* button shows and hides all point mass positions.
4. *Velocities* button shows and hides all point mass velocity vectors.
5. *Accelerations* button shows and hides all point mass acceleration vectors.
6. *Vectors* button shows and hides all Vector tracks.
7. *Stretch* button stretches all vectors.
8. *Dynamics* button multiplies all motion vectors by mass.

To create a new track, select the desired track type from the New button or menu list. A newly created track is automatically selected for identification and marking.

### **Marking Tracks**

Marking a track refers to the process of drawing its steps using the *Crosshair* cursor shown in Figure 16.23. For point mass and vector tracks, this means stepping through the video clip and, on each frame, clicking or dragging the crosshair with the mouse. Offset origin, calibration points and line profile tracks are marked on only a single video frame and automatically duplicated for all.



**FIGURE 16.23** Crosshair cursor.

Tracks that require marking on multiple video frames have an *Autostep* property (true by default) that for efficiency steps the video forward after each frame is marked. Obtaining the crosshair cursor depends on the value of the *Mark By Default* property (false by default): when true, the crosshair is displayed by default on any frame that is not yet marked; when false, the crosshair is displayed only when the user holds down the shift key.

Once a step is marked it can be selected by clicking any of its points. This typically displays data associated with the step in a set of editable fields on the toolbar. A selected step is edited by dragging it with the mouse, nudging it one pixel at a time with the arrow keys, or entering the desired value in a toolbar field. Very fine control is possible at a high zoom level.

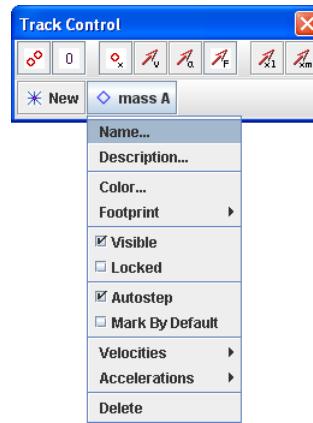
Every track has a *Locked* property (false by default). Locking a track prevents the creation, deletion or modification of its steps, though its identification properties remain editable.

### **Personal Track Menus**

Each track has its own “personal” menu that is added to both the track menu and video popup as a submenu and to the track control as a button. Most of a track’s

## 16.9 Track Types

355

**FIGURE 16.24** Menu for a point mass.

properties can be set using its personal menu as shown in Figure 16.24.

## 16.9 ■ TRACK TYPES

The track types covered in this section are Point Mass, Center of Mass, Vector, Vector Sum, and Line Profile.

### Point Mass

A Point Mass track shown in Figure 16.25 represents a mass moving as a point-like object. It is the most fundamental model of a moving inertial object. Point masses are the building blocks from which more complex and realistic models of physical systems are constructed in classical physics.

**FIGURE 16.25** Point mass steps and mass field.

A newly created point mass is given a default mass of 1.0 (arbitrary units). To change the mass, enter a new value ( $m \geq 0$ ) in the mass field on the toolbar.

### Velocity and Acceleration

The instantaneous velocities and accelerations of a point mass are estimated using numerical derivative algorithms. The default x-component algorithm for velocity

is  $v_n = (x_{n+1} - x_{n-1})/2$  and for acceleration is  $a_n = (2x_{n+2} - x_{n+1} - 2x_n - x_{n-1} + 2x_{n-2})/7$ . Note that these equations require three sequential positions to determine velocity and five to determine acceleration—if these conditions are not met, velocity and/or acceleration will not be calculated or displayed.

To display velocity or acceleration vectors, click the velocities or accelerations button on the track control. The vectors are drawn with dotted lines (Figure 16.26) and are initially attached to their positions (i.e. the tail of the velocity vector for step n is at the step n position).



**FIGURE 16.26** A velocity vector can be attached to position.

Note: Some motion vectors, especially accelerations, may be very short. You can artificially "stretch" them by a factor of 2 or 4 by clicking on the stretch button on the track control. Zooming in also increases their length.

Select a motion vector by clicking near its center to display its components, magnitude and direction on the toolbar. The fields will be grayed out since motion vectors are not editable (Figure 16.27).

t	0.10	vx	213.38	wy	316.90	v	382.04	theta	56.0
---	------	----	--------	----	--------	---	--------	-------	------

**FIGURE 16.27** Velocity data are displayed on the toolbar.

Drag a vector to detach it from its position and move it around. Drop the vector with its tail near its position to reattach—it will snap to the position. A vector will also snap and attach to the origin when the axes are visible. This is useful for estimating and visualizing its components. Attach all vectors quickly to the origin or their positions with the Tails to Origin or Tails to Position items in a track's personal menu.

Click the dynamics button on the toolbar to multiply all velocity and acceleration vectors by their mass. This changes them to momentum and net force vectors, respectively. Tracker changes the labels next to the momentum and net force vectors and draws these vectors with dashed lines to distinguish them from motion vectors.

Change the footprint of a vector by first selecting it, then clicking the footprint button on the toolbar and choosing from the list. The "big arrow" footprint (Figure 16.28) is particularly useful for large classroom presentations.

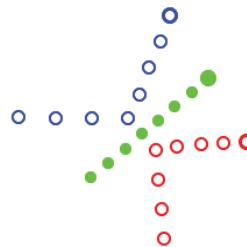
Motion vectors, like all vectors, can be linked tip-to-tail to visually determine their vector sum (see Linking Vectors).

## 16.9 Track Types

357

**FIGURE 16.28** The display properties of vectors can be set.**Center of Mass**

A *Center of Mass* (cm) track represents the center of mass of a collection of point masses, and is itself a point mass with the usual motion vectors. Its mass, however, is not settable, but instead is the sum of its point masses. Similarly, its steps are not marked but instead are determined by the positions and masses of its point masses. Figure 16.29 shows the green center of mass of a pair of colliding pucks. Center of mass footprints are always solid to distinguish them from independent point masses.

**FIGURE 16.29** The center of mass of colliding pucks footprints track.

Select the point masses to include in a cm by checking them in the Select Masses dialog. The dialog is displayed when the center of mass track is initially created or by choosing Select Masses in its personal menu.

**Vector**

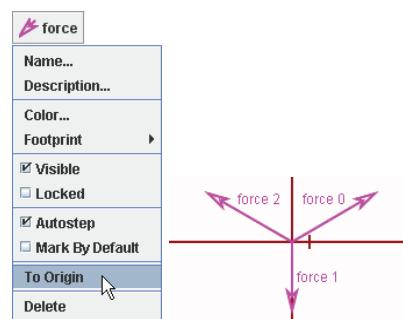
A Vector track can represent any vector but is commonly used as a force in a force diagram. For traditional static force diagrams the background video can be a still image (photo or drawing) or simply a blank screen. In Tracker, QuickTime videos present new opportunities to study force diagrams that vary with time as illustrated in the earlier Pendulum example.

To mark a vector, click the crosshair cursor at the tail and drag the tip with the pointer hand as shown in Figure 16.30. Vectors are drawn with solid lines to distinguish them from motion vectors. Select any point on a vector to display its components, magnitude and direction on the toolbar. Enter a desired value in the

**FIGURE 16.30** Marking and moving a vector.

appropriate field or select and drag/nudge a vector's tip to change the components. Drag or nudge the center of a vector to move it without changing its components.

When the axes are visible you can drop a vector with its tail near the origin and it will snap to the origin. This is useful for estimating and visualizing its components. Attach all of a vector's steps quickly to the origin with the To Origin item in its personal menu as shown in Figure 16.31.

**FIGURE 16.31** Vectors can be set to display at the origin.

### Linked Vectors

Vectors can be linked tip-to-tail to visually determine their vector sum as shown in Figure 16.32. To link vectors, drag and drop one with its tail near the tip of the other. The dropped vector will snap to the tip when it links. You may continue to link additional vectors in the same way to form a chain.

**FIGURE 16.32** Vectors can be linked in order to show a vector sum.

Note: Tracker makes no attempt to check whether it is mathematically appropriate or physically meaningful to link a given chain of vectors—it simply makes it possible.

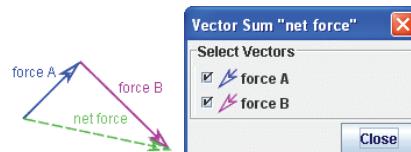
## 16.9 Track Types

359

When you drag the first vector in a chain of linked vectors, the chain moves as a unit and the vectors remain linked. When you drag a vector further up the chain, it breaks the chain into two smaller chains.

**Vector Sum**

A Vector Sum track represents the sum of a collection of vectors. Its vector steps are not marked but instead are determined by the components of the vectors in the collection. The green vector sum in Figure 16.33 represents the sum of vector forces A and B. Vector sums are drawn with a dashed line to distinguish them from independent vectors and motion vectors.



**FIGURE 16.33** A vector sum. Forces A and B are shown linked but need not be.

Select the vectors to include in a vector sum by checking them in its Select Vectors dialog. The dialog is displayed when the vector sum track is initially created or by choosing Select Vectors in its personal menu.

**Line Profile**

A Line Profile track (Figure 16.34) is a tool for measuring brightness and other pixel data along a horizontal line on a video image. At each pixel point along the line, it can average the image pixels above and below the line to reduce noise and/or increase sensitivity.



**FIGURE 16.34** Line profiling tool.

Drag the mouse horizontally to mark the line profile. You need only mark it once since it is identical on all frames. Drag either end of the line horizontally to change its length. The pixels covered by the line are the "pixel points" analyzed by the line profile tool. The maximum length of the line is the pixel width of the video image (for example, 640 pixel point length for a 640 x 480 image). Drag the center of the line in any direction to position it.

To increase the number of pixels sampled for a smoother average you can increase the line profile's Spread and/or Feather. Select the line and enter the

value in pixels in the appropriate field on the toolbar. The spread pixels, which extend immediately above and below the pixel point, are given the same (full) weight in the average as the pixel point itself. The feather pixels, which extend outside the spread pixels, are given a linearly decreasing weight in the average.

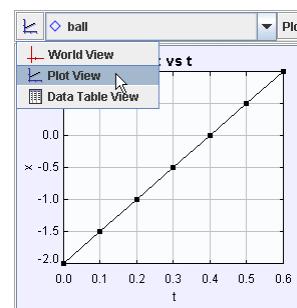


**FIGURE 16.35** Line profile spread and feather.

For a given pixel point, the total number of pixels given full weight is  $1 + 2s$ . The total width of the line profile in pixels is  $(1 + 2s + 2f)$ . The outline of the line profile shows all spread and feather pixels included in the average.

## 16.10 ■ VIEWS

### Data Views



**FIGURE 16.36** Selecting a plot view.

Data Views display information about the world data associated with tracks as measured by the current coordinate system. Tracker provides three standard data view types, discussed below, in up to four split panes. To open a split pane, drag or click on a divider or select the appropriate Window menu item. Each pane displays the view type selected from a list as shown in Figure 16.36. To display the list, click the View button in the upper left corner of the pane.

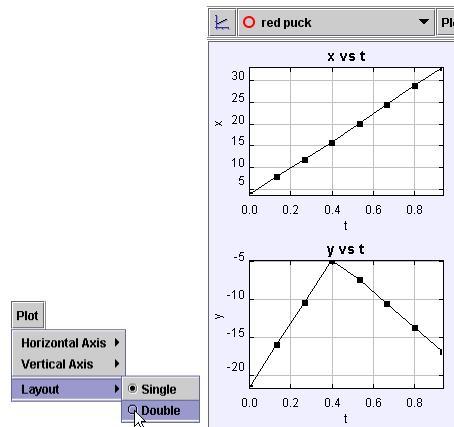
### Plot Views

The Plot View displays one or more graphs of a track's data. The toolbar allows users to select the track. Pressing the Plot button in the allows a user to choose

## 16.10 Views

361

the plot layout and variables for the selected track. A double layout for example, allows the user to display two graphs stacked vertically. Both graphs have the same horizontal variable as shown in Figure 16.37.



**FIGURE 16.37** A double layout for data from the red puck track.

Graphs include both Points and Lines and Autoscale is on by default, but right-clicking (control-click on Mac) brings up a popup menu for controlling these display properties and accessing Print and Help services. Uncheck the appropriate *Auto* box in the *Scale* dialog to set a maximum or minimum value manually. Choose *Scale to Fit* to rescale the graph so all data points are visible when autoscale is off.

### Datatable View

t	x	y
0	0.034	-0.029
0.133	5.179	-0.074
0.268	10.32	-0.054
0.402	15.497	-0
0.535	20.436	0.002
0.668	25.517	-0.048
0.802	30.534	-0.102
0.935	35.48	-0.099

**FIGURE 16.38** A datatable view of the center of mass track.

The Datatable View (Figure 16.38) displays a table of a track's world data. Like the plot view, it has its own toolbar for selecting the track and choosing the

variables. The data displayed in the table can be copied to the clipboard and pasted into a spreadsheet or other application.

Select the data columns to be included in the table by clicking the *Data* button and checking those desired in the dialog displayed.

To copy data to the clipboard, first drag to select the cells of interest, then right-click the datatable (control-click on Mac) and select Copy from the popup menu. If no cells are selected, all cells will be copied.

### World View



**FIGURE 16.39** Video and world views of a collision.

A World View shown in Figure 16.39 displays the video and tracks in world space, the drawing space used by standard drawing panels. The world view has a fixed x-axis pointing to the right and a fixed scale. All tracks that are visible in the main video view (including the axes and tape measure) are also visible in the world view.

Model animations may be overlaid on top of the video and tracks in a world view, enabling a direct visual comparison between theory and experiment. Right-click (control-click on Mac) to display a popup menu with Print, Help and animation tools.

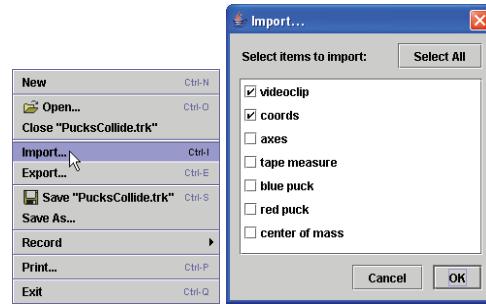
### 16.11 ■ TRACKER XML DOCUMENTS

Tracker saves properties of the video clip, coordinate system and tracks in xml documents with the default extension “trk” (pronounced track). When a saved file is opened, Tracker loads the specified video, sets the clip and coordinate system properties, and recreates the tracks in a new tab that displays the file name. Step positions are saved in image coordinates, so they are not suitable for direct analysis (to access the world data associated with a track, use a datatable view).

A Tracker data file is easily human-read and edited with any text editor. The xml format conforms to the doctype specification defined in osp10.dtd.

### Importing and Exporting Data

Videos and tracks can be imported from a Tracker xml file into the current tracker tab or exported from the tab to a file using the File | Import and File | Export menu items (Figure 16.40). When importing or exporting a data file, the available elements are displayed in a dialog that allows the user to select those desired.



**FIGURE 16.40** Importing data from a saved Tracker xml file.

### Tracker and Launcher

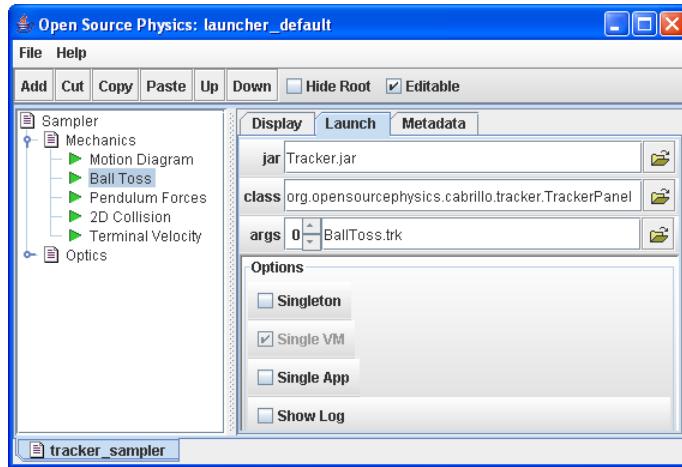
Launcher can be used to open saved xml files in Tracker. Figure 16.41 shows typical launch parameters that open the file "BallToss.trk". Note that the specified class is TrackerPanel rather than Tracker itself. This loads a new tab into the shared (static) Tracker application, so opening multiple files does not open multiple application windows. The *Single VM* option must be checked for this to work.

## 16.12 ■ GLOSSARY

### Videos

- Video: (noun) a digital file containing one or more images.
- Frame: (noun) a single video image.
- Video Clip: (noun) a subset of frames in a video defined by a start frame, step size (number of frames per step), and step count.
- Step: (noun) a frame included in a video clip.
- Filter: (noun) an image processing algorithm applied to a video image

### Coordinates



**FIGURE 16.41** Launching Tracker with the xml document "BallToss.trk".

- Image Position: (noun) the position of a point measured in pixel units relative to the top left corner of the video image. In a 320 x 240 pixel image the upper left corner is at image position (0.0, 0.0) and the lower right is at (320.0, 240.0). Image positions have sub-pixel resolution.
- World Coordinates: (noun) the position of a point measured in scaled world units relative to a reference frame.
- Coordinate System: (noun) the set of transformations used to convert image positions into world coordinates.
- Reference Frame: (noun) the set of origins and angles used to define the coordinate system. The reference frame can be changed without changing the scale.
- Scale: (noun) the number of image units per world unit.
- Calibrate: (verb) to set the scale using video features with known dimensions.

#### Tracks

- Track: (noun) a series of steps in time that define the position and orientation of an object or feature in a video; (verb) to follow and/or identify an object's track.
- Step: (noun) (1) the position and orientation of a track at a single time, typically specified by step number (integer  $\geq 0$ ). (2) a video frame included in a video clip; (verb) to move forward in time to the next step number.

## 16.12 Glossary

**365**

- Point: (noun) a single point on the video image. Points are used singly or in combination to define and/or manipulate a step.
- Footprint: (noun) the visible shape of a step.
- Mark: (verb) to draw a track's steps by clicking and/or dragging the mouse.

**C H A P T E R****17****Easy Java Simulations**

©2005 by Francisco Esquembre, July 2005

The Open Source Physics project includes *Easy Java Simulations*, a high-level authoring tool that can be used both by expert programmers as a fast-prototyping utility, and by novices as a simple tool that will help them create their first simulations. This Chapter provides an overview of this application.

### **17.1 ■ INTRODUCTION**

The OSP library provides a complete and articulated set of Java classes and utilities for programmers who want to create their own computational physics applications in Java. For these users, the library helps speed up the creation process.

However, OSP is just too powerful to reduce its potential use only to programmers. There are a great number of creative teachers (and students) who may not be fluent enough in Java to use the libraries as provided, but who could, if given the chance, develop effective and useful physics simulations.

For this reason, the OSP project aimed, from the very beginning, to find a way for non-programmers to access the concepts and to use the products of the OSP library. *Easy Java Simulations* is the answer to this goal.

*Easy Java Simulations* (*Ejs* for short) is a software authoring tool created in Java which sits on top of OSP libraries. It provides a simplified entry point for those who want to create Java applications or applets that simulate physical phenomena.

If you are an experienced Java programmer already or are on the way of becoming one, you may wonder why you need such an authoring tool when you can program directly? Even though it's true that direct programming gives you full control, before you decide to skip this Chapter completely, consider that you may want to learn more about a tool that can help you to:

- Quickly develop a prototype of an application in order to test an idea or algorithm.
- Boost the creation of sophisticated user interfaces with minimal effort.
- Create simulations whose structure and algorithms other people (especially non-programmers) can easily inspect and understand.
- Invite your students or colleagues (who may be new to Java) to create their own simulations.

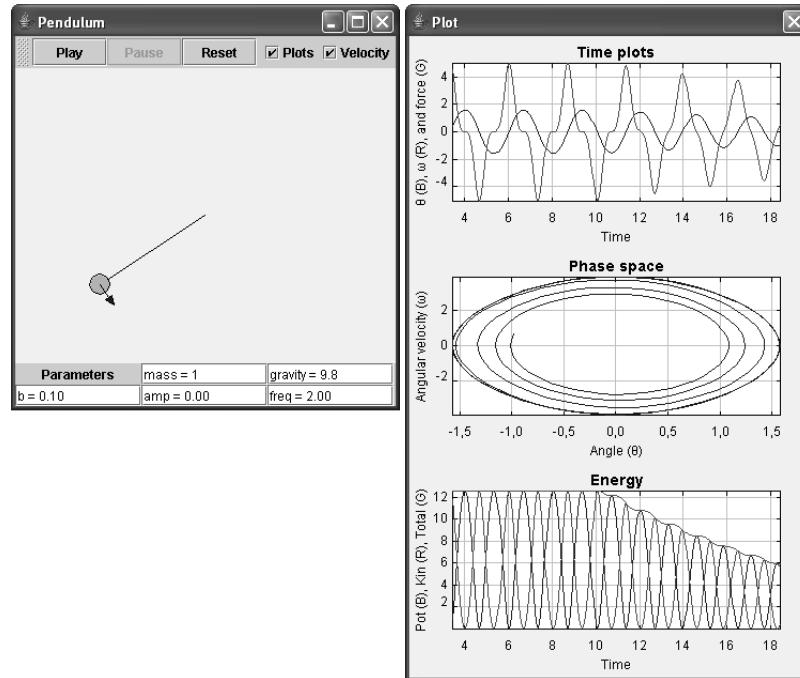
## 17.1 Introduction

367

- Automate production tasks such as preparing your simulations to be distributed using Web pages or Java Web Start technology.

This Chapter provides a general description of *Easy Java Simulations* and does not explain all of *Ejs*' features and possibilities, from the installation of the software to the Web distribution of the generated simulations. A detailed manual is, however, on the companion CD and on the *Ejs* web site. Instead, this Chapter is a short survey that describes how the tool works and how it simplifies the creation of professional OSP simulations.

We will illustrate how to use *Ejs* by working with one of the most famous Physics examples of all time, the simple pendulum. To get acquainted with *Ejs*' main features, we will begin by loading, inspecting, and running an existing basic simulation of this physical system. We will then extend the model to include damping and forcing and to improve the visualization of the phenomenon by including phase-space and energy graphs. Figure 17.1 shows the user interface of the simulation we will obtain once we have finished working with it.



**FIGURE 17.1** A simulation of a simple pendulum created with *Easy Java Simulations*. A damping term was introduced approximately at time  $t = 10$  seconds.

## 17.2 ■ THE MODEL–VIEW–CONTROL PARADIGM MADE SIMPLER

We have mentioned the model–view–control (MVC) paradigm earlier in this guide (see Section 1.2). This was introduced as a way of structuring the different parts of a computer program and has proved to be successful in creating versatile applications in a clean, well-defined, and relatively simple, way.

*Easy Java Simulations* uses this same paradigm. It structures a simulation in two main parts: the model and the view. The model is the collection of variables that define the different possible states of the system under study, together with the algorithms that describe how these variables change in time or how they respond to user interaction. The view is the generic name that *Ejs* adopts for both the visualization of the phenomenon and the user interface of the application.

In essence, the *Ejs* view combines the control and view of the MVC paradigm. This simplifies things for beginners because in modern computer simulations the control is achieved through interaction of the user with the application’s graphical user interface.

Because our simulations have mainly a pedagogical purpose, we will add to the model and view a textual (multimedia) part that is designed to contain a short introduction to the simulation or operating instructions for the user. Thus, an *Ejs* simulation consists of three main parts: the introduction, the model and the view. Figure 17.2 shows the interface of *Easy Java Simulations* (to which we have added some notes), which reflects this structure.

You are invited to run *Ejs* while you read this chapter. Although you can find detailed instructions on installing and running it in the manual included in the companion CD, here are some quick guidelines (for those who don’t like reading manuals!). To install and run *Ejs*, follow the next steps:

**Install Java 2 JDK.** *Easy Java Simulations* is a Java program that compiles Java programs, hence it requires that you install Java Development Kit (Java 2 JDK) in your computer. The recommended version at the moment of this writing is 1.5.0\_04.

**Copy *Ejs* to your hard disk.** Installing *Ejs* requires only copying the **Ejs** directory to your hard-disk. This directory is usually distributed as a zip (compressed) file. Just uncompress this file to any suitable directory. (In Unix-like systems, the directory may be uncompressed as read-only. In this case, please enable write permissions for the whole **Ejs** directory.)

**Run *Ejs*’ console.** *Easy Java Simulations* is now installed. The distribution includes an **EjsConsole.jar** file that runs a console that lets you do the final configuration for the installation and to run *Ejs* itself. Run this file by either double-clicking on it (if your system allows you to run it this way) or typing the command:

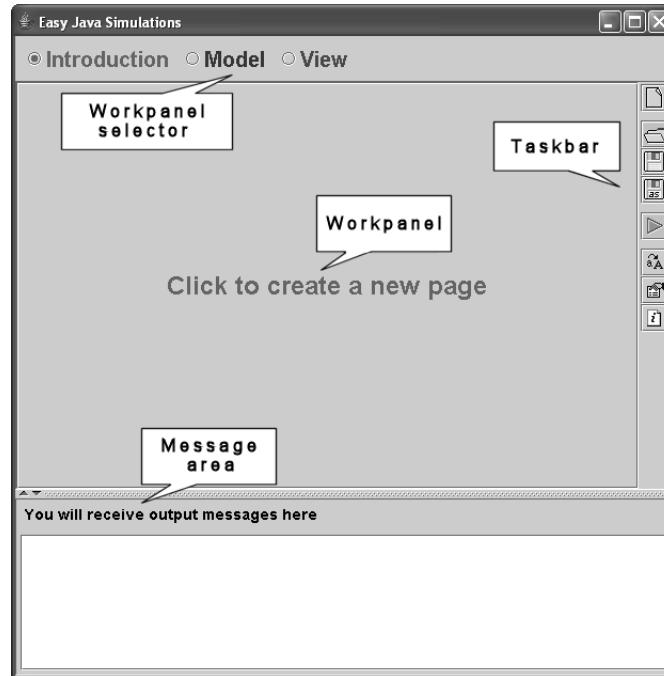
```
java -jar EjsConsole.jar
```

at a system terminal window. You should get the window shown in Figure 17.3.

**Tell *Ejs* where you installed the JDK.** (This is only required in Windows

## 17.2 The Model–View–Control Paradigm Made Simpler

369



**FIGURE 17.2** *Easy Java Simulations* user interface (with annotations).

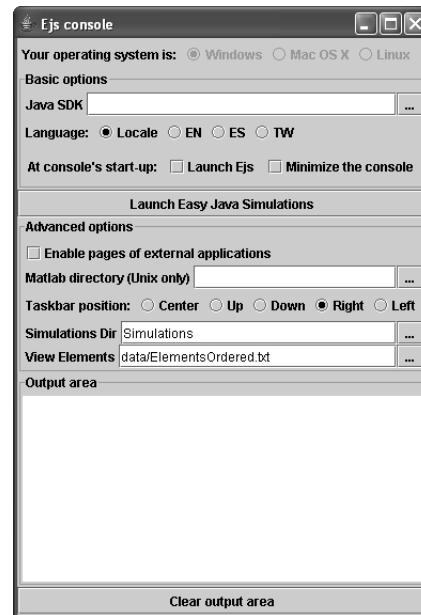
and Linux.) Because you may have installed Java 2 JDK in any directory, you will need to enter this directory in the Java JDK field of the console.

**Run *Ejs*.** Just click on the Launch Easy Java Simulations button of the console.

That's it! You should get the interface displayed in Figure 17.2. Again, you will find more detailed instructions in the CD and also in *Ejs*' home page <http://fem.um.es/Ejs>.<sup>1</sup>

As you can see in Figure 17.2, the interface of *Ejs* is rather basic. This was a deliberate design decision. We wanted to make clear from the very beginning that *Easy Java Simulations* is simple. Hence, we avoided providing a large and overwhelming number of icons and menu entries in the *Ejs* interface. Despite its simplicity, however, the tool has everything that it needs to build controls, models, and views.

<sup>1</sup>For operating system purists, *Ejs*' installation includes three script files that launch *Ejs* in the different platforms: **Ejs.bat** for Windows, **Ejs.macosx** for Mac OS X, and **Ejs.linux** for Linux. Before running the file corresponding to your operating system, you may need to slightly edit the script to correctly set the **JAVAROOT** variable (which is defined in the first lines of the script) that points to the directory where you installed the Java JDK.



**FIGURE 17.3** The *Ejs*' console that will help you run *Easy Java Simulations*.

We start exploring the interface by looking at the set of icons on the right hand side taskbar of Figure 17.2. The taskbar provides an icon for each of the main functionalities of *Ejs*. We will explain the purpose of some of these icons in this chapter, but their meaning and use should be rather natural.

You will also notice in Figure 17.2 that there is a blank area at the lower part of the window with a header that reads “You will receive output messages here”. This is a message area that *Ejs* uses to display information about the results of the actions we ask it to take.

Finally, the most important part of the interface is the central area of the interface (labeled the Workpanel in the figure), and the three radio buttons on top on it, Introduction, Model and View. These radio buttons are used to display the introduction, the model and the view in the workpanel.

### 17.3 ■ INSPECTING AN EXISTING SIMULATION

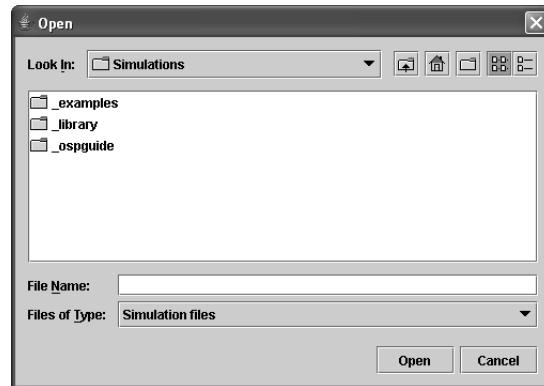
We can better understand the role of the different parts of a simulation, and the utility of the different panels of *Ejs*, by loading and inspecting an existing simulation. We choose for this a basic implementation of the simple pendulum that we created for your perusal and that is included in the distribution of *Ejs* in the companion CD.

If you click on the Open icon of the taskbar, , a file dialog box will open

## 17.3 Inspecting an existing simulation

371

from which you can load a simulation file. See Figure 17.4.



**FIGURE 17.4** File dialog box used to load an existing simulation file.

The file dialog box first displays the contents of *Ejs*' working directory called **Simulations**. From there you can access any directory or file in the standard way. Open the directory called **\_ospguide** and select from it the file called **PendulumBasic.xml**. Click the Open button and *Ejs* will load a basic version of the simulation of a simple pendulum.

The interface of *Ejs* will change noticeably. It will load the different parts of the simulation file in the corresponding panels, allowing us to see how the simulation has been designed. It will change its title to include the name of the simulation file, and, finally, it will also display a message, in the message area, reporting that the file has been loaded.

Two new windows will also appear. You can see them in Figure 17.5. They correspond to the view of the simulation, which we will describe a bit later.<sup>2</sup> For the moment, however, we will concentrate on the interface of *Ejs* itself.

What we see in this interface depends on which part of the simulation was visible when we loaded the file. Because we are interested in learning how to use *Ejs* to specify a real simulation, we will inspect the different parts of the simulation in turn by browsing the different panels of *Ejs*.

### The Introduction

The first panel is the introduction of the simulation, shown in Figure 17.6, which consists of a html page with a short introduction to the problem. This part of the interface of *Ejs* provides a simple editor that can be used to both visualize and edit standard html pages. Right now, we see the editor in its read-only mode. We will learn in Section 17.5 how to set it to edit mode and actually change its contents.

<sup>2</sup>To be more precise, these two windows are, as their titles state, *Ejs windows*. That is, windows that *Ejs* displays to help the author configure the view. Hence, they are really mock-ups of what the real

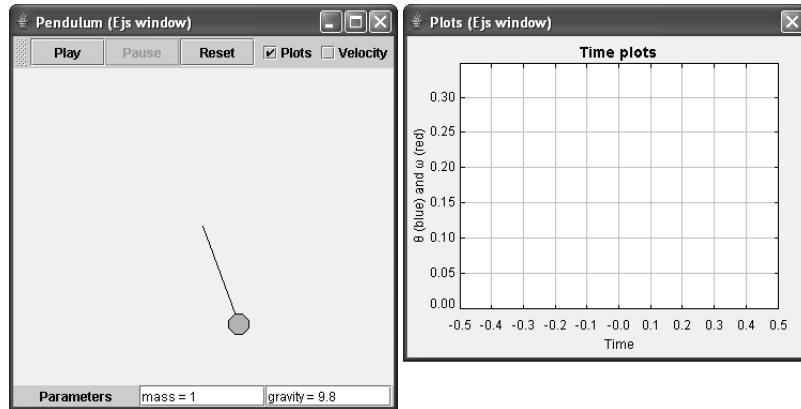


FIGURE 17.5 The two windows for the view of the simulation.

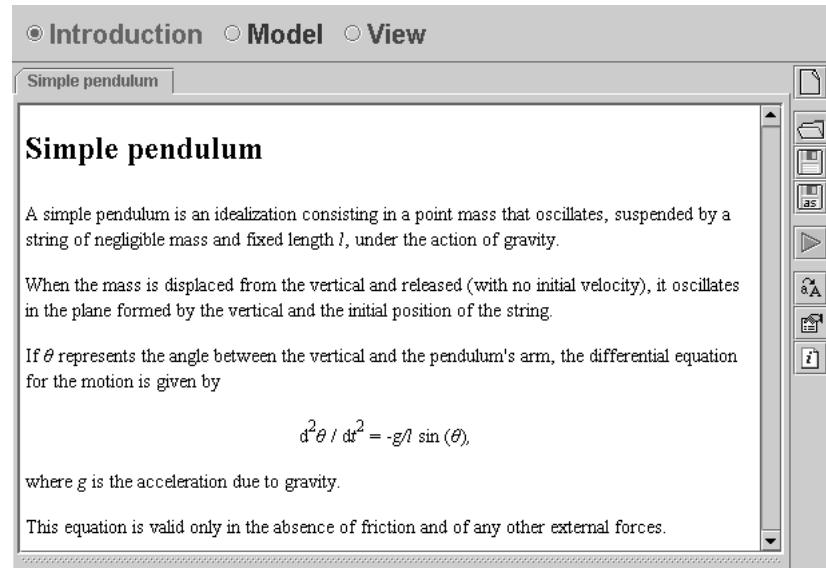


FIGURE 17.6 The introduction panel for the simulation of a simple pendulum.

### The Model

The second part of the simulation, the model, is more interesting. As you may expect, this is the part where all the physics goes. The physical model of a simple pendulum without friction and without an external driving force is contained in

view will look like, but they are not operative. Notice also that this view is different from the view of the final simulation we will obtain at the end of this chapter, the one displayed in Figure 17.1.

## 17.3 Inspecting an existing simulation

373

the differential equation:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l} \sin(\theta), \quad (17.1)$$

where the variable  $\theta$  corresponds to the angle between the pendulum's arm from the vertical,  $g$  is the acceleration due to gravity, and  $l$  is the length of the pendulum.

This second-order, non-linear, ordinary differential equation (ODE) must be solved numerically because there is simply no way to express its solution in terms of elementary functions.<sup>3</sup> To this end, we need to express the second-order differential equation as an equivalent system of two first-order ODEs, introducing the auxiliary variable  $\omega$ , the angular velocity:

$$\frac{d\theta}{dt} = \omega \quad (17.2)$$

$$\frac{d\omega}{dt} = -\frac{g}{l} \sin(\theta). \quad (17.3)$$

Solving this system of ODEs for the given parameters will give us the evolution of the angular position (and velocity) of the system in time.

The second part of *Easy Java Simulations*, the model, offers us a set of five subpanels that can be used to accomplish the tasks needed to solve this problem. These five subpanels are: Variables, Initialization, Evolution, Constraints, and Custom. In the *Ejs* pendulum example, select the Model radio button in order to view the model panel. We now consider the five subpanels one after the other.

#### *Declaring the Variables*

In the context of a computer program or simulation, the state of a physical system is determined by the values of a set of variables. Accordingly, to begin modeling our simple pendulum, we must provide *Ejs* with the set of variables that defines the state of the physical system. This is done by editing a simple table of variables in the Variables subpanel of the model part of the interface of *Ejs*. See Figure 17.7.

In this table we have declared all the variables involved in the differential equations as well as other parameters, such as the mass,  $m$ , (which we don't need for the moment, but that we will need later on) and the time interval,  $dt$ , at which we want to obtain information from the system. Finally, you will notice that we have also declared the variables  $x$ ,  $y$ ,  $vx$ , and  $vy$ , that hold the position and the velocity of the pendulum's bob. They will serve to configure the view to display a realistic visualization of the pendulum.

<sup>3</sup>To be more precise, the case we consider here in the absence of friction and of any other forces can be explicitly solved using elliptic functions. However, we adopt the numerical approach right from the start in order to be able to deal later with the more general case.

Name	Value	Type	Dimension
m	1.0	double	
g	9.8	double	
I	1.0	double	
theta	Math.PI/9	double	
omega	0.0	double	
t	0.0	double	
dt	0.05	double	
x	<code>l*Math.sin(theta)</code>	double	
y	<code>- l*Math.cos(theta)</code>	double	
vx	<code>omega*l*Math.cos(theta)</code>	double	
vy	<code>omega*l*Math.sin(theta)</code>	double	

Comment

**FIGURE 17.7** Table of variables that describe the state of a simple pendulum.

If you want information about the role of a particular variable, you can select it in the table and the comment field at the bottom of the page will display a short description of that variable.

#### *Initializing the Model*

The system must be initialized to a valid state before letting the time run. There are two basic ways of initializing variables in *Ejs*. The first one is using the corresponding cells of the column *Value* in the table of variables. Just type a constant value, or a simple expression, and it will be assigned to the variable at start-up. This is the option we have used for this simulation.

A second possibility, which is required if your program needs to do some more complex computations to initialize the system, is to use the *Initialization* subpanel provided by *Ejs*. In this subpanel you can type the Java code for the algorithm that will compute the correct values for the variables that require these extra computations. *Ejs* helps to keep this process simple, because you just need to write the Java sentences that contain the algorithm for your computations, and *Ejs* will automatically wrap these algorithms into a Java method and will take care of calling it at start-up or whenever you reset the simulation.

Because the system has been completely initialized in the table of variables, the initialization panel remains empty. We'll have the opportunity to show how to write such a page of Java code when we cover constraints.

## 17.3 Inspecting an existing simulation

375

*The Evolution of the Model*

Specifying the evolution of the model consists of providing the algorithms that describe how the state of the system changes in time. That is, what happens to the values of the variables of the system when it evolves in time. In our case, this corresponds to solving numerically the differential equations of the model.

*Ejs* offers two possibilities for this. The first one is a plain editor for Java code where the author can write directly the numerical algorithm required. You would typically use this option if your main interest is simply numerical algorithms. However, because it is a common situation that the evolution of a system is specified by a system of ODEs, and writing the code to solve this type of problem with accuracy can be tedious (if not difficult), *Ejs* also offers a dedicated built-in editor. This editor allows us to easily enter the system of ODEs, and it automatically generates the Java code (based on OSP numeric classes) corresponding to some of the most popular numerical algorithms to solve the equations.

This second possibility is the one we chose for our system. If you inspect the Evolution subpanel of *Ejs*, you will see that it contains our system of ODEs in this specialized editor, as shown in Figure 17.8.

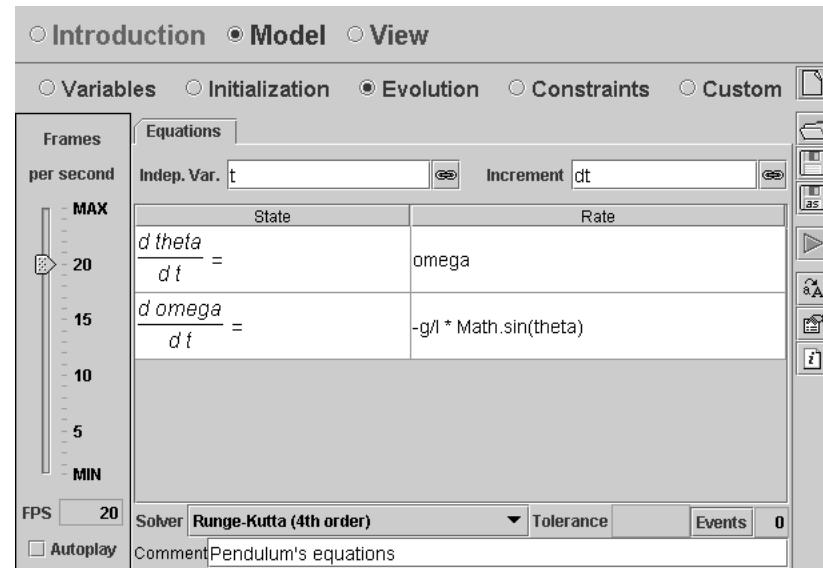


FIGURE 17.8 Equation editor for our simulation with the pendulum's system of ODEs.

One advantage of this editor is that it displays the system of ODEs in a very familiar way, one that your students can easily recognize and that is easy to understand and modify. The second advantage is that the editor takes care of the worst part of solving the equation: coding the algorithm. Solving differential equations numerically is a sophisticated task, but *Ejs* (with the help of OSP classes) has automated it in a very convenient way.

You will notice in Figure 17.8 that we have chosen  $t$  to be the independent variable in the model and  $dt$  to be the increment for it in each evolution step. This means that the evolution page should solve the equation to move from the current instant of time at  $t$ , to the next instant of time at  $t+dt$ .

We have chosen the standard 4th-order Runge-Kutta algorithm, as the Solver field immediately below the equations indicates. You will also notice that there are two extra fields, one called Tolerance (that is used only for adaptive algorithms), and one called Events. This last field is used to define and handle state-events, such as collisions, that may occur in the life of the differential equation. (Events are described in detail in the manual on the CD.) In our case, the simple pendulum model includes no events.

All together, this page describes, using the editor for ODEs, what will happen as time passes: the differential equations will be solved numerically for an increment of  $dt$  of the independent variable. The controls you see in the left-hand side of Figure 17.8 are used to tell *Ejs* how often the evolution should take place when the simulation runs. As the figure shows, we have instructed *Ejs* to run the evolution 20 times, or frames, per second. This, together with the value of 0.05 that we used for  $dt$ , results in a simulation which runs (approximately) in real time.

Finally, there is a checkbox labeled Autoplay. This instructs *Ejs* to run the evolution as soon as the program runs. We left it unchecked because we want to offer the user the possibility of changing the position of the pendulum and then to click on a button that will start the animation.

### *Constraints Among Variables*

The fourth subpanel of the model is called Constraints. This panel is used to write Java code that establishes fixed relationships among variables. Consider, again, the pendulum example as shown in Figure 17.9.

The evolution of our model solves the ODEs in terms of the angular magnitudes, theta and omega. However, we are interested in displaying on the screen the actual position of the pendulum and its velocity vector, and, for this, we need the corresponding cartesian coordinates of both position and velocity. These can be easily derived from the angular magnitudes using the formulas:

$$x = l \sin(\theta) \quad (17.4)$$

$$y = -l \cos(\theta) \quad (17.5)$$

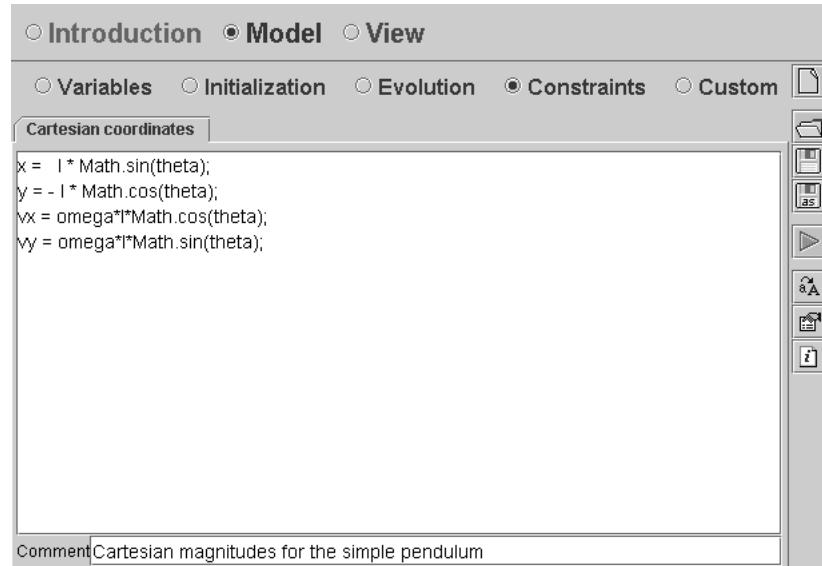
$$v_x = \omega l \cos(\theta) \quad (17.6)$$

$$v_y = \omega l \sin(\theta). \quad (17.7)$$

This is what we call “fixed relationships among variables.” This means that, once we know the values of  $l$ ,  $\theta$ , and  $\omega$ , the other variables can be easily obtained using the expressions above. Our constraints consist of translating these expressions into Java so that *Ejs* can use them whenever it is needed.

## 17.3 Inspecting an existing simulation

377



**FIGURE 17.9** Constraints pages that compute the cartesian magnitudes of a simple pendulum.

Notice that the page contains only the Java code for our expressions. *Ejs* will take care of wrapping this code into a Java method and automatically calling this method whenever it is necessary. This simplifies our programming task.

Now comes a subtle point. One of the questions most frequently asked by new users of *Ejs*: “Why don’t we write these equations into the evolution instead of in the constraints?”. The reason is that this relationship among variables must *always* hold, even if the evolution is not running. It could very well happen that the simulation is paused and the user interacts with the simulation to change the angle *theta* (or *omega*, or *l*). If we write the code to compute the cartesian magnitudes in the evolution, the values for the variables *x*, *y*, *vx*, and *vy* will not be properly updated, because the evolution is only evaluated when the simulation is playing.

Constraint pages, on the other hand, are always automatically executed after the initialization (at the beginning of the simulation), after every step of the evolution (when the simulation is playing), and each time the user interacts with the simulation. Therefore, any relationship among variables that we code in here will always be verified.

You could argue here that, since constraints are evaluated also at start-up, there was no reason to initialize the variables *x*, *y*, *vx*, and *vy* in the table of variables since the evaluation of the constraints will assign them the correct values at start-up. You would be almost right. There is still a reason for doing what we did, though.

*Ejs* doesn’t evaluate constraints itself (the generated simulation does), but it evaluates expressions in the *Value* column of the table of variables. Hence, we wrote the initial expressions for these variables so that the pendulum’s

bob appears at the right place in the mock-up of the view displayed. But it is true that, in general, constraints can also be used to initialize variables.

#### *Custom Pages of Code*

The final subpanel of the model is called *Custom*. This panel can be used by the author of the simulation to define his or her own Java methods. Different from the rest of panels of the model, which play a well-defined role in the structure of the simulation, methods created in this panel must be explicitly used by the author in any of the other parts of the simulation. Again, in our example, this panel is empty and is not displayed.

Although *Ejs* is designed to make programming as simple as possible and includes the typical tools an author may need, it also opens a way for programmers to use their own Java libraries. The custom panel of the model offers a simple mechanism to add external Java archives of compiled classes, packed in jar or zip form, to your simulation. The procedure is explained in detail in the manual.

#### *The Model as a Whole*

Our description of the model is ready, and we can look at it as one unit to describe the integral behavior of all the subpanels of the model.

To start the simulation, *Ejs* declares the variables and initializes them, using both the initial values specified in the table of variables and whatever code the user may have written in the initialization subpanel. At this moment, *Ejs* also executes whatever code the user may have written in the constraint pages. This is so that all dependencies between variables are properly evaluated. The system is now correctly initialized.

When the simulation plays, *Ejs* executes the code provided by the evolution subpanel and, immediately after, the possible constraints (for the same reason as above). Once this is done, the system will be ready for a new step of the evolution, which it will repeat at the prescribed speed (number of frames per second).

As mentioned already, note that methods in the custom subpanel are not automatically included in this process.

This simple mechanism provides a basic, but very effective, structure for novices (and experts!) to build their simulations. The author just fills in the subpanels of the model, usually from left to right, and *Ejs* handles the pieces, automatically taking care of all technical issues required (such as multitasking and synchronization).

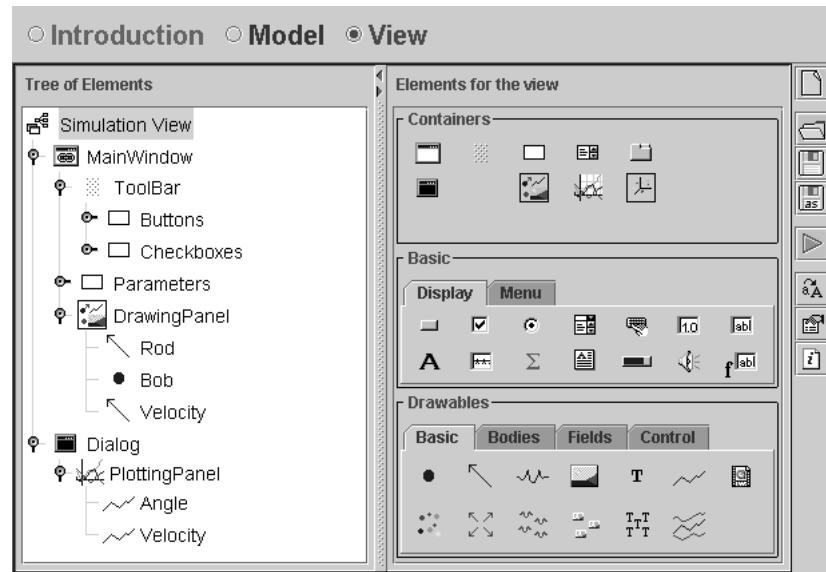
### **The View**

We now turn our attention to the view of the simulation. Recall that two new windows appeared when we loaded the simulation (see Figure 17.5). To learn how this view has been constructed, we can inspect the *View* panel of *Ejs*. Figure 17.10 displays this panel, where two frames, each with several icons, are shown. The frame on the right-hand side displays the set of graphical elements that *Ejs* offers

## 17.3 Inspecting an existing simulation

379

to authors for the creation of a view, grouped by functionality. The frame on the left-hand side shows the actual elements that have been used for this particular simulation.



**FIGURE 17.10** Tree of elements for the simulation (left) and set of graphical elements (right) of *Ejs*.

The view panel of *Ejs* can be considered as an advanced drawing tool which specializes in the visualization of scientific phenomena and its data and user interaction. Obviously, to completely master the creation of views, an author needs to become familiar with all the graphical elements offered and what they can do. (A description of all possible view elements is out of the scope of this chapter, but a complete reference can be found on the companion CD.)

Creating a view with *Easy Java Simulations* takes two steps. The first step is to build a tree-like diagram of the objects (elements) that will make up the user interface. Each element is designed for a given graphical or interactivity task, and our job is to select the elements that we need and combine them appropriately to build our view. Selecting and adding new elements to a view is done in a simple “click-and-use” way which we will illustrate in Section 17.5.

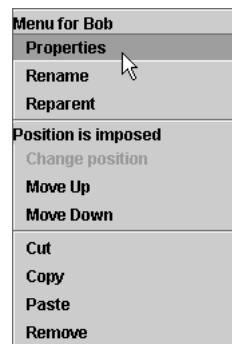
Some elements are of a special family called *containers*, which can be used to group other elements, thus forming the tree-like structure of the elements shown in the figure.

The second step, less evident but also simple, consists of customizing the selected view elements by editing their so-called *properties*. Properties are internal fields of an element that can be changed to make the element look and behave in a particular way. The key point is that properties can be given constant values (for

instance to customize fonts and colors), but they can also be *linked* to variables of the model (typically for positions, sizes and labels with numerical displays).

Because linking is a two-way mechanism, this second possibility is what really turns the view into a dynamic, interactive visualization of the physical phenomenon. Hence, once an element property is linked to a model variable, any change in the variable (due for instance to the evolution of the model) is automatically reported to the view element which changes its graphical aspect accordingly. But, also, if the user interacts with any view element to modify any of its properties (typically doing a gesture with the mouse or keyboard), the change is automatically reported back to the model variable, therefore changing the state of the system.

This basic mechanism is a very simple and effective way to design interactive user interfaces. Let us see an example of how it works in practice. Select the panel for the view and right-click on the element called Bob of the tree of elements of our simulation. This element corresponds to the circle displayed as the bob of the pendulum. The popup menu shown in Figure 17.11 will appear.



**FIGURE 17.11** Popup menu for the element Bob of the view.

Select the option **Properties** from this menu (the one highlighted in the figure) and a new window will appear with the table of properties for this element.<sup>4</sup> All that is required is to edit this table according to our needs. Figure 17.12 reflects the properties we defined for this particular element.

This table of properties illustrates very well all the possibilities offered by element properties. In the first place, you can see in the figure that some properties are given constant values. For instance, those which specify the color, drawing style, and size of the element (which will be displayed as a cyan-colored ellipse of size (0.2,0.2) units). You can also see that the element is enabled, that is, that it will respond to user interaction.

Secondly, you will observe that other properties of the element are given the value of model variables. In particular, the properties X and Y, which correspond to

<sup>4</sup>Double-clicking on the element's node in the tree is a shortcut for this option.

## 17.3 Inspecting an existing simulation

381

the center of the circle in its parent drawing panel, have been linked to the model variables  $x$  and  $y$  by the simple fact of typing their names in the corresponding property field. This connection is the magic. Automatically, the circle will move according to the successive values of the model variables  $x$  and  $y$  when the simulation runs. Also, if the user drags the element Bob with the mouse, the model variables  $x$  and  $y$  will be accordingly changed.

As described above, every time the user interacts with the simulation's view, *Ejs* will also automatically execute the constraints of the model. This ensures that any change that the interaction caused to variables linked to properties is correctly propagated to other variables which may depend on those.

Finally, there is a third type of property that needs to be taken care of for this example to work properly. Recall that the model computes primarily the values of the angular magnitudes, and that we wrote constraints to make sure that the cartesian magnitudes (which includes  $x$  and  $y$ ) were readily computed to match those. This means that if we interact with the pendulum bob to change the values of  $x$  and  $y$ , the change will be overwritten by the constraints unless the change does affect the angular variables. This can be easily taken care of by means of the (somewhat special) *action* properties of the element. These correspond to pieces of Java code that the computer will evaluate whenever the prescribed interaction takes place.

In our case, we need to edit the action property called *On Drag*, which is evaluated every time we drag the element on the screen. We have set this property to execute the following sequence of sentences:<sup>5</sup>

```
theta = Math.atan2 (x,-y);
l = Math.sqrt (x*x + y*y);
omega = 0.0;
```

<sup>5</sup>Unlike other properties, action properties can span more than one line. When this happens, the property field changes its background color slightly. To better display this code, we can click on the first button to its right, , and an editor window will display it more clearly.

Properties for element Bob									
Position and Size				Visibility and Interactivity			Graphical Aspect		
X	x			Visible	<input type="checkbox"/>			Style	ELLIPSE 
Y	y			Enabled	true			Position	
Z				Actions			Rotate		
Size X	0.2			On Press				Fill Color	cyan 
Size Y	0.2			On Drag	theta = Math.atan2 (			Edge Color	
Size Z				On Release	l = Math.sqrt (x*x + y*y);			Stroke	
Scale X									
Scale Y									
Scale Z									

FIGURE 17.12 Table of properties for the element Bob of the view.

The first two statements use the (new) values of the `x` and `y` variables to compute the new length of the pendulum and its angular position. We have added a third sentence that sets the angular velocity to zero, because we want the motion to re-start from rest. All together, these sentences help set up a new state for the model of the system.

The three types of customization we did to the properties of this view element illustrate how *Easy Java Simulations* combines the creation of the control and the visualization tasks of a simulation in one single process (the view). By using either constants, model variables, and Java expressions and sentences, we can configure a user interface that is used simultaneously to display data, to provide input, and to execute control actions on the simulation.

You can inspect the properties of other elements of this view to become familiar with the different types of view elements offered by *Ejs* and their properties. Of particular interest are the elements called `Angle` and `Velocity`, which correspond to time plots of these magnitudes. For each element, you can click on , the first button to the right of each property box, to bring up an editor for that property. For example, the `Style` property editor will allow you to select the shape of the element, and the `Fill Color` property editor will allow you to select the color from a palette.

## 17.4 ■ RUNNING A SIMULATION

Once we have inspected the different parts of the simulation, we are ready to run it. Here goes a warning that may arrive too late if you are running *Ejs* while reading these pages: don't try to run the simulation by clicking on the buttons of the windows in Figure 17.5! Recall that these were just "Ejs windows". Hence, they are not part of the real simulation, but just a mock-up of what the real view will look like. Their purpose is to help the author design the view, but they are not operative.

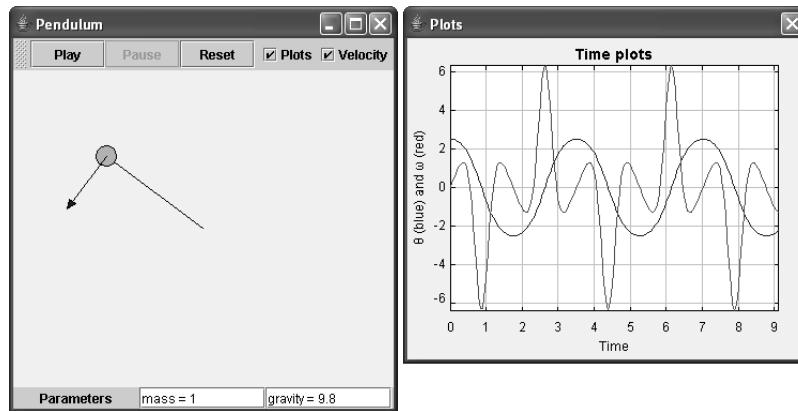
To actually run the simulation, you need to click on the Run icon of *Ejs*' toolbar, . When you do this, *Easy Java Simulations* collects all the information provided in its panels and subpanels, and constructs a complete, independent simulation out of it, taking care of all the required technical subtleties. This includes generating the complete Java source code for the simulation, compiling it, and packing it into a single jar file. Finally, it also runs this file, which will initialize the model and display the simulation's view in the computer screen.

This view is now fully interactive. You can drag the pendulum to any desired position, change any of its parameters, and then click the Play button. The pendulum will oscillate as the model solves the underlying differential equations. The view will display both the pendulum's oscillations and the time plot of the position and (optionally) the velocity. See Figure 17.13.

Simulations created with *Easy Java Simulations* are independent of it once generated. This means that final users don't need to install *Ejs* to run the simulations. They simply double-click the jar file. Moreover, when you have success-

## 17.4 Running a Simulation

383



**FIGURE 17.13** The simple pendulum displaying oscillations with a big amplitude. The largest plot corresponds to the angular velocity.

fully run a simulation once, you'll find everything you need to run and distribute the simulation in the **Simulations** directory. We now briefly discuss the different ways to run a simulation created with *Easy Java Simulations*, together with some remarks about its distribution.

### Running the Simulation as an Application

After running the simulation, you will find in the **Simulations** directory a jar file with the same name as the simulation file we first loaded (if only with its first letter in lowercase, a style custom in Java programming). Recall that the name of this file was **PendulumBasic.xml**. Hence, you will find there a jar file called **pendulumBasic.jar**.

This is a self-running jar file. Hence, if your operating system is properly configured, you will be able to run the simulation by double-clicking on the jar file icon. (Windows and Mac OS X are usually automatically configured for this if the Java runtime environment is installed.) If double-clicking won't work, then you can still use the launch file called **PendulumBasic.bat**. This has been generated by *Ejs* specifically for your operating system and can be executed like any other batch (Windows) or shell script (Unix-like systems) on your operating system.

This jar and launch file will work correctly assuming that you have the Java runtime environment (JRE) installed in your system and that you run them from within the **Simulations** directory. If you want to move your simulation to a different directory or computer, read the distribution notes below.

### Running the Simulation as an Applet

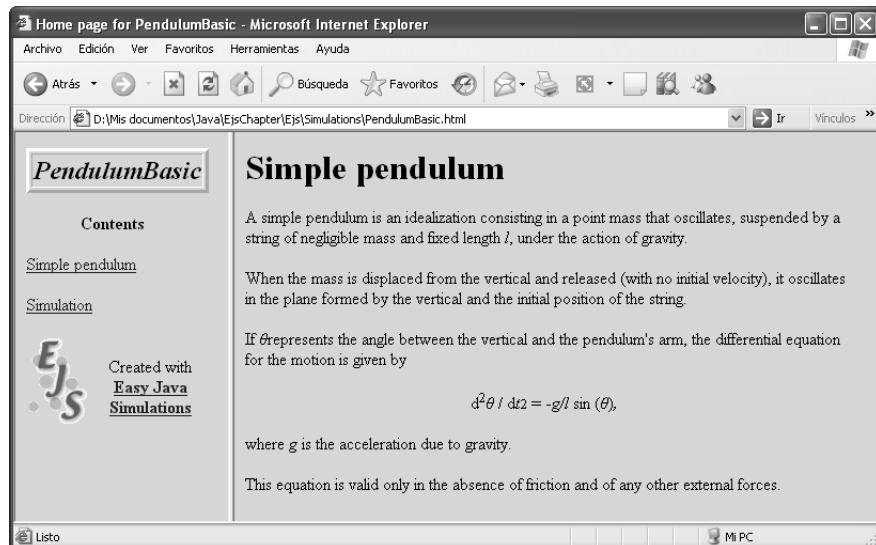
A second possibility to use the simulations created with *Ejs* is to run them as Java applets from within html pages. This is a very attractive possibility because it

## Chapter 17 Easy Java Simulations

opens the world for distributing simulations over the Web (see also the note about the Java Web Start technology below).

This possibility is also taken care of by *Easy Java Simulations*. Every time you run a simulation from *Ejs*, it also generates the html pages required to wrap the simulation in form of an applet. More precisely, it will create a complete set of html pages, one for each of the introduction pages (those in the introduction panel of *Ejs*) and one that contains the simulation as an applet. It finally creates a master html file that structures all the others using a simple set of frames.

All these html files are easily identified because they begin with the same name as your original simulation file. In particular, the name of the master file is the same as that of the simulation file. In our case, **PendulumBasic.html**. If you load this file in a Java-enabled browser, you will see something like Figure 17.14.



**FIGURE 17.14** The set of html pages created for our simulation.

Notice that the frame on the left displays a table of contents that includes the introduction page that we created for our simulation (which is also shown in the right frame) and a second link for the simulation itself. If we click on this link, the frame to the right will change to display a html page with the simulation embedded as an applet. See Figure 17.15. (The dialog window with the time plots is displayed separately. We don't reproduce it here.)

Notice, finally, that the html page that includes the simulation also displays a set of buttons that can be used to control the simulation using JavaScript.<sup>6</sup>

<sup>6</sup>JavaScript is a scripting language that can be used in html pages for simple programming tasks. The use of JavaScript to access methods of simulations created with *Ejs* is described in the *Ejs* manual.

## 17.4 Running a Simulation

385

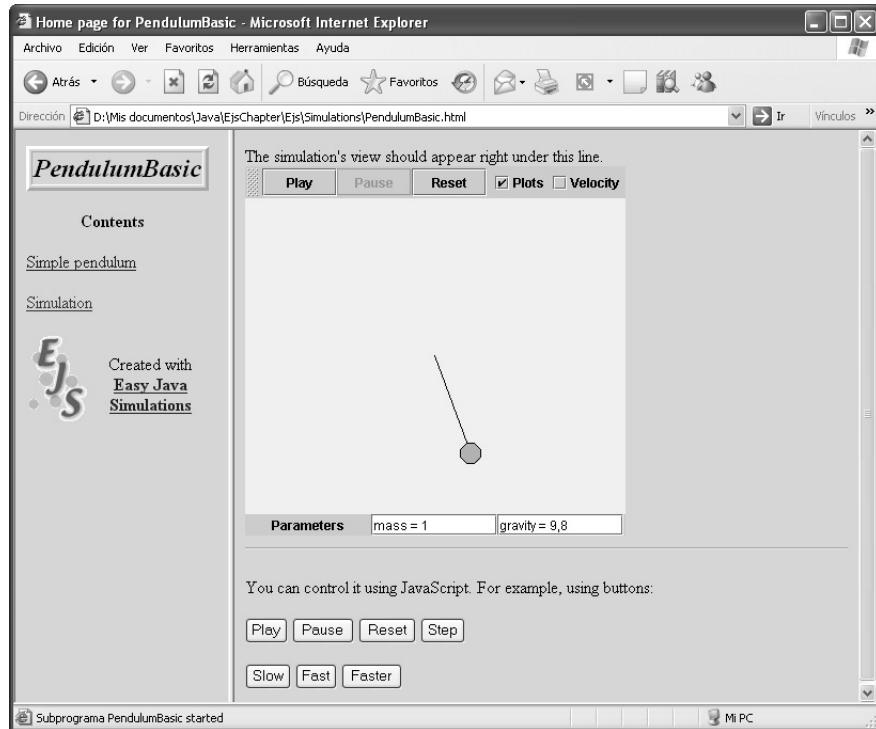


FIGURE 17.15 The simulation running as an applet.

Simulations created with *Ejs* require Java 2 to run. Thus, your browser will need to have a recent plug-in installed to display them properly. We recommend the Java plug-in version 1.5.0.04.

### Distributing a Simulation

As we said before, simulations created with *Easy Java Simulations* are independent of it once generated. However, the simulation will need to use a set of library files that include all the compiled OSP classes that provide the background functionality for your simulation, from numerical methods to the visualization elements. This library can be freely distributed and is contained in the **\_library** subdirectory of the **Simulations** directory. Finally, if you designed your simulation to use any additional file, such as a GIF image or sound file, you will need to distribute this too along with the simulation.

With this said, the distribution process is quite simple. You just need to copy the required files and the **\_library** directory to the distribution media or Web server. Simulations created with *Ejs* can be distributed using any of the following ways:

**Web server** Just copy the jar, html, and auxiliary files for the simulation in a suit-

able directory of your Web server. Finally, copy the `library` directory into the same directory as your simulation on the Web server. Recall that your users will need to have a Java 2 plug-in enabled web browser to properly display the simulation.

**CD-ROM** This procedure is similar to the previous one. Just copy the jar, html, and auxiliary files, and the `library` directory into your distribution media. Your users will need to have the JRE installed to run your simulation.

*Easy Java Simulations* is compatible with **Launcher** (see Chapter 15) and the *Ejs*' installation includes a copy of **LaunchBuilder** that will scan your **Simulations** directory and will automatically generate the XML file that you need to run your simulations using **Launcher**. You can then distribute this XML file along with your simulation files, so that your users can use **Launcher** to run them.

**Java Web Start** This is a technology created by Sun to help deliver Java applications from a Web server. The programs are downloaded from a server at a single mouse click, and they install automatically and run as independent applications. *Easy Java Simulations* is prepared to help you deliver your simulations using this technology, and it can automatically generate a Java Web Start jnlp file for your simulation. Details are provided in the manual. Your users will need to have the JRE installed, which includes Java Web Start. Finally, your server will need to report a MIME type of `application/x-java-jnlp-file` for any file with the extension `jnlp`.

**Together with *Ejs*** A final possibility that is worth considering is that of distributing your simulation files together with *Ejs*. That is, asking your users to run the simulations using *Ejs* itself. This requires your users to learn how to use *Ejs*, if only basically to load and run the simulations. But it has, in our opinion, the enormous advantage that your users can not only run the final simulation, but also inspect it in detail and learn how you actually simulated a given phenomenon. This possibility also offers a simple way for students to begin programming to simulate physical phenomena, which we find of great pedagogical value.

## 17.5 ■ MODIFYING A SIMULATION

If you read up to this point, you should already have a general impression on how *Easy Java Simulations* works, how it can be used to create a simulation, and how to run and distribute the simulations you create with it. In this section we will modify the simulation of the simple pendulum to include new features. This will further illustrate the operating procedures required to work with the different panels of *Ejs*.

In particular,

## 17.5 Modifying a Simulation

387

1. We will modify the model to add friction and an external driving force. The resulting second-order differential equation is:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l} \sin(\theta) - \frac{b}{m} \frac{d\theta}{dt} + \frac{1}{ml} f_e(t), \quad (17.8)$$

where  $b$  is the coefficient of dynamic friction,  $m$  is the mass of the pendulum, and  $f_e(t)$  is a time-dependent external driving force. We will use, in particular, a sinusoidal driving force of the form  $f_e(t) = A \sin(Ft)$ , where  $A$  and  $F$  are the amplitude and frequency of this force, respectively.

2. We will modify the view so that it displays a phase-space diagram of the system, that is, a plot of angular position versus angular velocity.
3. We will modify both the model and the view to compute and plot the potential and kinetic energies of the system and their sum.
4. We will show how to modify the introduction pages to update the description of the simulation.

### Modifying the Model

We need to revisit the different subpanels for the model and make the necessary changes to each of them.

#### *Adding New Variables*

The introduction of friction and an external driving force requires adding new variables to the model. We do this by creating a second table of variables. Although we could add the new variables to the existing table, it is sometimes preferable, for clarity, to organize the variables into separate tables. For this, we select the **Variables** subpanel of the **Model** panel of *Ejs* and right-click on the upper tab of the existing page. A popup menu will appear as shown in Figure 17.16.

From this menu, we select the **Add a new page** option (the one highlighted in the figure), and *Ejs* will create a page with an empty table of variables. (Before creating it, though, *Ejs* will ask you for the name of this new page. You can choose, for instance, **Damping, forcing, and energy**.)

In this new table we can type all the new variables that help us extend the model in the prescribed way. The mechanism to add a variable is simple. We just need to type a name for the variable in the column **Name**, select one of the possible types in the **Type** column, and, optionally, provide an initial value in the **Value** column. (The column labeled **Dimension** is used to declare arrays, and we won't use it for this model.)

Double-click a cell in the table in order to edit that cell. Use the tab key or arrow keys to move from cell to cell. *Ejs* will automatically add rows as needed. In this way, create new variables of type **double** called **b**, **amplitude**, **frequency**, **potentialEnergy**, **kineticEnergy**, and **totalEnergy**. Assign the first three variables the initial values of 0.1, 0.0, and 2.0, respectively. The energy variables will be initialized as result of the evaluation of constraints. The final new

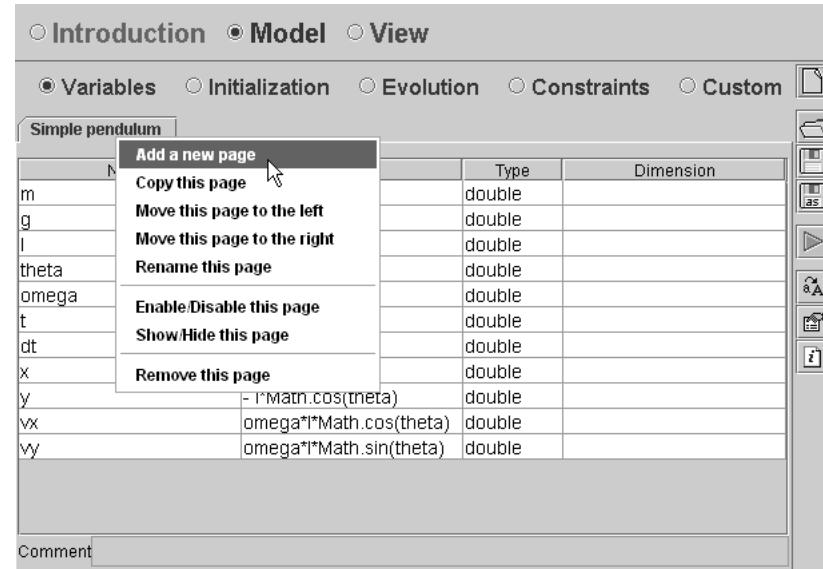


FIGURE 17.16 The popup menu for a page of variables.

table of variables is displayed in Figure 17.17. Note that *Ejs* will add an empty row to the end of the table. This can be deleted by right-clicking on the row and selecting **Remove this variable**.

Simple pendulum Damping, forcing and energy			
Name	Value	Type	Dimension
b	0.1	double	
amplitude	0.0	double	
frequency	2.0	double	
potentialEnergy		double	
kineticEnergy		double	
totalEnergy		double	

FIGURE 17.17 The new table of variables for our simulation.

## 17.5 Modifying a Simulation

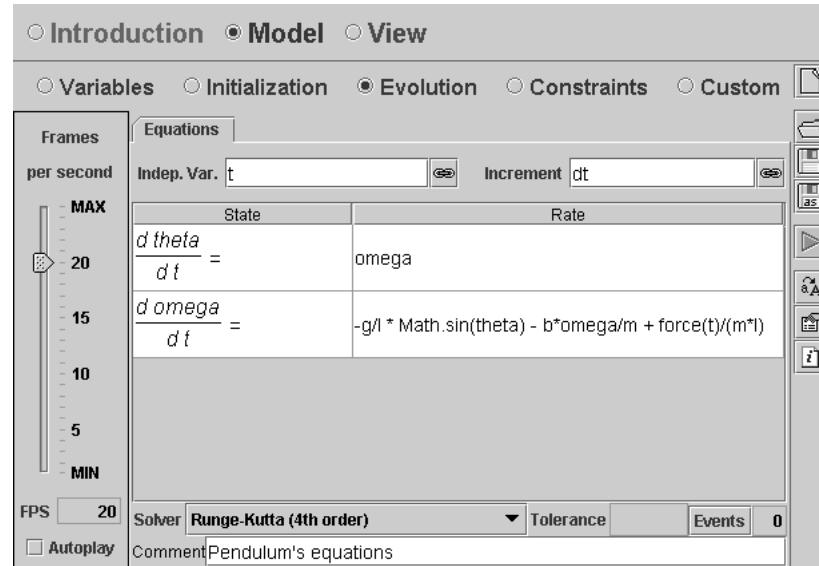
389

*Modifying the Evolution*

We need to edit the differential equations for the system to add the new forces. For this, go to the Evolution subpanel and edit the right-hand side of the second differential equation so that it reads:

```
-g/l * Math.sin(theta) - b*omega/m + force(t)/(m*l)
```

The result is shown in Figure 17.18. Notice that we are using in this expression the method `force(t)` that is not yet defined. We will need to create it as a user-defined method, when we get to the Custom subpanel.



**FIGURE 17.18** The edited differential equations.

*Computing the Energy*

Select the Constraints subpanel and follow a procedure similar to what we did for the table of variables to create a second page of constraints. Call the new page Energies and, in the blank editor that appears, type the following code:

```
potentialEnergy = m*g*(y+1);
kineticEnergy = 0.5*m*(vx*vx+vy*vy);
totalEnergy = potentialEnergy + kineticEnergy;
```

(This sets the potential energy to zero when the pendulum is hanging straight down.) The reasons to compute the energies in a page of constraints (instead of in the evolution) are the same as those for the computation of the cartesian magnitudes explained in Section 17.3. Any expression that we specify as constraints will

be evaluated every time the state of the system changes. Thus, the corresponding relationships (the value of the energy variables) are always kept up to date.

### *Coding the external force*

To finish our changes to the model, we need to specify the expression for the external force. We do this using a page of *Custom code*. Move to this subpanel and click in the empty work area to create a new page called *External force*. The new page that appears looks very much like the other editors of code that we have used previously. But there is an important difference.

Since custom code is not automatically used by *Ejs*, the code we write here is not wrapped into an internal Java method, but must be explicitly defined as a valid Java method (and later invoked in your code). For this reason, type in the editor exactly the following:

```
public double force (double time) {
    return amplitude * Math.sin(frequency*time);
}
```

This correctly defines the custom method, and concludes our changes to the model.

### **Modifying the View**

Let us start the changes to the view of the simulation by including the phase-space graph of angular velocity versus angular position. For this, go to the *View* panel and, from the right-hand side collection of elements offered by *Ejs*, click on the icon for a *PlottingPanel*, .

When you click on it, the icon will be highlighted, and the cursor will change to a magic wand, . With this wand, go to the left-hand side frame of the view, and, in the tree of elements, click on the element called *Dialog*. You are then asking *Ejs* to create a new plotting panel as child of the *Dialog* window. This is the simple mechanism used to add new elements to the view of the simulation. Figure 17.19 illustrates this action.

When you do this (and after providing a name for the new element), a new plotting panel will appear in the dialog window, sharing the available space with the previous plotting panel. Because the *Dialog* window is too small to host both panels, we need to enlarge it. Get rid of the magic wand by clicking on any blank area of the left frame. Double-click the *Dialog* element and set its *Size* property to 372,600. You can also edit the properties of the new plotting panel to customize its title and axis labels by double-clicking on the new plotting panel element that you created.

Now, let's add a *Trace* element  to the new plotting panel. A trace is an element that can display a graph consisting of a sequence of points. Follow the creation process described above (again using the magic wand), name the new element *PhaseSpace*, and edit the table of properties so that it looks like Figure 17.20. These properties simply instruct the element to add to the graph a new

## 17.5 Modifying a Simulation

391

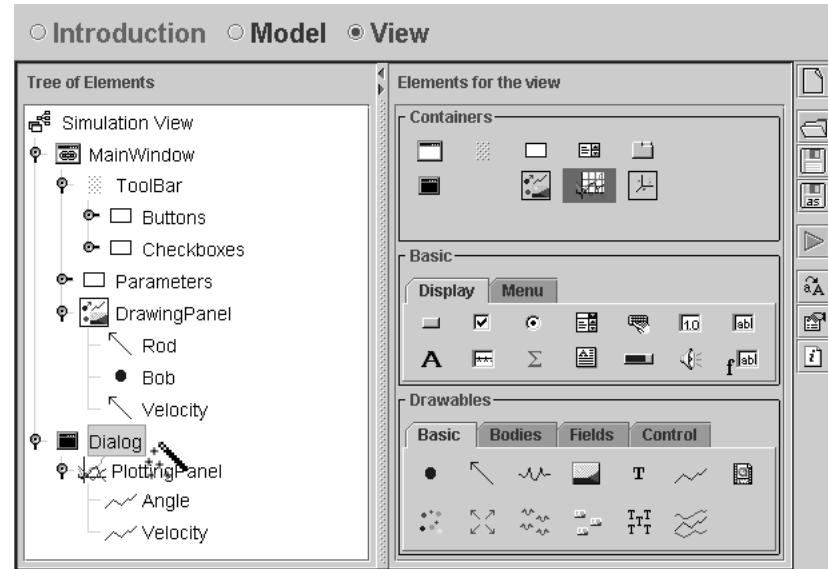


FIGURE 17.19 Adding a new plotting panel element to the Dialog window.

point  $(\theta, \omega)$  after each evolution step, displaying only the last 300 added points, and connecting them with a blue line.

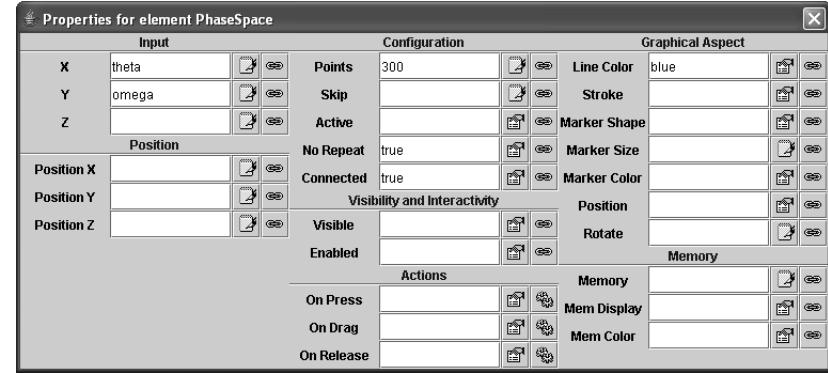
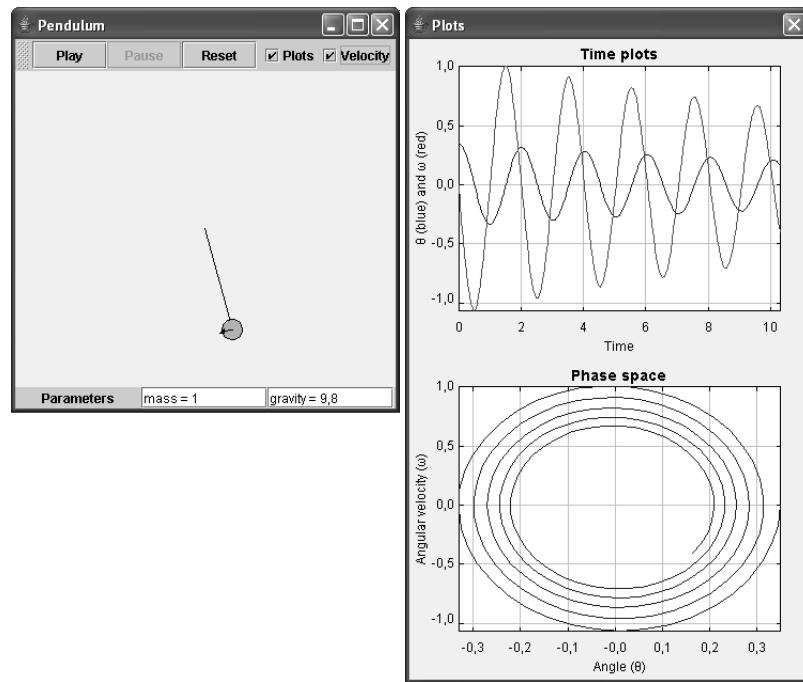


FIGURE 17.20 Properties for the PhaseSpace trace element.

The changes we did so far illustrate very well how to add and customize new elements to the view. It is as simple as it looks. Just use the magic wand to add new elements, and edit their properties to match your needs. In most cases, you will use some of the variables of the model for the properties of the view element. The connection between model and view is then automatically handled by *Easy Java Simulations*.

If we now run the simulation, we would obtain something like Figure 17.21.



**FIGURE 17.21** The simulation displaying both time and phase space plots.

We can now proceed similarly to add time plots of the energies of the system. We leave it to you as an exercise to create a third plotting panel as a child of the Dialog window and three traces (with different colors) in this panel that will plot the potential, kinetic and total energies of the system.

We will end the changes to the view by adding new fields for the user to visualize and edit the values of the variables `b`, `amplitude` and `frequency`. Click on the icon for view elements of type `NumberField`, , and add three of these to the view element called `Parameters`. Give the three fields the same name as the corresponding variables, that is, `b`, `amplitude`, and `frequency`. (Names of elements must be unique in the view, but they don't clash with the name of model variables.)

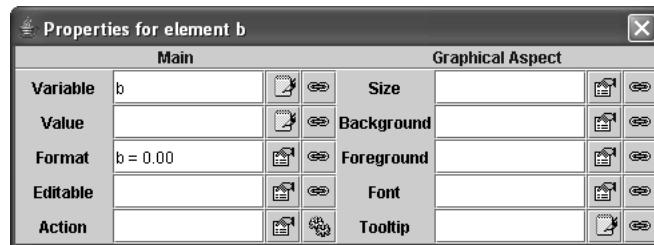
`Parameters` is a basic Swing<sup>7</sup> panel that has been configured (using its `Layout` property) to display its children using a grid with one single row. When we add the new three fields, the six children of the panel will look pretty small. To solve this, change the `Layout` property of `Parameters` to `grid:2,3`. This will organize the children in two rows of three elements each.

<sup>7</sup>Swing is the standard Java library for graphical components such as panels, buttons, and labels.

## 17.5 Modifying a Simulation

393

Now, we need to edit the table of properties of each of the field elements so that they display the corresponding variables. We show how to do this for the first element. Double-click the element b and edit its properties as shown in Figure 17.22.



**FIGURE 17.22** Properties for the b field element.

The association of the property **Variable** with the variable b of the model tells the element that the value displayed or edited in this field is that of b.

To facilitate the association of properties with variables of the model, you can use the icon that appears to the right of the text field for the corresponding property. If you click on this icon, a list of all the variables of the model that can be associated to this property will be offered to you. You can then comfortably select the one you want with the mouse.

Also, to help edit some of the technical properties (such as layouts, colors, and fonts, for instance), *Ejs* offers dedicated editors that can be accessed clicking on the icon , when shown.

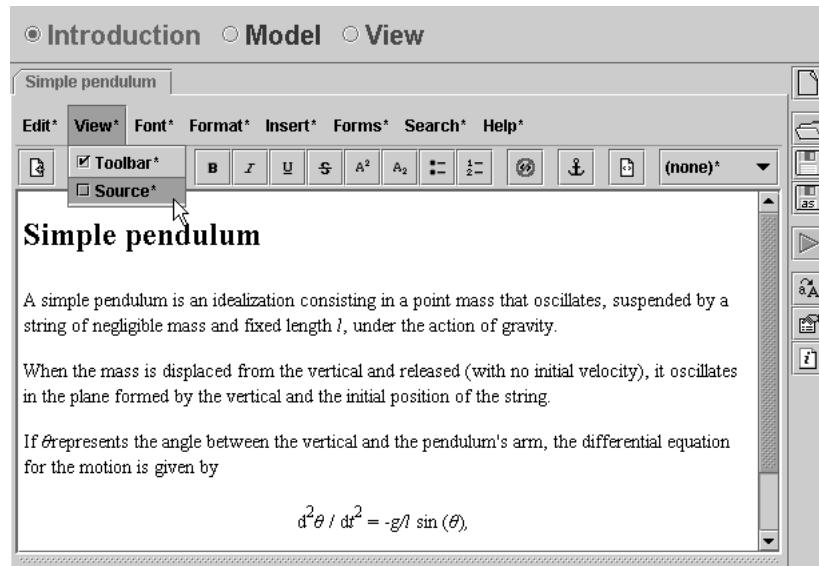
The property called **Format**, to which we assigned the value  $b = 0.00$ , has a special meaning. It doesn't indicate that b should take the value of 0, but is interpreted by the element as an instruction to display the value of b with the prefix "b =" and with two decimal digits.

This completes our changes to the view.

### Modifying the Introduction

We now want to modify the introduction to reflect the new situation. Select the **Introduction** panel of *Ejs* and right-click on the tab of the existing page to bring it in its popup menu. From this menu, select the **Edit/View this page** option. This will activate the edit mode for the displayed html page. See Figure 17.23.

HTML pages are text pages that include special instructions or tags that allow web browsers to give a nice format to the text, as well as to include several types of multimedia elements. The editor allows you (when in edit mode) to work in a WYSIWYG (what you see is what you get) mode. However, if you are familiar with html and want to work directly on the code (which, sometimes, is preferable), you can select the option highlighted in Figure 17.23 to access directly the html source for the page.



**FIGURE 17.23** Edit mode of the html editor for the introduction page.

You can write as many introduction pages as you want. As we saw earlier, each of the pages will turn into a link in the main html page that *Ejs* generates for the simulation. To return to View mode, right-click the tab of the existing page and select again Edit/View this page. In this way, you toggle the edit/view mode.

### An Improved Laboratory

Our new simulation is finished. We need now to save it to disk. Because we don't want to lose our original simulation, click on the Save As icon of *Ejs*'s taskbar, and a file dialog will allow you to save the simulation to a new file.

This simulation allows you to explore the behavior of a simple pendulum playing with different possible values of the parameters. Running the simulation can produce situations such as that displayed in Figure 17.1, at the beginning of this Chapter.

You will find the complete improved simulation in the `_ospguide` subdirectory of your **Simulations** directory with the name **PendulumComplete.xml**.

## 17.6 ■ A GLOBAL VISION

We end this chapter with an overview of what we did. We learned that *Easy Java Simulations* is an authoring tool that provides a simplified way to design and build complete interactive simulations in Java. For this, it structures a simulation into three main parts, and provides, for each of these parts, a specialized editor that helps you implement them using high-level access to many of the OSP classes

and utilities.

To investigate in more detail how each of these editors work, we loaded an existing simulation and inspected all of the panels and subpanels of *Ejs* in turn. This helped us understand how the different parts of the simulation are specified and how all the pieces fit together.

We also learned how to run and distribute the simulation either using physical media or through the Internet. Finally, we modified the simulation in order to learn some of the operating procedures of *Easy Java Simulations*.

The three parts of a simulation are the introduction, the model and the view. Each of these parts has its function in the simulation and its own panel in *Ejs*' interface, each with its own look and feel.

The introduction offers an editor for the html pages required to create the multimedia narrative that introduces the simulation. This editor allows us to edit this narrative, either working in WYSIWYG mode or writing directly the html code.

The model is the engine of the simulation and is created using a sequence of subpanels where we specify the different parts of it: definition of variables, initialization, evolution of the system, constraints (or relationships among variables) and custom methods. Each panel provides editing tools that facilitate the job of creation (including sophisticated tasks such as solving differential equations and treatment of events).

Finally, the view contains a set of predefined elements, based on Swing components and OSP graphics classes, that can be used as individual building blocks to construct a structure in form of a tree for the interface of our simulation. These elements, that can be added to a view through a simple procedure of click and create (our magic wand), have in turn a set of properties that indicate how each element looks and behaves. These properties, when associated to variables from the model (or Java expressions that use them), turn the simulation into a true dynamic and interactive visualization of the phenomenon under study.

And that's it! Though the chapter is long because we accompanied the description with details and instructions, the process can be summarized in the few paragraphs above. Obviously, learning to manipulate the interface of *Easy Java Simulations* with fluency requires a bit of practice, as well as the familiarization with all the possibilities that exist. In particular, with respect to the creation of the view, you'll need to learn the many types of elements offered and what each of them can do for you.

If you want to know more about *Easy Java Simulations* or want to see more examples of simulations created with *Ejs*, you are cordially invited to read the manual found in the companion CDROM and to visit *Ejs*'s home page at <http://fem.um.es/Ejs>.

## CHAPTER

## 18

## Dissemination And Databases

©2005 by Anne J. Cox and William Junkin, August 2005

We describe technologies, such as the BQ-OSP database and ComPADRE, which allow OSP-based materials to be disseminated across the internet and show how these technologies can be used to adopt, adapt, and run OSP-based curricular materials.

### 18.1 ■ OVERVIEW

With the resources available from OSP developers, including Ejs (Chapter 17) and Tracker (Chapter 16), and the ease of using LauncherBuilder to author curricular material (Chapter 15), there is one final aspect to address: dissemination. As the OSP user community grows and curricular materials are developed, some instructors will want to take existing simulations, change initial conditions, or customize the tutorials and guided activities based on these simulations.<sup>1,2,3,4</sup> This Chapter describes how we use internet technologies to enable others to use the simulations and curricular materials that we have written and how you can add your material to this collection.

### 18.2 ■ DISSEMINATING PROGRAMS USING THE INTERNET

You have already seen that we can distribute material as a single double-clickable jar file using Launcher packages such as `osp_demo.jar`. This material can then be distributed on a CD or downloaded from a web site and then run on the user's computer as an application. But there are other ways to disseminate material over the internet. Two common technologies are Java Web Start which installs a program on a local computer or Java applets which execute a program from within a web browser. Both methods impose security restrictions which are necessary precautions to protect the local computer from malicious software.

<sup>1</sup>Consider curricular innovations such as Peer Instruction, Just-in-Time Teaching, and Physlets that are widely used in the physics community not only because they are based on sound pedagogical practices, but are easily adaptable to a variety of instructional settings. It is this adaptability that is crucial and that the BQ-OSP Database is designed to facilitate.

<sup>2</sup>E. Mazur, *Peer Instruction: A User's Manual*, Prentice Hall (1997).

<sup>3</sup>G. Novak, et al., *Just-in-Time Teaching: Blending Active Learning with Web Technology*, Prentice Hall (1999).

<sup>4</sup>W. Christian and M. Belloni, *Physlets: Teaching Physics with Interactive Curricular Material*, Prentice Hall (2001) or see <http://webphysics.davidson.edu/Applets/Applets.html>.

### Java Web Start

Java Web Start allows for the one-click delivery (downloading) of standalone applications from the web. Java Web Start requires that a `jnlp` web page be posted on a server. This page is similar to an html page except that a user accessing the page is prompted by a security dialog box, as shown in Figure 18.1, and Java Web Start then copies a jar file and other resources into a cache on the local computer. After downloading, a Java Web Start application runs from the desktop just like other Java applications but with restricted access to the operating system. Several



**FIGURE 18.1** A standard security dialog box regarding the execution of a program from within a web browser.

examples can be found and run from the OSP Java Web Start page:

<http://www.opensourcephysics.org/webstart>

Because Java Web Start automatically checks the server for newer versions of a program, using it ensures that the most current version of the application will always be deployed. Detailed information about Java Web Start is available at:

<http://java.sun.com/products/javawebstart/>

### Applets



**FIGURE 18.2** A button titled “Launcher” which when pressed uses `ApplicationApplet` to execute a Launcher package.

An applet is a Java program that runs within a web browser. Because applications and applets have different security requirements and because they use different startup methods, a Java program must be modified to run as an applet. The

**ApplicationApplet** is an applet that runs other OSP programs in a web browser without modification. The applet itself consists of a single button, as shown in Figure 18.2, which when pressed, creates (instantiates) the program. This applet can be used to run the `osp_demo.jar` Launcher package by embedding the following applet tag in an html page:

```

1 <applet codebase="jars"
2   code="org.opensourcephysics.davidson.applets.
3     ApplicationApplet.class"
4   archive="osp_demo.jar" name="demo"
5   width="250" height="50">
6   <param name="target"
7     value="org.opensourcephysics.tools.Launcher">
8 </applet>
```

The html page now contains a button which runs Launcher and reads `launcher_default.xset` packaged in the jar.

**ApplicationApplet** supports the following parameter tags:

**htmldata** displays an html page that provides the questions or instructions for the animation user (provides the narrative).

**xmldata** reads an xml file with parameters for the particular application (provides the initial OSP Control parameters) that over-rides the program's default parameters.

**target** is the location of a program such as `org.opensourcephysics.davidson.-nbody.PlanarNBodyApp`.

The **xmldata** parameter tag enables us to pass data to a program running within **ApplicationApplet**. The html page again contains of a button that runs a program.

```

1 <applet codebase="jars"
2   code="org.opensourcephysics.davidson.applets.
3     ApplicationApplet.class"
4   archive="osp_demo.jar" name="superposition"
5   width="230" height="50">
6   <param name="target" value="org.opensourcephysics.
7     davidson.qm.QMSuperpositionExpectationXApp">
8   <param name="xmldata" value="..../sho_3_4x.xml">
9 </applet>
```

Compare this applet tag with the one that runs the entire Launcher package. The **target** parameter signals **ApplicationApplet** to run **QMSuperpositionExpectationXApp** from within `osp_demo.jar`. The optional **xmldata** parameter loads previously saved parameters from the `..../sho_3_4x.xml` data file.

Finally, it will be possible with the introduction of Java 1.6 (expected release in late 2006) to script a Java applet in an html page with a JavaScript interpreter

## 18.3 The BQ-OSP Database

399

*built into* the Java VM.<sup>5</sup> The current version of Java-to-JavaScript communication (LiveConnect) is run by the web browser and therefore its functionality is dependent on the operating system, the browser, and the browser version. The OSP API that supports this scripting model will be released in 2006 but it is likely that we will add a `script` parameter tag to `ApplicationApplet` that will load a text file containing JavaScript code.

### 18.3 ■ THE BQ-OSP DATABASE

Now that we have seen a variety of ways OSP curricular material can be distributed across the internet, we focus on distribution mechanisms.

The BQ-OSP Database provides our primary mechanism of dissemination for OSP-based curricular materials, which can be used with or without modification. It is located at

<http://www.bqlearning.org>

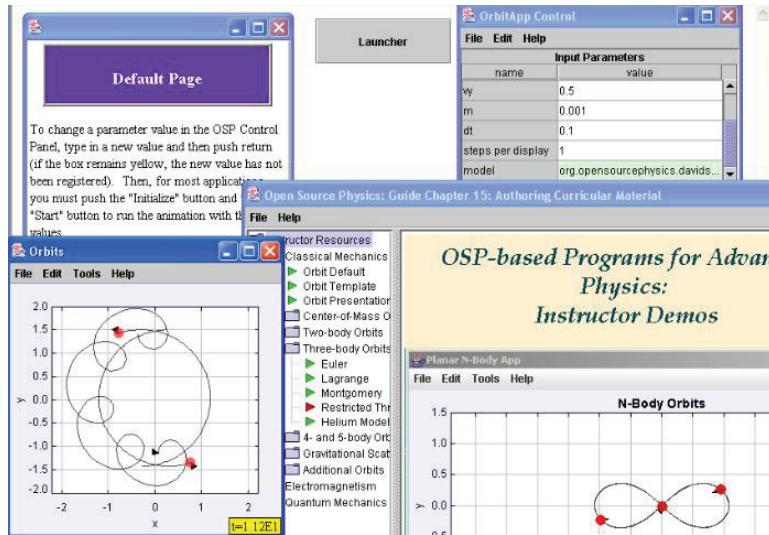
This database<sup>6</sup> simplifies the customization of OSP simulations and curricular material, and enables broad OSP collaboration and dissemination. It accomplishes this with the following features:

- content that is searchable by topic and by keyword using the database search or through ComPADRE and other digital libraries.
- narratives that can be customized and associated with one or more simulations to create guided tutorials.
- simulations can be customized by editable xml files that are then saved in the database.
- links to these materials that can easily be embedded in local web pages and presentations for students to access.
- local (desktop) installation and content that can be exported from and imported to this local database.

For example, the curricular material distributed with the Launcher package described in Chapter 15 can be found in the database by entering the word “Launcher” the search text box. When you select the Launcher package associated with Chapter 15, you will find an html page with an button like that of Figure 18.2. When you click this button, the Launcher will open, as shown in Figure 18.3. To deliver this to students, you simply give them the URL.

<sup>5</sup>This interpreter is based on the Mozilla Rhino interpreter.

<sup>6</sup>The BQ-OSP Database is one part of a collection of instructional resources (BQ Program Suite) to support the use of Peer Instruction, Just-in-Time Teaching and Physlets. Instructors interested in using other aspects of the BQ Program Suite are encouraged to go to <http://www.bqlearning.org> for a demonstration.



**FIGURE 18.3** Launcher and text window “Default Page” that opens when you press the “Launcher” (applicationApplet) button from an html page delivered by the database.

However, you may wish to find and use other available curricular materials, edit materials submitted by others (e.g. the “Default Page” in Figure 18.3), or upload your own customized OSP-based curriculum materials. The Sections that follow will describe how to edit and search for existing materials as well as how to add resources to the database. The goal of the database is to make it easy for others to use or modify resources even if they are not programmers or code developers.

We will begin by describing how to find a single simulation (Sections 18.4 and 18.5), how to customize it (Sections 18.6 and 18.7) and then how to upload your curricular materials including an xml file and a narrative.

## 18.4 ■ SEARCHING AND BROWSING

Before developing new curricular materials, it is worthwhile to investigate curricular materials that already exist. There are two primary ways of finding materials in the database: searching and browsing by topic. Begin at the BQ-OSP homepage (follow the link to the database from <http://www.bqlearning.org>) and click on the “Search” tab (Figure 18.4).

For example, suppose you want to find a three-body orbits, such as the one from Section 15.5. You can “browse” by clicking on the topic: “Classical Mechanics” under “Upper Level Physics.” Notice that for each subcategory (Figure 18.5), you get a list of available resources for each type of curricular materials (Physlets, Open Source (OSP), EJS, and Tracker).

In this case, because you want to find the OSP materials from Section 15.5 on three-body orbits, you would click on the number under the “Open source”

## 18.4 Searching and Browsing

401

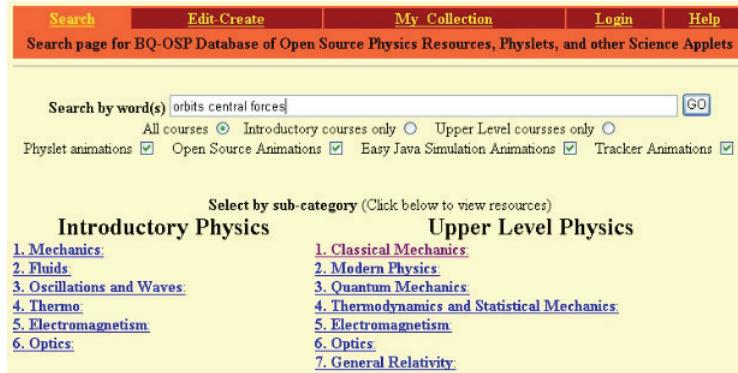


FIGURE 18.4 Search field available within the database.

Select by sub-category (Click below to view resources)		physlet	osp	ejs	tracker	Upper Level Physics
<b>Introductory Physics</b>						<b>1. Classical Mechanics:</b>
<a href="#">1. Mechanics</a>						<a href="#">2. Modern Physics</a>
<a href="#">2. Fluids</a>						<a href="#">3. Quantum Mechanics</a>
<a href="#">3. Oscillations and Waves</a>						<a href="#">4. Thermodynamics and Statistical Mechanics</a>
<a href="#">4. Thermo</a>						<a href="#">5. Electromagnetism</a>
<a href="#">5. Electromagnetism</a>						<a href="#">6. Optics</a>
<a href="#">6. Optics</a>						<a href="#">7. General Relativity</a>
A. Mathematics						1. Classical Mechanics
10. Vector Calculus						(2) (2) (0) (0)
20. Complex Plane						(0) (1) (0) (0)
B. Newtonian Mechanics						2. Modern Physics
10. Conservative Forces						(1) (2) (1) (0)
20. Non-conservative Forces						(2) (0) (3) (2)
C. Oscillations						3. Quantum Mechanics
10. Linear Oscillations						4. Thermodynamics and Statistical Mechanics
30. Chaotic Systems						5. Electromagnetism
D. Central Forces						6. Optics
10. Orbits						7. General Relativity
20. Stability and Multi-body Orbits						
Orbits						
40. Scattering						

FIGURE 18.5 List of materials found while browsing the database by topic.

heading in the subcategory: “Stability and Multi-Body Orbits” in the “Central Forces” category. There you will see a list of resources along with thumbnails of each (Figure 18.6).

Clicking on the title or the thumbnail, opens up a separate window that loads the `ApplicationApplet`, allowing you to run the animation and read the text.

Alternatively, you can find these resources from the “Search” page by typing text into the search textbox. Searching supports a search of keywords, description, title, and topic. Searches can also be limited to a particular type of curricular material (OSP, EJS, Tracker and/or Physlets) and a content level (see Figure 18.4). If you type in the words: “orbits central force” and limit the search to “Upper Level” courses and “Open Source Animations,” you will find the same three-body orbit materials.

If you want to use the resource as is, simply click on the “Use” button and you will be given the URL of the resource. If you are logged into the database, you also have the option to add the link to your course web page stored in the database. In order to customize resources or add your own resources, you are required to

Title	Description	Keywords	Thumbnail
<a href="#">Lagrange's Three Body Solution</a>	Objects of equal mass in a Lagrange solution of the three body problem	Orbits Gravity Central Forces	

**FIGURE 18.6** Materials returned in an example search of the database. Clicking on the thumbnail image or the title opens a window for running the animation and using or editing it.

log into the database as well.

To login, simply click on the “Login” page and complete the required fields (if you are the first one from your institution to login, you will also need to complete the information in that field). Please note that none of the information collected will be shared with anyone except those running this database and ComPADRE (a National Digital Library Project for the physics and astronomy communities; for more on ComPADRE see Section 18.5), which gathers material from this database into its collection.

## 18.5 ■ INTEGRATION WITH DIGITAL LIBRARIES

The BQ Database is searchable from beyond the database itself through ComPADRE (Communities for Physics and Astronomy Digital Resources in Education: <http://www.compadre.org>). The goal of ComPADRE, a joint project of several American physics and astronomy societies, is to organize digital collections of education materials in physics and astronomy. One of ComPADRE’s services is a federated, joint search of a number of digital collections. The database is a part of this federated search (available by putting a check mark in the “Physlet/OSP Database” checkbox). If a ComPADRE user chooses to include the database in a federated search, the database provides URL links to materials that match the search fields. These are links to a Web Start page or to an html page that contains an applet. The database is designed specifically for organizing and storing Open Source Physics materials, but through the ComPADRE federated search, it provides an easy way to disseminate new animations and make them available to the wider physics community. An instructor can simply submit materials directly through any number of national digital libraries without using the database. However, the advantages of the database are that it is specifically designed to support Open Source Physics resource dissemination,

**18.7 Editing Animation by Changing XML****403**

already has ComPADRE protocols built in, and is part of a federated search. An advantage of a federated search with other digital libraries is that other related materials, such as lesson plans, textbooks, and research, can be discovered at the same time as the OSP materials. This puts the simulations in a broader context, but still links the instructor to the features and tools of the database program.

**18.6 ■ EDITING CURRICULAR MATERIAL**

Although you may find resources that fit the instructional needs for your course, at times you or your colleagues may want to customize the materials. As a user of this *Guide*, you will likely be able to customize resources fairly easily, but others may not. Thus, another goal of the database is to lower the technological barrier for other instructors and users of OSP resources. Typical resources in the BQ Database have three parts:

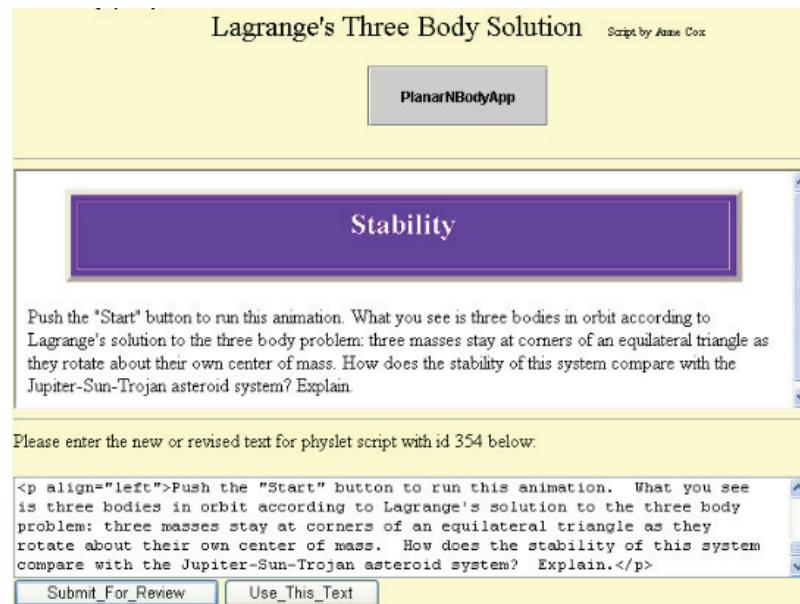
1. narrative (or text) that provides the questions students are expected to answer or a tutorial designed to help students use the simulation to understand a given physics concept.
2. initial conditions of the simulation (parameter set in OSP Control panel) set by the parameters defined in an xml file.
3. the simulation itself.

You may decide to use an animation as it exists and only change the question asked of the students (turn it into a multiple choice question, give more guidance in solving the problem, asking a different question, etc.). To change the text or narrative associated with the exercise, log in (see Section 18.4) and find the animation of interest (such as the three-body example from above). Now click on the “Edit” button next to the text (if you are not logged in, you will not see an “Edit” button). You can then type in new text, edit the existing text, or upload a new html page (see Figure 18.7). Once loaded, when someone else chooses to “Use” this text, the ApplicationApplet will call this new html page from the parameter tag `htmlData` when the applet is loaded.

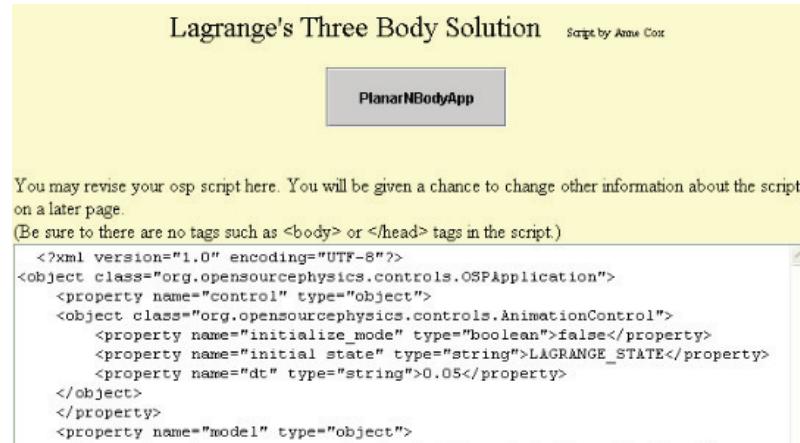
Anyone who simply edits the text associated with the animation does not need to know about these parameters. To increase the ease of use, the database program automatically takes care of these details and fills in the proper values for the parameters. After editing a text, you may make these available for others to see (public) or save it to your own private collection so that only you and your students may view it (private).

**18.7 ■ EDITING ANIMATION BY CHANGING XML**

You may find an animation that you want to use, but you would prefer to deliver it to your students with different initial conditions as set by the OSP Control

**FIGURE 18.7** Editing the narrative.

parameters. In this case, you need to change the xml file. Choosing the button after “Edit this script” above the button, opens up a screen that shows the xml file in a frame (Figure 18.8). Alternatively, you can also upload an xml file directly from your local computer.

**FIGURE 18.8** Xml editing mode.

After changing xml parameters, a click on the applet’s button shows the ani-

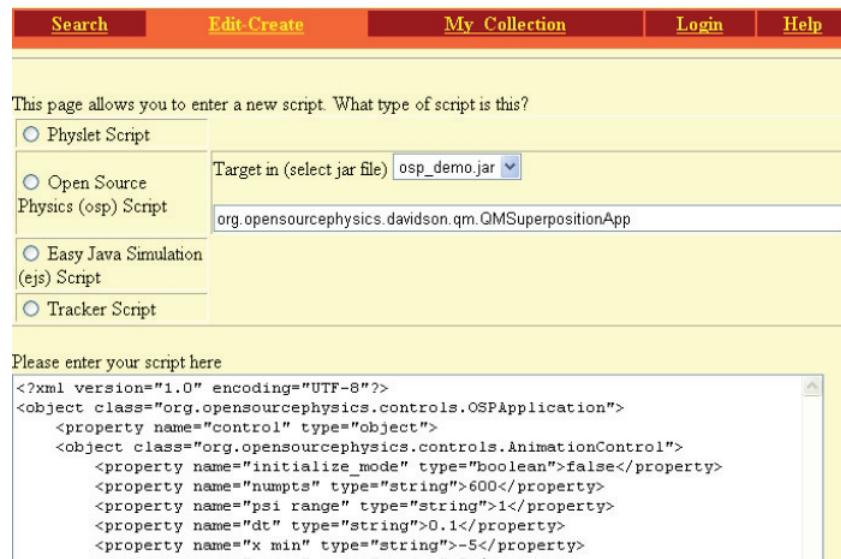
## 18.8 Uploading New Materials

405

mation with the changes. When you have made all the changes you wish to the parameter set, you simply save it and it is ready for delivery to your students. You may choose to save it to your private collection or to all users of the database (public). You will also be asked to choose a topic, category and sub-category for the animation (from a pull-down list). In order to upload parameter sets from multiple files when using Launcher, see the following Section.

## 18.8 ■ UPLOADING NEW MATERIALS

Instructors are encouraged to submit new curricular material to the BQ Database. Some new materials will be based on the original OSP library and the jar files already provided in the database (`osp_demo.jar`, `osp_guide.jar`, etc.). The instructor will provide the target (the application class file in the OSP library), the xml data file (initial parameters for the OSP Control panel) and the html data file (narrative). Consider the `quantum_mechanics.xml` file that you used with the Launcher in Section 15.3, shown in Figure 15.8. To upload just that animation (not the entire Launcher package) and its associated xml file (you saved this as `test.xml` in Section 15.3), choose the “Edit/Create” tab, and then the “Upload New Materials” button to get the page shown in Figure 18.9.



**FIGURE 18.9** Page for uploading new resources into the database.

Select the appropriate jar file, complete the target information: `osp.opensourcephysics.davidson.qm.QMSuperpositionApp` and then either copy in

the xml file (shown in Figure 18.9) or use the browse button to upload the associated xml (`test.xml`) file. You can check that the animation runs, and then save it to the database. To save it, you will need to select a category (from the pull-down list) and give it a title. You can then choose whether it will be part of the public collection (for anyone to see) or your private selection (for you and your students only), pick keywords (from a list maintained by the database administration) and have the option of providing a brief description. Once it has been submitted, you will have the option of providing a thumbnail of a screen shot of the animation and a narrative (an associated html page) for the animation. Providing these data makes it easier for other instructors to find and use the simulations and will support wider dissemination of the resources.

Submitting curricular materials designed using Launcher works in much the same way. In that case, the target information is `org.opensourcephysics-tools.Launcher` and the xset, xml, and html files must be provided in a single zip file which must be named `launcher_default.zip` as described in Section 15.3.

If you wish to submit an animation based on your own jar file, the database can store signed jar files that are not too big. The current limitation is 3 MB, but this size limitation is likely to change over time. If the jar file is too big to be stored in the database, the jar file must be maintained elsewhere and its URL is stored in the database. The BQ Database program will use that URL to obtain the jar file and provide it correctly when the animation is run. Of course, this means that these animations can only be run on computers that have internet access.

While the database will store curricular material and most jar files, it will not store the sources files for any jar files. Instead the database will provide a link to the source files along with a record of the date the source files were last modified and the source file size. Thus, the database will provide a warning to other users if the source files (associated with a jar of interest) have been changed.

## 18.9 ■ DATABASE INSTALLATION

Although the BQ Database is available on-line, it can also be run from a local computer, with the local computer serving out the web pages. A CD distribution of the database is available, as is a zip file of the CD contents from <http://www.bqlearning.org>. The installation instructions are in the `DB_Instructions.doc` file. On machines running the Windows operating system, installing the database on a local computer requires creating the folder `C:/Util` and then running the required batch file (`InstallBQdbCDrive.bat`). This will install the database program, MySQL, and Apache, which allows your local computer to serve out web pages as well as the database and the rest of the BQ Program Suite.

As you generate curricular material based on OSP resources, consider adding them to the on-line BQ Database as part of the public collection. The goal of the OSP Project is to share resources not only with the community of OSP developers

18.9 Database Installation

**407**

that this guide serves, but to the wider physics instructional community.