



2D游戏开发的优秀教程
内容全面，示例实用，覆盖必备核心技能



Java 2D 游戏 编程入门

FUNDAMENTAL 2D GAME
PROGRAMMING WITH JAVA

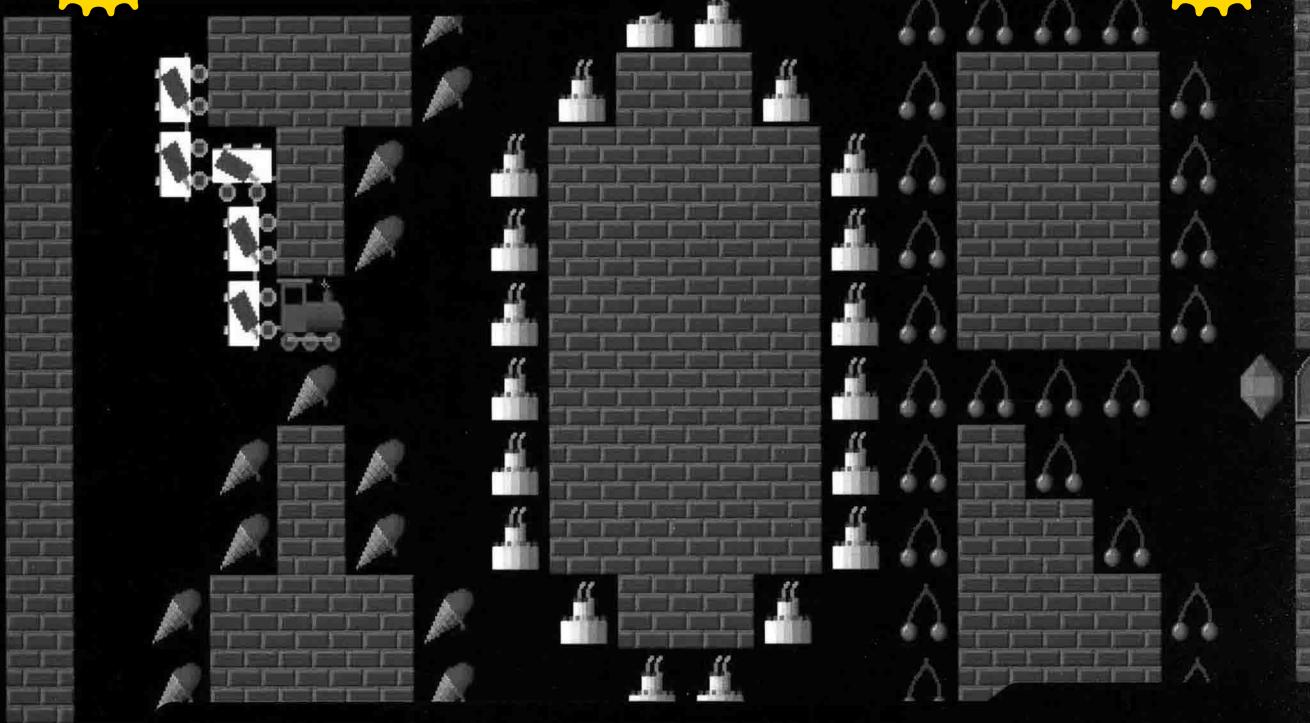
[美] Timothy Wright 著 李强 裴云 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS



2D游戏开发的优秀教程
内容全面，示例实用，覆盖必备核心技能



Java 2D 游戏 编程入门

FUNDAMENTAL 2D GAME
PROGRAMMING WITH JAVA

[美] Timothy Wright 著 李强 裴云 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS



图书在版编目 (C I P) 数据

Java 2D游戏编程入门 / (美) 莱特 (Wright, T.) 著;
李强, 裴云译. -- 北京 : 人民邮电出版社, 2015.7
ISBN 978-7-115-39277-0

I. ①J... II. ①莱... ②李... ③裴... III. ①游戏—
JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2015)第106147号

版权声明

Fundamental 2D Game Programming with Java

Timothy Wright

Copyright © 2015 Course Technology, a part of Cengage Learning.

Original edition published by Cengage Learning. All Rights reserved.

本书原版由圣智学习出版公司出版。版权所有, 盗印必究。

Posts & Telecom Press is authorized by Cengage Learning to publish and distribute exclusively this simplified Chinese edition. This edition is authorized for sale in the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

本书中文简体字翻译版由圣智学习出版公司授权人民邮电出版社独家出版发行。此版本仅限在中华人民共和国境内(不包括中国香港、澳门特别行政区及中国台湾)销售。未经授权的本书出口将被视为违反版权法的行为。未经出版者预先书面许可, 不得以任何方式复制或发行本书的任何部分。

978-7-115-39277-0

Cengage Learning Asia Pte. Ltd.

151 Lorong Chuan, #02-08 New Tech Park, Singapore 556741

本书封面贴有 Cengage Learning 防伪标签, 无标签者不得销售。

◆ 著 [美] Timothy Wright
译 李 强 裴 云
责任编辑 陈冀康
责任印制 张佳莹 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京艺辉印刷有限公司印刷
◆ 开本: 800×1000 1/16
印张: 35.25
字数: 672 千字 2015 年 7 月第 1 版
印数: 1~2 500 册 2015 年 7 月北京第 1 次印刷
著作权合同登记号 图字: 01-2015-2397 号

定价: 79.00 元

读者服务热线: (010) 81055410 印装质量热线: (010) 81055316
反盗版热线: (010) 81055315



内容提要

学习 2D 游戏开发是快速积累游戏开发经验的关键。理解了 2D 环境的要素，就能够为游戏开发打下坚实的基础。

本书讲解使用 Java 进行 2D 游戏开发的基础知识和基本技能。本书共 18 章，分为基础知识、提高技能和完整游戏 3 个部分，详细介绍应用编程、全屏游戏、输入处理、矩阵变换、基础物理、相交测试、碰撞检测等知识点和技能，最后采用介绍的所有概念，从头到尾开发了一款完整的游戏。

本书适合游戏开发的初学者阅读，尤其适合想要学习 2D 游戏编程基础的人。本书假设读者理解核心编程概念、面向对象软件以及 Java 编程语言，但不需要读者具备任何游戏编程的知识。



致谢

首先，我要感谢 www.gamedev.net 的所有论坛版主。你们真诚而有帮助的评论使得我有勇气展示自己并写好本书。

特别感谢 Drew Sikora 出版了我早期的教程，并且在时日艰难的时候发来了读者的邮件，鼓励我继续前行。感谢 John Hattan 和 Jason Zink 提供的出版信息。感谢审阅过本书、帮助调试代码示例，以及修正所有拼写错误的每一个人。我还想感谢 Emi Smith 以及 Cengage 团队的其他人，感谢他们帮助编辑、出版本书，并且努力搞清楚如何用他们的 Macs 和我的 Windows 协同工作。我还要感谢我的编辑 Kezia Endsley，他容忍了我所有的傻里傻气的问题，并且总是乐于解决这些问题。我还要感谢 Dustin Clingman，他做了很好的技术编辑工作，保证了代码示例的正确性和清楚明了。

最重要的，我要感谢我的妻子 Jimmi。20 年来，她是我最好的朋友，没有她的持续帮助和信任，这一切都不可能。我爱你，我的宝贝！感谢你在本书完成之前没有约我出去。



作者简介

Timothy Wright 是一名软件工程师，目前正在从事下一款美妙的独立游戏开发。在成人生涯的早期，他是一名军队的钢琴演奏师，从那时候起，他就发现自己热爱编程。他拥有计算机科学的学士学位，并且曾经在公司的研发部门工作了近 10 年时间。在开发游戏或玩游戏以外的时间，他喜欢打扑克、做饭和自己酿造啤酒。他和妻子、两个女儿、3 只狗以及一名外国交换学生住在一起，每一周的生活都充满了戏剧性。**Timothy** 几乎在美国各地居住过，并且花了 5 年时间在意大利、德国居住和在欧洲旅行。多年来，他是 www.gamedev.net 的一名论坛版主，你偶尔可以看到他以 Glass_Knife 为名字在论坛上冒泡。



前言

多年前，当我第一次将软件开发作为专业工作的时候，有人请我编写一个 applet。那时候，我对于 Java 语言知道得并不多。在整个上学期间，我很广泛地使用 C++。我确实用 Java 编写过一些代码，但认为它太慢并且是 C++ 的没落版。

同时，我购买和阅读了很多可以接触到的游戏编程图书。我通读了一本关于人工智能的书，其中包含很多不错的示例，但它们都是用 C++ 和 DirectX 编写的。由于忙着学习 Java 以便在工作中使用，我认为将示例转换为由 Java 编写可能是学习这门语言的一种好办法。

毕竟，Java 游戏编程又会有多难呢？

很快我发现，关于用 Java 编写游戏，可以获取的信息实在太少了。我找到的那些信息，都是关注于 apple 编程的，实际上并没有介绍如何制作应用程序类型的游戏。在将示例转换为 Java 版并发现这可能并不是很难之后，我开始搜索关于这一切内容的教程。我没有找到这样的教程，所以我自己的编写了一本。

在 gamedev.net 上发布了两本教程并且从头开始用 Java 编写了软件渲染程序之后，我意识到 Java 这种语言给游戏社群提供了很多东西。在开发下一个系列教程的同时，我最终得到了很多的信息，使得编写一本书成为一种更好的解决方案。

这就是你现在所读到的书，它是一个开发者的代码工具集和说明，它的目的是帮助你加快学习 Java 游戏编程的过程。希望后面的内容可以解答所有必须的问题，并且帮助你学习重要的内容：开发游戏。

本书的组织结构

本书分为 3 个部分。第 1 部分是“基础知识”，覆盖了创建一个简单的原型游戏所需要的所有概念。第 2 部分是“提高技能”，介绍了编写一款完整的游戏所需要的其他概念。最后一个部分是“完整游戏”，采用介绍的所有概念，从头到尾开发了一款完整的游戏。内容的概览如下。

第 1 部分：基础知识

第 1 章 Hello World。本章创建了一个帧速率类，使用它来监控应用程序的速度。然后，介绍了窗口和全屏游戏循环。

第 2 章 输入。本章创建了类来监控键盘输入，以及绝对和相对鼠标输入。示例代码展示了一个游戏循环中的输入处理。



第 3 章 变换。本章介绍了向量、点和矩阵数学，以及点和向量的平移、旋转、缩放和切变；探讨了矩阵变换以及手动方法，还介绍了 Swing 的变换对象。

第 4 章 时间和空间。本章给循环添加了一个游戏时钟，以允许独立于帧速率的对象操作；还介绍了屏幕映射，允许从标准设备坐标映射任何的屏幕—空间对象。

第 5 章 简单游戏框架。本章使用前面展示的所有代码，创建了一个简单的游戏框架，以供在后面的示例中使用。

第 6 章 Vector2f 更新。本章更新了 Vector2f 类，这个类是在第 3 章中初次介绍的，它拥有进行相交测试所必需的所有线性代数代码。

第 7 章 相交测试。本章介绍了确定对象是否重叠或者是否在另一个对象之中所必需的数学和算法。点、圆、直线和矩形都被用来探讨这一话题。

第 8 章 游戏原型。本章使用目前为止给出的代码，配合简单框架，创建了一款原型游戏。尽管这还不是一款完整的游戏，但它展示了创建一款游戏所必需的所有代码的思路。

第 2 部分：提高技能

第 9 章 文件和资源。本章介绍了读取和写入 XML 文件，以及加载和保存 Java 属性文件。本章还介绍了访问类路径上的存档文件中所包含的资源。

第 10 章 图像。本章介绍了如何操作图像，从头创建图像，以及操作图像文件。本章还介绍了透明度的使用，以及用来绘制、旋转和缩放图像的不同方法。

第 11 章 文本。本章介绍了使用 Java 的 Font 库绘制文本所需的所有内容，还对输入库进行了更新，以支持录入。

第 12 章 线程。本章介绍了 Java 中可用的多线程库，并且展示了创建一个非阻塞的类、用一个阻塞类包装非阻塞的类，将多线程代码集成到一个单线程游戏循环中，以及使用一个线程库来加速资源加载。

第 13 章 声音。本章介绍了 Java 中支持声音的问题。既探讨了小的剪辑文件，也探讨了大文件的流播放，还有使用声音控件，例如平衡和音量等。本章中所开发的声音包装库，用于将声音集成到游戏循环之中，这要用到第 12 章所介绍的概念。

第 14 章 用 ANT 进行开发。本章对 ANT 构建工具提供了一个概览，并且带领你学习为游戏开发创建一个可扩展的构建脚本的过程。

第 15 章 碰撞检测。本章介绍了如何确定碰撞物体的交点，以及如何处理碰撞响应。本章通过学习很多的示例探讨了点、直线、矩形和圆。

第 3 部分：完整游戏

第 16 章 工具。本章介绍了制作完整游戏所需的工具。针对窗口游戏、全屏游戏开发了框架，还有 Swing 编辑器。工具库进行了更新，创建了一个简单的 Swing 编辑器来绘



制多边形并且将其保存为 XML。创建了一个 `sprite` 类，还开发了一个简单的粒子引擎。

第 17 章 太空火箭。本章使用所给出的所有代码，从头到尾开发了一款完整的游戏。尽管这款游戏很短并且很容易，但它很完备，带有加载屏幕、观赏模式、游戏进程、高分输入和保存以及一个部署 ANT 脚本。了解创建一款完整游戏所需的工作，将会激发你完成自己的游戏。

第 18 章 结论。本章列出了所有的代码示例，从而圆满结束了本书。本书给出了各章代码示例的列表，并且每个示例都解释了所学的内容，使你可以很容易地找到一个示例。

本书的目标读者

本书写给想要学习 2D 游戏编程基础的人。

尽管当前的图形发展非常快，但 3D 图形还只是迷宫的一块。如果一个程序员不理解游戏循环、输入处理、矩阵数学、相交测试、碰撞检测、图像渲染、线程、声音或者部署，只是 3D 图形也无法让游戏变得更好。即便 3D 渲染令人激动，但学习基础知识对于打下牢固基础来说仍很重要，并且处理任何情况都需要掌握工具。

本书假设读者理解核心编程概念、面向对象软件以及 Java 编程语言，但并不假设你需要有游戏编程的知识。

你应该已经熟悉 Java 语言了，本书并不会介绍它。

软件和版本

本书代码在 Windows 7 上、使用 Java SE 6.0 和 Java SE 7.0（编写本书时可用的最新版本）进行过测试。使用 Eclipse Helios IDE 进行代码的编译和执行。使用 1.8.2 ANT 版本进行部署。

示例源代码、声音、图像和资源文件，可以通过如下链接获取：

<http://www.indiegameprogramming.com>

也可以从本书的 Web 站点下载它们，可以使用书名或者 ISBN 来搜索：

<http://www.cengageptr.com/downloads>



本书体例

对于以下内容使用代码字体：

- Java 类；
- Java 方法；
- 变量名称；
- 数学变量和常量；
- 代码示例；
- 代码段；
- XML；
- 命令行输出。

问题和反馈

请将关于本书的任何评论、勘误或问题，发送到如下的 Email 地址：

indiegameprogramming@gmail.com

本书的 Web 站点上有相关的信息、更新、勘误和源代码，可以通过如下地址访问：

<http://www.indiegameprogramming.com>



目录

第1部分 基础知识

第1章 Hello World	2	4.3 调整视口高宽比	93
1.1 使用 FrameRate 类	2	4.4 大炮实例	100
1.2 创建 Hello World 应用程序	3	4.5 资源和延伸阅读	107
1.3 使用主动渲染	6	第5章 简单游戏框架	108
1.4 创建定制的渲染线程	8	5.1 屏幕到世界的转换	108
1.5 创建一个主动渲染的窗口	11	5.2 理解简单框架	110
1.6 修改显示模式	13	5.3 使用简单框架模板	116
1.7 全屏显示模式中的主动渲染	18	5.4 资源和延伸阅读	118
1.8 资源和延伸阅读	21	第6章 Vector2f 更新	119
第2章 输入	22	6.1 inv()	119
2.1 处理键盘输入	22	6.2 add()	119
2.2 键盘改进	26	6.3 sub()	120
2.3 处理鼠标输入	29	6.4 mul()	120
2.4 相对鼠标移动	37	6.5 div()	121
2.5 资源和延伸阅读	45	6.6 len() 和 lenSqr()	121
第3章 变换	46	6.7 norm()	122
3.1 使用 Vector2f 类	46	6.8 perp()	123
3.2 使用极坐标	54	6.9 dot()	123
3.3 理解点和向量	60	6.10 angle()	125
3.4 使用矩阵变换	62	6.11 polar()	125
3.5 行主序矩阵和列主序矩阵	63	6.12 toString()	126
3.6 理解 Matrix3x3f 类	65	6.13 资源和延伸阅读	128
3.7 仿射变换	76	第7章 相交测试	129
3.8 资源和延伸阅读	79	7.1 多边形中的点的测试	129
第4章 时间和空间	80	7.1.1 多边形特例	
4.1 计算时间增量	80	7.1.2 中的点	131
4.2 屏幕映射	86	7.1.3 多边形示例	



中的点	133
7.2 使用轴对齐边界框进行相交测试	137
7.3 使用圆形测试相交	139
圆和 AABB 的重合	140
7.4 使用分隔轴方法	147
7.5 使用线段一线段的重叠方法	149
7.6 使用矩形—矩形的重叠方法	152
7.7 优化测试	156
7.8 资源和延伸阅读	157
第 8 章 游戏原型	158
8.1 创建一个多边形包装类	158
8.2 创建一个原型小行星	166
8.3 创建一个原型编辑器	171
8.4 用原型小行星工厂 生产小行星	175
8.5 原型 Bullet 类	181
8.6 原型 Ship 类	183
8.7 编写原型游戏	188
8.8 资源和延伸阅读	195

第 2 部分 提高技能

第 9 章 文件和资源	198
9.1 理解 Java 如何处理文件和 目录	198
9.2 理解输入/输出流	201
9.3 创建 Resources.jar 文件 进行测试	206
9.4 将资源放到类路径上	208
9.5 制作资源加载工具包	211
9.6 利用 Java 属性	214
9.7 XML 文件概览	218
9.8 资源和延伸阅读	227
第 10 章 图像	228
10.1 学习 Java 中的颜色	228
10.2 了解不同的图像类型	229
10.3 执行颜色插值	235
10.4 使用 VolatileImage 提高速度	240
10.5 创建透明图像	244
10.6 使用 alpha 合成的规则	247
10.7 绘制精灵	254
第 11 章 文本	282
11.1 理解 Java 字体	282
11.2 制作绘制字符串工具	287
11.3 使用文本度量进行布局	291
11.4 支持线程安全的键盘输入	302
11.5 资源和延伸阅读	309
第 12 章 线程	310
12.1 使用线程实现 Callable 任务	310
12.2 使用线程加载文件	313
12.3 使用 FakeHardware 类测试	316
12.4 使用等待/通知方法	321
12.5 在游戏循环中使用线程	323
12.6 使用状态机	331
12.7 OneShotEvent 类	332
12.8 LoopEvent 类	334
12.9 RestartEvent 类	337
12.10 多线程事件示例	340
12.11 资源和延伸阅读	342



第 13 章	声音	343
13.1	操作声音文件	345
13.2	使用声音库的问题	347
13.3	开发阻塞音频类	351
13.4	用阻塞的 Clip 类	353
13.5	使用 AudioDataLine 类	357
13.6	BlockingDataLine 类	363
13.7	创建一个 SoundEvent 类	366
13.8	使用 OneShotEvent 类	368
13.9	使用 LoopEvent 类	369
13.10	使用 RestartEvent 类	370
13.11	添加声音控件	376
13.12	资源和延伸阅读	385
第 14 章	用 ANT 进行开发	386
14.1	安装 ANT 软件	386
14.2	理解构建脚本的格式	387
14.3	学习常见 ANT 任务	389
14.4	构建一个可扩展的 构建脚本	392
14.5	资源和延伸阅读	398
第 15 章	碰撞检测	399
15.1	带碰撞检测的弹球	399
15.2	使用直线参数方程	404
15.3	查找直线—矩形相交	405
15.4	查找圆一直线相交	412
15.5	查找直线一直线相交	418
15.6	计算一个反射向量	424
15.7	在一个多边形中弹回 一个点	428
15.8	资源和延伸阅读	436

第 3 部分

完整游戏

第 16 章	工具	438
16.1	创建一个游戏框架	438
16.2	更新多边形编辑器	450
16.3	绘制精灵	465
16.4	创建一个简单的粒子引擎	469
16.5	资源和延伸阅读	474
第 17 章	太空火箭	475
17.1	Bullet 类	476
17.2	PolygonWrapper 类	477
17.3	Particle 类	480
17.4	Asteroid 类	482
17.5	AsteroidFactory 类	485
17.6	AsteroidExplosion 类	488
17.7	Ship 类	490
17.8	ShipFactory 类	495
17.9	ShipExplosion 类	497
17.10	添加游戏常量	498
17.11	Acme 类	499
17.12	GameState 类	500
17.13	Score 类	501
17.14	QuickLooper 类	502
17.15	QuickRestart 类	503
17.16	HighScoreMgr 类	505
17.17	管理状态	509
17.18	StateController 类	510
17.19	CompleteGame 类	511
17.20	GameLoading 状态	514
17.21	AttractState 类	520
17.22	PressSpaceToPlay 模式	522
17.23	HighScore 状态	523



17.24	GameInformationState 类	524	
17.25	LevelStarting 状态	525	
17.26	LevelPlaying 状态	526	
17.27	GameOver 状态	533	
17.28	EnterHighScoreName 状态	534	
17.29	创建构建脚本	539	
17.30	资源和延伸阅读	540	
第 18 章	结论	541	
第 1 章	Hello World	541	
第 2 章	输入	542	
第 3 章	变换	542	
第 4 章	时间和空间	542	
第 5 章	简单游戏框架	543	
	第 6 章	Vector2f 更新	543
	第 7 章	相交测试	544
	第 8 章	游戏原型	544
	第 9 章	文件和资源	544
	第 10 章	图像	545
	第 11 章	文本	545
	第 12 章	线程	546
	第 13 章	声音	546
	第 14 章	用 ANT 进行开发	547
	第 15 章	碰撞检测	547
	第 16 章	工具	547
	第 17 章	太空火箭	548
		资源和延伸阅读	549



第 1 部分

基础知识



第 1 章

Hello World

大多数编程图书的第 1 章是创建一个 Hello World 应用程序，它执行一个简单的函数以确认一切都能工作。最简单的计算机游戏是一个黑色背景的窗口，还有一个紧凑的 while 循环，它负责尽可能快地清除屏幕并且重新绘制黑色的背景。添加一个帧速率计算器将可以度量应用程序的性能，并且验证窗口正在重新绘制。本书中的大多数示例都会测量帧速率，因此帧速率计算器是一个很好的起点。

1.1 使用 FrameRate 类

FrameRate 类位于 javagames.util 包中。本书中所开发的工具代码都会放到这个工具包中，随后，我们将把这个包变成一个工具库。这个类用来测量本书中所开发的应用程序的每秒的帧数（frames per seconds, FPS）。FPS 按照“FPS 100”的格式存储为一个字符串。每秒钟都会计算这个值。

在开始测量帧速率之前，需要先调用 initialize()方法。这个方法把帧速率字符串初始化为 0，并且把最近时间初始化为当前时间（以毫秒为单位）。

System.currentTimeMillis() 调用返回了从 1970 年 1 月 1 日午夜开始的毫秒数。不同的操作系统，测量时间的精度可能不同。例如，一些 Windows 版本只能保证 10 毫秒的精度。

对于每一个渲染的帧，都应该调用一次 calculate()方法。要计算帧速率，从最近时间减去当前时间，并且将其存储到 delta 变量中。每一帧中帧计数都会增加，并且当 delta 时间超过一秒的时候，会产生新的 FPS。delta 变量很少确切地等于 1 秒钟，因此从 delta 变量减去 1000 毫秒，以略去额外的毫秒数。一旦保存了新的帧速率，就会重置帧计数并且再次开始处理。

```
package javagames.util;  
public class FrameRate {
```



```
private String frameRate;
private long lastTime;
private long delta;
private int frameCount;

public void initialize() {
    lastTime = System.currentTimeMillis();
    frameRate = "FPS 0";
}

public void calculate() {
    long current = System.currentTimeMillis();
    delta += current - lastTime;
    lastTime = current;
    frameCount++;
    if( delta > 1000 ) {
        delta-= 1000;
        frameRate = String.format( "FPS %s", frameCount );
        frameCount = 0;
    }
}

public String getFrameRate() {
    return frameRate;
}
}
```

1.2 创建 Hello World 应用程序

图 1.1 所示的 Hello World 应用程序是第一个游戏窗口的示例。HelloWorldApp 位于 javagames.render 包中。除了清除和重新绘制背景，这个窗口中不再渲染其他内容。HelloWorldApp 扩展了 JFrame 类，这是 Java 的 Swing 库中一个顶级的窗口组件。这个应用程序包含一个 FrameRate 对象，该对象用来测量应用程序的帧速率。

由于 Swing 库不是线程安全的，因此你应该总是在 Swing 事件线程上创建并展示一个 JFrame。然而，该程序的 main()方法并不是在事件线程上调用的，因

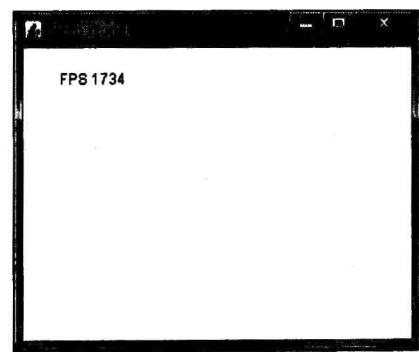


图 1.1 Hello World 应用程序



此，必须要使用 `SwingUtilities` 类来启动游戏窗口。通过使用 `SwingUtilities` 类，我们在相同的线程上创建了 GUI 组件。在使用 Swing 组件进行渲染的时候，遵从 Java 对于线程的规则是很重要的，因为忽略这些规则可能会导致不确定的行为，而这些行为是很难调试的。



我知道有的程序员会针对要在整个站点上部署的应用程序注释掉 `SwingUtilities` 代码，以测试是否真的需要 `SwingUtilities` 类来启动游戏窗口。每过几天，一些人就会报告应用程序在刚启动的时候崩溃。注意，我从来不会做这种事情，但这种做法似乎确实管用。

```
final HelloWorldApp app = new HelloWorldApp();
    SwingUtilities.invokeLater( new Runnable() {
        public void run() {
            app.createAndShowGUI();
        }
    });
}
```

在 `HelloWorldApp` 内部，是 `GamePanel` 类，它扩展了一个 `JPanel`。`GamePanel` 类用于在屏幕上绘制图形。覆盖了 `paint` 方法，这使得应用程序可以访问 `Graphics` 对象。注意，之所以能使用背景颜色来清除这个面板，只是因为调用了 `super.paint()`方法。如果把这个方法的调用去除掉，就不会再绘制背景了。`onPaint()`方法中的实际代码并没有做任何令人兴奋的事情。目前，它计算并显示了帧速率。

注意，在 `paint()`方法中，调用了 `repaint()`。如果该应用程序在其自身内部递归地调用 `paint()`，该方法将会无法返回（直到程序抛出一个栈溢出异常）。`repaint()`方法发出一个绘制请求，只要当前的绘制请求完成了，就会启动该请求。应用程序通过这种方式来尽可能快地持续绘制。

`createAndShowGUI()`方法是实际创建和显示窗口的地方。应在 `GamePanel` 上设置优先大小，而不是在 `JFrame` 上设置优先大小，这一点很重要。注意，当创建 `GamePanel` 的时候，已经设置了其优先大小。如果在 `JFrame` 上设置应用程序的大小，一些绘制区域将会被帧所占用，并且绘制区域将会变得更小一些。在面板上设置大小，可以保证它完全符合指定的大小。

在设置窗口之后，显示窗口之前，会初始化帧速率类。设置该类的初始启动时间是必须的，这样才能正确地计算第一帧。一旦通过调用 `setVisible(true)`使得应用程序变得可见，就会调用绘制方法，该方法继续计算帧速率并且重新绘制自身直到应用程序关闭。





可以在站点 <http://www.indiegameprogramming.com> 找到本书的完整的源代码。这里还包括了本书的更新和勘误，以及其他的相关信息。

HelloWorldApp 示例的代码如下所示。

```
package javagames.render;

import java.awt.*;
import javax.swing.*;
import javagames.util.*;

public class HelloWorldApp extends JFrame {

    private FrameRate frameRate;
    public HelloWorldApp() {
        frameRate = new FrameRate();
    }

    protected void createAndShowGUI() {
        GamePanel gamePanel = new GamePanel();
        gamePanel.setBackground( Color.BLACK );
        gamePanel.setPreferredSize( new Dimension( 320, 240 ) );
        getContentPane().add( gamePanel );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        setTitle( "Hello World!" );
        pack();
        frameRate.initialize();
        setVisible( true );
    }

    private class GamePanel extends JPanel {
        public void paint( Graphics g ) {
            super.paint( g );
            onPaint( g );
        }
    }

    protected void onPaint( Graphics g ) {
        frameRate.calculate();
        g.setColor( Color.WHITE );
        g.drawString( frameRate.getFrameRate(), 30, 30 );
        repaint();
    }

    public static void main( String[] args ) {
        final HelloWorldApp app = new HelloWorldApp();
```



```
SwingUtilities.invokeLater( new Runnable() {
    public void run() {
        app.createAndShowGUI();
    }
});
}
}
```

1.3 使用主动渲染

前面的示例使用了一种叫做被动渲染(*passive rendering*)的技术。该应用程序在 `paint()` 方法中重新绘制自身，但是，由 Swing 库来决定什么时候调用该方法。事件分派线程处理 Swing 组件的渲染，而这并不由你来控制。尽管这对于常规应用程序来说很好，但对于游戏不推荐这么做。

要处理渲染，可以使用 `BufferStrategy` 类，但这需要对应用程序的结构做一些修改。这将允许应用程序把渲染代码放在一个单独的线程中，并且由整个进程来控制。

把游戏代码放在 `paint()` 方法之外，通常有 3 个理由。首先，在 `paint()` 方法中不应该有任何需要很长的执行时间或阻塞时间的代码，因为这会阻止 GUI 接受绘制事件。其次，要使用全屏独占模式的话，渲染代码需要在一个不同的线程中。最后，并且也是最重要的原因，当从头开始处理绘制的时候，渲染代码会更快一些。为了利用主动渲染的好处，不管是在窗口还是全屏模式下，都要创建一个定制的渲染线程。

正如你在第一个示例中看到的那样，没有什么实用的方法将绘制代码放入到一个 `JPanel` 绘制方法中，却从定制的渲染线程调用该代码。为了在一个定制的循环中处理绘制，我们可以使用 `BufferStrategy` 类。这个类用来执行双缓冲(*double-buffering*)和页交换模式(*page-flipping*)。

如图 1.2 所示，双缓冲用来避免在进行绘制的时候，看到一幅图像的实际绘制过程。在内存中绘制图像，然后一次性复制整个图像，这样，绘制的过程就不会被看到。

可以在全屏独占模式下使用的页交换模式也采用了同样的思路，但是，它保持了两个离屏图像，并且直接从一个缓冲到另一个缓冲交换视频指针。通过这种方式，没有绘制到屏幕上的那个图像，可以被清除并重绘。当绘制完成的时候，指针再次交换，将新的图像绘制到屏幕上，如图 1.3 所示。

通过使用 `BufferStrategy` 类，程序是全屏模式并使用页交换模式，还是程序是窗口模式

并使用双缓冲，都无关紧要了——这些技术都会在幕后处理。为了在一个渲染循环中使用 `BufferStrategy`，用 `getDrawGraphics()`方法创建一个图形对象。这个图形对象将会绘制到离屏表面。一旦这个图形对象变得可用，它的使用方法完全像传入到 `JPanel` 类的 `paint()`方法中的图形对象一样。

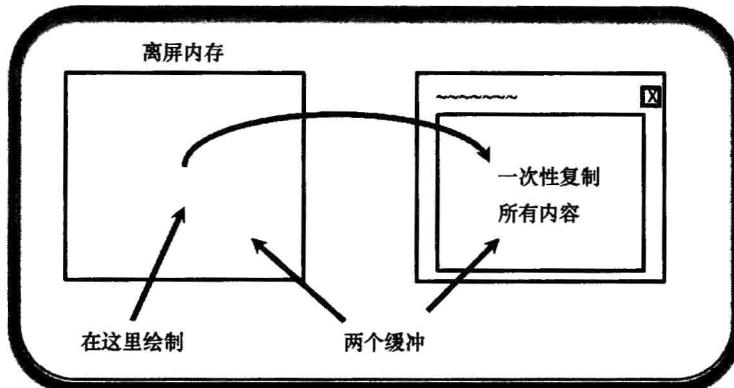


图 1.2 双缓冲

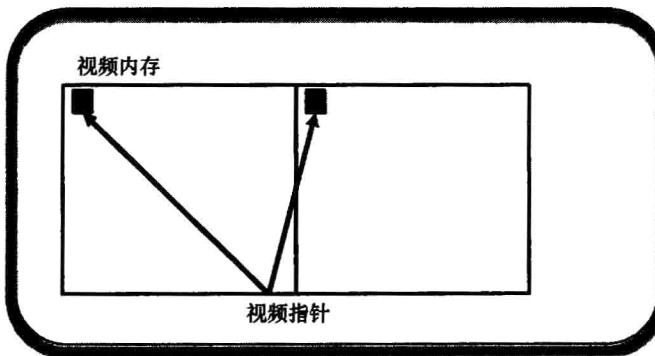


图 1.3 页交换模式

调用 `contentsLost()` 来确保离屏表面可用，这一点也是很重要的。一些操作系统允许用户通过 Alt-Tab 方式离开全屏应用程序，这会导致离屏图像变得不可用。

`show()` 方法执行双缓冲/图像复制或者页交换模式/指针交换来显示图像。注意，这段代码包含在 `try/finally` 语句块中。和其他绘制代码不同，因为这个图形对象已经创建了，所以当渲染循环完成的时候，必须要丢弃它。不调用 `Graphics.dispose()` 方法的话，将会导致内存泄露以及程序崩溃。



8

章

Hello World



应当总是确保调用 dispose() 来清理图形对象。

```
// The bs object is a BufferStrategy object, and
// will be explained later in the chapter
public void gameLoop() {
    do {
        do {
            Graphics g = null;
            try {
                g = bs.getDrawGraphics();
                g.clearRect( 0, 0, getWidth(), getHeight() );
                render( g );
            } finally {
                if( g != null ) {
                    g.dispose();
                }
            }
        } while( bs.contentsRestored() );
        bs.show();
    } while( bs.contentsLost() );
}
```

1.4 创建定制的渲染线程

除了使用前面的代码来执行定制的渲染外，还需要一个定制的渲染线程。为了只是关注线程问题，位于 `javagames.render` 包中的 `RenderThreadExample` 实际上不会在屏幕上绘制任何内容，它只是在渲染代码应该出现的地方打印出“Game Loop”。下面的示例将 `RenderThreadExample` 和前面的代码组合起来，创建了一个主动渲染的应用程序。

在 `RenderThreadExample` 中，首先需要注意的是，它实现了一个可运行的接口。这个可运行的接口包含一个单个的方法，即 `public void run()`。这个方法包含了渲染循环，并且它只调用一次。当它返回的时候，线程结束并且不能再次使用。为了防止 `run` 方法中的代码在程序完成之前退出，`Boolean` 类型的 `running` 变量保持 `run` 方法重复运行。

```
private volatile boolean running;
```



注意，该变量的声明中带有 `volatile` 关键字。由于这个变量是一个基本类型，并且可以从多个线程中访问它，因此必须要告诉编译器总是从内存中读取该变量。没有使用 `volatile` 这个关键字的话，变量可能会被 Java 虚拟机（JVM）用一个缓冲值来进行优化，并且线程可能变得无法停下来。

此外，注意 `RenderThreadExample` 中的 `sleep()` 方法，它用来将应用程序减慢到一个较为合理的运行速度。`sleep()` 方法接受一个毫秒值，然后将自身挂起到达到一定的时间，以允许其他的线程使用 CPU。

一旦应用程序关闭了，渲染线程也应该停止。通过添加一个窗口监听器，程序可以响应窗口关闭事件。在这个例子中，程序调用 `onWindowClosing()`。如果你的程序需要关闭那些不再需要的资源和文件，那么在这里做这些事情。

```
app.addWindowListener( new WindowAdapter() {
    public void windowClosing( WindowEvent e ) {
        app.onWindowClosing();
    }
});
```

要关闭渲染线程，`running` 变量应设置为 `false`。然而，渲染线程可能只是读取 `running` 变量的值，并且进入睡眠。要确保该线程已经停止了，应调用 `join()` 方法。该方法将会等待，直到该线程结束，并且 `run` 方法已经返回。如果没有向 `join()` 方法传入一个超时值，将会在线程结束前造成阻塞，因此，如果没有提供超时值的话，请确保线程将会结束。

最后，必须通过手动调用 `System.exit(0)` 来关闭程序。之前，当在 `JFrame` 中设置 `JFrame.EXIT_ON_CLOSE` 标志的时候，程序将会结束。这时应用程序负责处理关闭，只有在调用了 `exit` 方法的时候，程序才会结束。如果应用程序在关闭后挂起，通常是因为程序员忘了调用 `System.exit()`。

```
package javagames.render;
import java.awt.event.*;
import javax.swing.*;

public class RenderThreadExample extends JFrame implements Runnable {
    private volatile boolean running;
    private Thread gameThread;
    public RenderThreadExample() {
    }
    protected void createAndShowGUI() {
        setSize( 320, 240 );
        setTitle( "Render Thread" );
    }
}
```



```
setVisible( true );

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

public void run() {
    running = true;
    while( running ) {
        System.out.println( "Game Loop" );
        sleep( 10 );
    }
}
private void sleep( long sleep ) {
    try {
        Thread.sleep( sleep );
    } catch( InterruptedException ex ) { }
}
protected void onWindowClosing() {
    try {
        System.out.println( "Stopping Thread..." );
        running = false;
        gameThread.join();
        System.out.println( "Stopped!!!" );
    } catch( InterruptedException e ) {
        e.printStackTrace();
    }
    System.exit( 0 );
}
public static void main( String[] args ) {
    final RenderThreadExample app = new RenderThreadExample();
    app.addWindowListener( new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
            app.onWindowClosing();
        }
    });
    SwingUtilities.invokeLater( new Runnable() {
        public void run() {
            app.createAndShowGUI();
        }
    });
}
}
```



1.5 创建一个主动渲染的窗口

可以使用主动渲染，将渲染代码从事件分派线程移动到定制游戏线程。Windows 类和 Canvas 类是两个可用的类，它们允许创建一个 BufferStrategy。通过添加画布，我们可以访问缓冲策略，并且强迫画布的大小与所要求的大小完全一致，就像 Hello World 应用程序中的 JPanel 示例一样。

由于渲染循环为应用程序完成了所有的绘制，因此可以在 JFrame 上设置 setIgnoreRepaint() 标志。调用 Canvas.createBufferStrategy()，传入想要缓冲的数目，然后调用 Canvas.getBufferStrategy() 创建主动渲染所需的缓冲。由于应用程序正在处理绘制，因此这里不需要响应重绘方法。Component.setIgnoreRepaint() 方法负责忽略额外的绘制消息。



如果在窗口布局之前创建了 BufferStrategy 的话，将会有一个奇怪的错误发生：
Exception in thread "AWT-EventQueue-0"
java.lang.IllegalStateException: Component must have a valid
peer

调用 pack() 方法可以解决这个问题。如果抛出了前面的异常，要注意，在尝试创建 BufferStrategy 之前，确保 Component 是可见的或者已经包装了。

游戏循环现在执行渲染，就像之前的小节中所讨论的那样。使用 getDrawGraphics()、contentsLost()、show() 和 dispose() 方法，渲染循环控制了应用程序绘图。一旦图形对象变得可用，并且之前的屏幕已经清除了，就可以渲染场景了。ActiveRenderingExample 代码可以在 javagames.render 包中找到。

```
package javagames.render;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
import javagames.util.*;
public class ActiveRenderingExample extends JFrame implements Runnable {
    private FrameRate frameRate;
    private BufferStrategy bs;
    private volatile boolean running;
    private Thread gameThread;
```



```
public ActiveRenderingExample() {
    frameRate = new FrameRate();
}
protected void createAndShowGUI() {
    Canvas canvas = new Canvas();
    canvas.setSize( 320, 240 );
    canvas.setBackground( Color.BLACK );
    canvas.setIgnoreRepaint( true );
    getContentPane().add( canvas );
    setTitle( "Active Rendering" );
    setIgnoreRepaint( true );
    pack();
    setVisible( true );
    canvas.createBufferStrategy( 2 );
    bs = canvas.getBufferStrategy();
    gameThread = new Thread( this );
    gameThread.start();
}
public void run() {
    running = true;
    frameRate.initialize();
    while( running ) {
        gameLoop();
    }
}
public void gameLoop() {
    do {
        do {
            Graphics g = null;
            try {
                g = bs.getDrawGraphics();
                g.clearRect( 0, 0, getWidth(), getHeight() );
                render( g );
            } finally {
                if( g != null ) {
                    g.dispose();
                }
            }
        }
    } while( bs.contentsRestored() );
    bs.show();
} while( bs.contentsLost() );
}
private void render( Graphics g ) {
```



```
frameRate.calculate();
g.setColor( Color.GREEN );
g.drawString( frameRate.getFrameRate(), 30, 30 );
}

protected void onWindowClosing() {
    try {
        running = false;
        gameThread.join();
    } catch( InterruptedException e ) {
        e.printStackTrace();
    }
    System.exit( 0 );
}
public static void main( String[] args ) {
    final ActiveRenderingExample app = new ActiveRenderingExample();
    app.addWindowListener( new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
            app.onWindowClosing();
        }
    });
    SwingUtilities.invokeLater( new Runnable() {
        public void run() {
            app.createAndShowGUI();
        }
    });
}
}
```

1.6 修改显示模式

要创建全屏的应用程序，需要修改显示模式。DisplayModeTest 如图 1.4 所示，它使用了 Swing 组件。这个示例也位于 javagames.render 包中。如果你还没有编写过任何的 Swing 程序，一些代码可能会看上去很陌生。有很多相关的图书和教程可供参考，并且如果要详细介绍 Swing 所提供的所有功能，需要比这本书更大的篇幅。

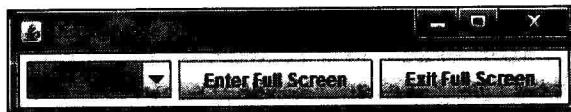


图 1.4 显示模式应用程序

这个示例应用程序列出了所有可用的屏幕分辨率，这允许用户在全屏模式之间来回切换，以使用任何可用的显示模式。这个示例不仅展示了使用 Swing 组件编程，而且这段代码还用来创建一个初始的游戏界面，以供在切换到全屏游戏之前使用。这使得用户可以使用常规的 Swing 组件来配置系统。

这个示例还包含了一个名为 `DisplayModeWrapper` 的内部类。尽管我并不痴迷于内部类，但这个类使得该示例更容易在单个文件中展示。我宁愿每个文件中只有一个类，在其他类的内部创建类以使它们可以访问私有方法这种做法很有趣，但为了简短起见，我还是另寻它途。包装类使用 `equals()` 方法只比较显示模式的宽度和高度，而不包括位深度或刷新速率。根据操作系统的不同，很多显示模式可能只是在位深度或刷新速率上有所差异：

- 640×480 32 bit 59 Hz
- 640×480 32 bit 60 Hz
- 640×480 32 bit 75 Hz
- 640×480 16 bit 59 Hz
- 640×480 16 bit 60 Hz

由于该程序把显示模式创建为 32 位并且忽略了其他模式，因此 16 位的模式是不可用的。为了让软件决定使用哪种刷新速率，使用了 `REFRESH_RATE_UNKNOWN` 值。通过这种方式，只有 `640×480` 的显示模式会被使用，而其他的模式会被略过。

构造方法包含了保存当前显示模式所需的代码。在软件启动之前，这个模式和用户的显示模式进行匹配。当应用程序返回到窗口模式的时候，使用这个显示模式来离开全屏模式。

```
public DisplayModeExample() {  
    GraphicsEnvironment ge =  
        GraphicsEnvironment.getLocalGraphicsEnvironment();  
    graphicsDevice = ge.getDefaultScreenDevice();  
    currentDisplayMode = graphicsDevice.getDisplayMode();  
}
```

`getMainPanel()`方法创建了 Swing 组件。其中列出了可用显示模式的一个下拉列表框，切换到全屏模式的一个按钮，以及切换回窗口模式的另一个按钮。`listDisplayModes()`方法



把显示模式的一个列表，返回到前面所介绍的一个包装器类中。包装器不仅允许搜索列表以找到高度和宽度相匹配的模式，还覆盖了 `toString()` 方法以生成在下拉列表框中易于读取的值。

`onEnterFullScreen()` 方法首先检查以确保支持全屏模式，然后切换到全屏模式，并且随后修改显示模式。`getSelectedMode()` 方法真正地创建一个全新的 `DisplayMode`，它带有一个未知的刷新频率，以便能够使用默认的刷新频率。

```
// DisplayModeExample.java
protected void onEnterFullScreen() {
    if( graphicsDevice.isFullScreenSupported() ) {
        DisplayMode newMode = getSelectedMode();
        graphicsDevice.setFullScreenWindow( this );
        graphicsDevice.setDisplayMode( newMode );
    }
}
```

`onExitFullScreen()` 方法使用所保存的显示模式，将显示返回到窗口模式。这些方法的调用，是按照进入全屏模式时相反的顺序来进行的。由于只有在全屏模式中才可以修改显示模式，因此在更改显示模式之前必须先切换模式，并且在最初的显示模式重置之前，不能设置回窗口模式。

```
package javagames.render;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class DisplayModeExample extends JFrame {
    class DisplayModeWrapper {
        private DisplayMode dm;
        public DisplayModeWrapper( DisplayMode dm ) {
            this.dm = dm;
        }
        public boolean equals( Object obj ) {
            DisplayModeWrapper other = (DisplayModeWrapper)obj;
            if( dm.getWidth() != other.dm.getWidth() )
                return false;
            if( dm.getHeight() != other.dm.getHeight() )
                return false;
            return true;
        }
        public String toString() {
```



```
        return "" + dm.getWidth() + " x " + dm.getHeight();
    }
}

private JComboBox displayModes;
private GraphicsDevice graphicsDevice;
private DisplayMode currentDisplayMode;
public DisplayModeExample() {
    GraphicsEnvironment ge =
        GraphicsEnvironment.getLocalGraphicsEnvironment();
    graphicsDevice = ge.getDefaultScreenDevice();
    currentDisplayMode = graphicsDevice.getDisplayMode();
}
private JPanel getMainPanel() {
    JPanel p = new JPanel();
    displayModes = new JComboBox( listDisplayModes() );
    p.add( displayModes );
    JButton enterButton = new JButton( "Enter Full Screen" );
    enterButton.addActionListener( new ActionListener() {
        public void actionPerformed( ActionEvent e ) {
            onEnterFullScreen();
        }
    });
    p.add( enterButton );
    JButton exitButton = new JButton( "Exit Full Screen" );
    exitButton.addActionListener( new ActionListener() {
        public void actionPerformed( ActionEvent e ) {
            onExitFullScreen();
        }
    });
    p.add( exitButton );
    return p;
}
private DisplayModeWrapper[] listDisplayModes() {
    ArrayList<DisplayModeWrapper> list = new
        ArrayList<DisplayModeWrapper>();
    for( DisplayMode mode : graphicsDevice.getDisplayModes() ) {
        if( mode.getBitDepth() == 32 ) {
            DisplayModeWrapper wrap = new DisplayModeWrapper( mode );
            if( !list.contains( wrap ) ) {
                list.add( wrap );
            }
        }
    }
}
```



```
    return list.toArray( new DisplayModeWrapper[0] );
}

protected void createAndShowGUI() {
    Container canvas = getContentPane();
    canvas.add( getMainPanel() );
    canvas.setIgnoreRepaint( true );
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    setTitle( "Display Mode Test" );
    pack();
    setVisible( true );
}

protected void onEnterFullScreen() {
    if( graphicsDevice.isFullScreenSupported() ) {
        DisplayMode newMode = getSelectedMode();
        graphicsDevice.setFullScreenWindow( this );
        graphicsDevice.setDisplayMode( newMode );
    }
}

protected void onExitFullScreen() {
    graphicsDevice.setDisplayMode( currentDisplayMode );
    graphicsDevice.setFullScreenWindow( null );
}

protected DisplayMode getSelectedMode() {
    DisplayModeWrapper wrapper =
        (DisplayModeWrapper)displayModes.getSelectedItem();
    DisplayMode dm = wrapper.dm;
    int width = dm.getWidth();
    int height = dm.getHeight();
    int bit = 32;
    int refresh = DisplayMode.REFRESH_RATE_UNKNOWN;
    return new DisplayMode( width, height, bit, refresh );
}

public static void main( String[] args ) {
    final DisplayModeExample app = new DisplayModeExample();
    SwingUtilities.invokeLater( new Runnable() {
        public void run() {
            app.createAndShowGUI();
        }
    });
}
```



1.7 全屏显示模式中的主动渲染

位于 `javagames.render` 包中的 `FullScreenRenderingExample`，包含了主动渲染框架和切换到全屏模式的显示模式代码；它创建了一个简单的全屏游戏框架。这个示例包含了前面各部分中展示的很多代码。此外还可以直接给 `JFrame` 设置背景颜色并且忽略重绘，以及设置 `setUndecorated()` 标志。由于在前面的示例中应用程序是从窗口模式切换到全屏模式的，因此没有设置该标志；但是当只使用全屏模式的时候，应该对 `JFrame` 进行该项设置。

保存当前的显示模式，切换到全屏模式，并且修改显示模式之后，应使用 `JFrame` 方法而不是窗口模式示例中的 `Canvas` 方法来创建缓冲策略。

即便还没有涉及键盘，但你还是需要知道退出程序的一些方法。因为 `JFrame` 是未装饰的，所以没有控件能够关闭窗口。当用户按下 `Escape` 键的时候，如下的代码将会关闭应用程序。

```
// FullScreenRenderingExample.java
addKeyListener( new KeyAdapter() {
    public void keyPressed( KeyEvent e ) {
        if( e.getKeyCode() == KeyEvent.VK_ESCAPE ) {
            shutDown();
        }
    }
});
```

在这个示例中，为了简单起见，显示模式直接编码为 `800x600`、`32` 位。在实际的产品级应用程序中，可用的显示模式应该像前面例子中那样进行枚举。如果你的系统不支持这种显示模式，请确保修改你的代码。

```
private DisplayMode getDisplayMode() {
    return new DisplayMode(
        800, 600, 32, DisplayMode.REFRESH_RATE_UNKNOWN );
}
```

由于没有办法关闭该窗口，也就不需要有一个窗口监听器。当按下 `Escape` 键并且示例关闭时，在游戏循环关闭后，显示模式返回为常规模式。

```
package javagames.render;
import java.awt.*;
import java.awt.event.*;
```



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

public class FullScreenRenderingExample
    extends JFrame implements Runnable {
    private FrameRate frameRate;
    private BufferStrategy bs;
    private volatile boolean running;
    private Thread gameThread;
    private GraphicsDevice graphicsDevice;
    private DisplayMode currentDisplayMode;
    public FullScreenRenderingExample() {
        frameRate = new FrameRate();
    }
    protected void createAndShowGUI() {
        setIgnoreRepaint( true );
        setUndecorated( true );
        setBackground( Color.BLACK );
        GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        graphicsDevice = ge.getDefaultScreenDevice();
        currentDisplayMode = graphicsDevice.getDisplayMode();
        if( !graphicsDevice.isFullScreenSupported() ) {
            System.err.println( "ERROR: Not Supported!!!" );
            System.exit( 0 );
        }
        graphicsDevice.setFullScreenWindow( this );
        graphicsDevice.setDisplayMode( getDisplayMode() );
        createBufferStrategy( 2 );
        bs = getBufferStrategy();
        addKeyListener( new KeyAdapter() {
            public void keyPressed( KeyEvent e ) {
                if( e.getKeyCode() == KeyEvent.VK_ESCAPE ) {
                    shutDown();
                }
            }
        });
        gameThread = new Thread( this );
        gameThread.start();
    }
    private DisplayMode getDisplayMode() {
        return new DisplayMode(
```



```
    800, 600, 32, DisplayMode.REFRESH_RATE_UNKNOWN );
}
public void run() {
    running = true;
    frameRate.initialize();
    while( running ) {
        gameLoop();
    }
}
public void gameLoop() {
    do {
        do {
            Graphics g = null;
            try {
                , g = bs.getDrawGraphics();
                g.clearRect( 0, 0, getWidth(), getHeight() );
                render( g );
            } finally {
                if( g != null ) {
                    g.dispose();
                }
            }
        } while( bs.contentsRestored() );
        bs.show();
    } while( bs.contentsLost() );
}
private void render( Graphics g ) {
    frameRate.calculate();
    g.setColor( Color.GREEN );
    g.drawString( frameRate.getFrameRate(), 30, 30 );
    g.drawString( "Press ESC to exit...", 30, 60 );
}
protected void shutDown() {
    try {
        running = false;
        gameThread.join();
        System.out.println( "Game loop stopped!!!" );
        graphicsDevice.setDisplayMode( currentDisplayMode );
        graphicsDevice.setFullScreenWindow( null );
        System.out.println("Display Restored...");
```



```
        System.exit( 0 );
    }
    public static void main( String[] args ) {
        final FullScreenRenderingExample app = new
            FullScreenRenderingExample();
        SwingUtilities.invokeLater( new Runnable() {
            public void run() {
                app.createAndShowGUI();
            }
        });
    }
}
```

1.8 资源和延伸阅读

Martak, Michael, “Full-Screen Exclusive Mode API,” 1995, <http://docs.oracle.com/javase/tutorial/extr fullscreen/index.html>.

第2章

输入

如图 2.1 所示，输入对于视频游戏来说是非常重要的。游戏之所以与电影不同，在于游戏有输入，而这也是本章的主题。尽管很多人在游戏的外观上投入了较多的精力，但不管游戏外观看上去有多么壮观，如果操控不稳定、糟糕或很难的话，玩家会感到非常沮丧。

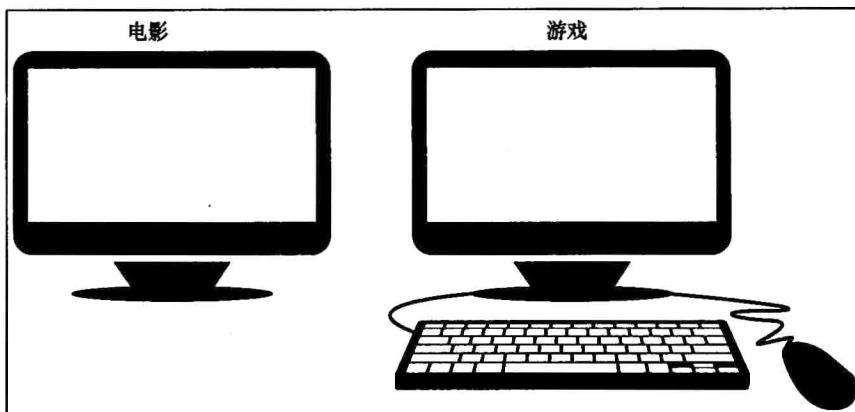


图 2.1 电影和游戏之间的差异

尽管如今有很多的输入设备可供使用，但本书只介绍最常见的两种：键盘和鼠标。

2.1 处理键盘输入

在大多数应用程序中，软件都不需要处理键盘事件。当某些事情发生变化的时候，由任意的组件（如文本框）来处理输入并通知软件。但是，大多数计算机游戏使用键盘不是为了录入，而是为了游戏输入。根据游戏的不同，虽然可能会有录入，但键盘按键常用做方向键和发射激光武器。很多计算机游戏具有不同的输入配置，并且有些游戏甚至允许用



户根据自己的意愿来设置按键。不管游戏如何使用键盘（并且它可能采用一种全新的方式来使用键盘，监听键盘事件的常用方法都不适用于游戏循环程序设计。

Swing 组件允许在实现了 `KeyListener` 的接口的对象上添加监听器。

- `void keyTyped(KeyEvent e)`—按下并释放键。
- `void keyPressed(KeyEvent e)`—按下键。
- `void keyReleased(KeyEvent e)`—释放键。

按下一个按键的时候，调用 `keyPressed()` 方法。正如所预期的那样，释放按键的时候，调用 `keyReleased()` 方法。只有在按键按下并释放之后，才会调用 `keyTyped()` 方法。动作按键和修饰按键，例如 Shift 按键和方向箭头按键，不会产生 `keyTyped()` 事件。第 11 章介绍文本的时候将会讨论这一事件。

问题在于，键盘是由操作系统维护的一种硬件。由操作系统而不是软件来产生键盘事件，并将其分派给相关的应用程序。没什么办法能够阻止用户从游戏窗口切换回 Web 浏览器并查看 Email。因此，所有的键盘事件都通过一个不同的线程到达，并且可供游戏循环使用。

大多数游戏遵从某种循环结构：

```
while( true ) {  
    processInput();  
    updateObjects();  
    // other stuff...  
    renderScene();  
}
```

如果在游戏循环之外处理输入，状态可能会随时改变。此外，还可能会同时按下多个按键，因此，处理每个事件自身并不允许用户组合按键。为了简化输入过程，应保存键盘事件并使其可供游戏循环使用。

存储键盘状态时，理解程序如何共享键盘的状态是很重要的。键盘是非常复杂的硬件，不仅那些字符串字符可用，甚至每个并不代表字符的按键（例如 Shift 键），也可以通过虚拟按键代码变得可用。键盘上的每个按键都映射为 `KeyEvent` 类中的一个键代码。如下是一些示例值。

- `KeyEvent.VK_E`—E 键。
- `KeyEvent.VK_SPACE`—空格键。
- `KeyEvent.VK_UP`—向上箭头键。

这些常量中的每一个，都映射为传递给 `KeyEvent` 对象中的按键监听器的一个数字值。针对产生事件的任何键盘按键，`KeyEvent.getKeyCode()` 方法都返回虚拟的键代码。



SimpleKeyboardInput 类位于 javagames.util 包中，它非常小。这个类实现了 KeyListener 接口，因此，它可以监控键盘事件。它保存了 256 个键的一个 Boolean 数组，其中都是需要取样的虚拟键代码。在键盘状态数组中，存储了键的状态，如果按下的话是 true，否则就是 false。最后，使用 synchronized 关键字来防止从多个线程访问键状态数组，然后，通过 keyDown(int keyCode)方法允许访问当前按键状态。

```
package javagames.util;
import java.awt.event.*;
public class SimpleKeyboardInput implements KeyListener {
    private boolean[] keys;
    public SimpleKeyboardInput() {
        keys = new boolean[ 256 ];
    }
    public synchronized boolean keyDown( int keyCode ) {
        return keys[ keyCode ];
    }
    public synchronized void keyPressed( KeyEvent e ) {
        int keyCode = e.getKeyCode();
        if( keyCode >= 0 && keyCode < keys.length ) {
            keys[ keyCode ] = true;
        }
    }
    public synchronized void keyReleased( KeyEvent e ) {
        int keyCode = e.getKeyCode();
        if( keyCode >= 0 && keyCode < keys.length ) {
            keys[ keyCode ] = false;
        }
    }
    public void keyTyped( KeyEvent e ) {
        // Not needed
    }
}
```



别忘了，你可以在本书的 Web 站点 <http://www.indiegameprogramming.com> 找到本书的所有源代码。

SimpleKeyboardExample 类位于 javagames.input 包中，它是使用键盘输入类的一个简单测试，它使用输入处理代码来替代渲染代码。注意，使用 addKeyListener()方法将 SimpleKeyboardInput 添加到应用程序中。在游戏循环中，游戏循环会检查空格或箭头按键，



并且在这些按键按下时打印一条消息，而不是清除图像并显示帧速率。

当检查到空格时，示例使用一个变量来保存按键的状态，针对每次按键只打印到控制台一次。而对于箭头按键，会持续将其状态输出到控制台，直到按键释放。

```
package javagames.input;

import java.awt.event.*;
import javax.swing.*;
import javagames.util.*;

public class SimpleKeyboardExample extends JFrame implements Runnable {
    private volatile boolean running;
    private Thread gameThread;
    private SimpleKeyboardInput keys;
    private boolean space;
    public SimpleKeyboardExample() {
        keys = new SimpleKeyboardInput();
    }
    protected void createAndShowGUI() {
        setTitle( "Keyboard Input" );
        setSize( 320, 240 );
        addKeyListener( keys );
        setVisible( true );
        gameThread = new Thread( this );
        gameThread.start();
    }
    public void run() {
        running = true;
        while( running ) {
            gameLoop();
        }
    }
    public void gameLoop() {
        if( keys_KeyDown( KeyEvent.VK_SPACE ) ) {
            if( !space ) {
                System.out.println( "VK_SPACE" );
            }
            space = true;
        } else {
            space = false;
        }
        if( keys_KeyDown( KeyEvent.VK_UP ) ) {
            System.out.println( "VK_UP" );
        }
        if( keys_KeyDown( KeyEvent.VK_DOWN ) ) {
```



```
        System.out.println( "VK_DOWN" );
    }
    if( keys_KeyDown( KeyEvent.VK_LEFT ) ) {
        System.out.println( "VK_LEFT" );
    }
    if( keys_KeyDown( KeyEvent.VK_RIGHT ) ) {
        System.out.println( "VK_RIGHT" );
    }
    try {
        Thread.sleep( 10 );
    } catch( InterruptedException ex ) { }
}
protected void onWindowClosing() {
    try {
        running = false;
        gameThread.join();
    } catch( InterruptedException e ) {
        e.printStackTrace();
    }
    System.exit( 0 );
}
public static void main( String[] args ) {
    final SimpleKeyboardExample app = new SimpleKeyboardExample ();
    app.addWindowListener( new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
            app.onWindowClosing();
        }
    });
    SwingUtilities.invokeLater( new Runnable() {
        public void run() {
            app.createAndShowGUI();
        }
    });
}
```

2.2 键盘改进

尽管键盘输入类允许在游戏循环中访问键盘状态，但是实现起来还是有一些问题。首



先，游戏循环代码执行的时候，如果键盘按键没有按下，将会错过 `keypress` 事件。尽管对于这些简单的示例来说，不太可能发生这种情况，但当应用程序变得更强大并且游戏循环需要更多的时间来处理代码时，游戏循环就可能变得太慢而导致错过了输入。现在，只需要知道这可能是一个问题就行了。我们将会在第 11 章中讨论确保事件不会被错过的一种解决方案。

第二个问题是，当按键第一次被按下的时候，很难进行测试。如果需要跟踪 20 个按键，并且其中一些按键会根据游戏的状态而改变行为，那么，试图使用类似下面的代码来跟踪所有这些状态，可能会变得非常糟糕：

```
// SimpleKeyboardExample.java
if( keys_KeyDown( KeyEvent.VK_SPACE ) ) {
    if( !space ) {
        System.out.println( "VK_SPACE" );
    }
    space = true;
} else {
    space = false;
}
```

为了更新 `KeyboardInput` 类来跟踪初始的按键事件以及键盘状态，添加了一个整型数值的数组。这些值将会记录按键被按下了多少帧。实现 `KeyListener` 接口的代码并不会改变，但是 `keyDown()` 方法不再会从布尔的按键数组提取值。

`poll()` 方法同步地保护共享的按键数组，将键盘状态从布尔型数组转换为整型数组。如果按键按下，这个值将会增加 1；否则的话，它将会设置为 0。现在，`keyDown()` 方法将检测这个值是否为 0，并且当这个值确实为 1 的时候，新的方法 `keyDownOnce()` 将会返回真。

```
package javagames.util;
import java.awt.event.*;
public class KeyboardInput implements KeyListener {
    private boolean[] keys;
    private int[] polled;
    public KeyboardInput() {
        keys = new boolean[ 256 ];
        polled = new int[ 256 ];
    }
    public boolean keyDown( int keyCode ) {
        return polled[ keyCode ] > 0;
    }
}
```



```
public boolean keyDownOnce( int keyCode ) {
    return polled[ keyCode ] == 1;
}
public synchronized void poll() {
    for( int i = 0; i < keys.length; ++i ) {
        if( keys[i] ) {
            polled[i]++;
        } else {
            polled[i] = 0;
        }
    }
}
public synchronized void keyPressed( KeyEvent e ) {
    int keyCode = e.getKeyCode();
    if( keyCode >= 0 && keyCode < keys.length ) {
        keys[ keyCode ] = true;
    }
}
public synchronized void keyReleased( KeyEvent e ) {
    int keyCode = e.getKeyCode();
    if( keyCode >= 0 && keyCode < keys.length ) {
        keys[ keyCode ] = false;
    }
}
public void keyTyped( KeyEvent e ) {
    // Not needed
}
```



别忘了，针对每一帧要调用 KeyboardInput.poll() 方法。

通过使用如下代码替换前面例子中的游戏循环代码，可以使用这个新的类。检查空格键按下一次所需的代码大大简化了。

```
// replacing the game loop
public void gameLoop() {
    keys.poll();
    if( keys.keyDownOnce( KeyEvent.VK_SPACE ) ) {
        System.out.println( "VK_SPACE" );
    }
}
```



```
if( keys.keyDown( KeyEvent.VK_UP ) ) {  
    System.out.println( "VK_UP" );  
}  
if( keys.keyDown( KeyEvent.VK_DOWN ) ) {  
    System.out.println( "VK_DOWN" );  
}  
if( keys.keyDown( KeyEvent.VK_LEFT ) ) {  
    System.out.println( "VK_LEFT" );  
}  
if( keys.keyDown( KeyEvent.VK_RIGHT ) ) {  
    System.out.println( "VK_RIGHT" );  
}  
try {  
    Thread.sleep( 10 );  
} catch( InterruptedException ex ) { }  
}
```

2.3 处理鼠标输入

SimpleMouseInput 类位于 javagames.util 包中，它和前面小节中开发的键盘输入类非常相似。处理鼠标按键的方式与处理键盘按键的方式相同。实现 MouseListener 接口的类包含了如下的方法：

```
mouseClicked(MouseEvent e)  
mouseEntered(MouseEvent e)  
mouseExited(MouseEvent e)  
mousePressed(MouseEvent e)  
mouseReleased(MouseEvent e)
```

这个接口中的两个方法，mouseEntered() 和 mouseExited()，负责处理鼠标光标移动。另外 3 个方法用于鼠标按键。就像键盘监听器一样，按下和释放方法会记录鼠标按钮的状态，而点击方法则会被忽略。为了确定按下了哪个鼠标按键，鼠标事件包含了如下的方法：

```
public int MouseEvent.getButton()
```

这个方法所返回的值，映射到如下的常量：

```
MouseEvent.NOBUTTON = 0  
MouseEvent.BUTTON1 = 1
```



```
MouseEvent.BUTTON2 = 2  
MouseEvent.BUTTON3 = 3
```

按键的编号是从 1 开始而不是从 0 开始的，0 表示没有按键，在引用鼠标按键数组的时候，不需要将按键编号减 1。除了这一点小小的差别，鼠标按键和键盘按键的处理方式相同，也包括 `buttonDown()` 和 `buttonDownOnce()` 方法。

还有其他的鼠标状态可供程序使用，例如鼠标指针的位置和状态。和 `MouseListener` 接口的 `mouseEntered()` 和 `mouseExited()` 方法一样，鼠标输入类也实现了 `MouseMotionListener` 接口。

```
mouseDragged(MouseEvent e)  
mouseMoved(MouseEvent e)
```

针对鼠标事件，如果鼠标进入或离开了组件监听，所有这 4 个方法（进入、退出、拖拽和移动）都会捕获鼠标的位置并通知程序。如下的方法会获取鼠标的当前位置：

```
public Point MouseEvent.getPoint()
```

当轮询鼠标输入时，会复制这个值并使其可供游戏循环使用。当前的鼠标位置，对于如下的方法来说是可用的：

```
public Point SimpleMouseInput.getPosition()
```

最后，为了监控鼠标滚轮的输入，输入类实现了 `MouseWheelListener`：

```
mouseWheelMoved(MouseEvent e)
```

如下的方法返回了鼠标滚轮的点击。如果这个数值是负值，表示滚轮已经从用户那里移走了。如果这个值是正的，表示滚轮已经朝着用户移动了。

```
public int MouseWheelEvent.getWheelRotation()
```

这个值也保存在 `poll()` 方法中，并且可通过如下方法供游戏循环使用：

```
public int SimpleMouseInput.getNotches()
```

`SimpleMouseInput` 类的代码如下：

```
package javagames.util;  
import java.awt.*;  
import java.awt.event.*;  
  
public class SimpleMouseInput  
implements MouseListener, MouseMotionListener, MouseWheelListener {  
    private static final int BUTTON_COUNT = 3;  
    private Point mousePos;  
    private Point currentPos;
```



```
private boolean[] mouse;
private int[] polled;
private int notches;
private int polledNotches;
public SimpleMouseInput() {
    mousePos = new Point( 0, 0 );
    currentPos = new Point( 0, 0 );
    mouse = new boolean[ BUTTON_COUNT ];
    polled = new int[ BUTTON_COUNT ];
}
public synchronized void poll() {
    mousePos = new Point( currentPos );
    polledNotches = notches;
    notches = 0;
    for( int i = 0; i < mouse.length; ++i ) {
        if( mouse[i] ) {
            polled[i]++;
        } else {
            polled[i] = 0;
        }
    }
}
public Point getPosition() {
    return mousePos;
}
public int getNotches() {
    return polledNotches;
}
public boolean buttonDown( int button ) {
    return polled[ button - 1 ] > 0;
}
public boolean buttonDownOnce( int button ) {
    return polled[ button - 1 ] == 1;
}
public synchronized void mousePressed( MouseEvent e ) {
    int button = e.getButton() - 1;
    if( button >= 0 && button < mouse.length ) {
        mouse[ button ] = true;
    }
}
public synchronized void mouseReleased( MouseEvent e ) {
    int button = e.getButton() - 1;
    if( button >= 0 && button < mouse.length ) {
```



```
        mouse[ button ] = false;
    }
}

public void mouseClicked( MouseEvent e ) {
    // Not needed
}

public synchronized void mouseEntered( MouseEvent e ) {
    mouseMoved( e );
}

public synchronized void mouseExited( MouseEvent e ) {
    mouseMoved( e );
}

public synchronized void mouseDragged( MouseEvent e ) {
    mouseMoved( e );
}

public synchronized void mouseMoved( MouseEvent e ) {
    currentPos = e.getPoint();
}

public synchronized void mouseWheelMoved( MouseWheelEvent e ) {
    notches += e.getWheelRotation();
}
}
```

SimpleMouseExample 如图 2.2 所示，它位于 javagames.input 包中，为第一个示例添加了很多新的内容。这个示例以主动渲染示例代码为基础，添加了键盘和鼠标输入类；它不仅展示了所有 3 种类型的鼠标输入，而且也是使用 Graphics 对象绘制到屏幕的第一个示例。

首先需要注意的是，创建 GUI 时的方法调用。这些调用添加了 KeyboardInput 和 SimpleMouseInput 作为组件的监听器。注意，尽管给 JFrame 和 Canvas 都添加了键盘，但是只给 Canvas 添加了鼠标。当应用程序初次启动时，如果没有给 JFrame 添加键盘事件的话，它不会处理键盘事件。一旦画布对象接受到焦点，它将会接受键盘输入，但在画布被选中之前，都只有 JFrame 接受键盘输入。在游戏循环中，添加了如下的方法调用：

```
public void processInput()
```

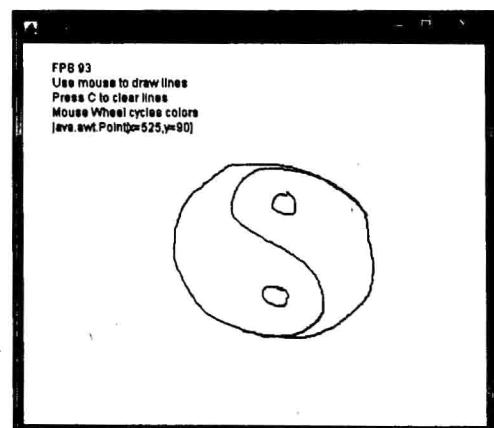


图 2.2 简单鼠标示例



在这个方法中，对键盘和鼠标都进行了轮询，以确保它们的数据可用。当初次按下鼠标按键时，会设置绘制标志。对于保持鼠标按键按下的每一帧，都会向该行的数据结构添加一个新的点。当按键释放的时候，该标志被清除，并且向列表添加一个空的对象，标志该行的结束。通过这种方式，该数据结构可以同时保存所有的行。

当按下 C 键的时候，该行从数据结构中清除。如果用户缺乏绘画技巧的话（就像我一样），这会允许他们重新开始。渲染方法也有新的代码加入。除了显示帧速率，还给出了使用该应用程序的说明，并把当前鼠标的位置显示为字符串。

鼠标滚轮用来选择一个颜色索引。如下的代码使用模除运算符和绝对值来保证索引是一个有效值，无论所获取的索引有多大或多少。

```
// SimpleMouseExample.java
colorIndex += mouse.getNotches();
Color color = COLORS[ Math.abs( colorIndex % COLORS.length ) ];
g.setColor( color );
```

%操作符将会保证值在(-3, 3)之间。因为对一个负值进行模除运算，会得到 0 或者一个负值，所以使用绝对值来保证数组索引位于(0, size -1)之间。

最后，绘制了线条。由于在数据结构中插入了空的值，我们添加了代码以确保没有点为空的时候才绘制线条。

```
package javagames.input;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.util.*;
import javagames.util.*;
import javax.swing.*;

public class SimpleMouseExample extends JFrame
    implements Runnable {
    private FrameRate frameRate;
    private BufferStrategy bs;
    private volatile boolean running;
    private Thread gameThread;
    private SimpleMouseInput mouse;
    private KeyboardInput keyboard;
    private ArrayList<Point> lines = new ArrayList<Point>();
    private boolean drawingLine;
    private Color[] COLORS = {
        Color.RED,
        Color.GREEN,
```



```
        Color.YELLOW,
        Color.BLUE
    };
    private int colorIndex;
    public SimpleMouseExample() {
        frameRate = new FrameRate();
    }
    protected void createAndShowGUI() {
        Canvas canvas = new Canvas();
        canvas.setSize( 640, 480 );
        canvas.setBackground( Color.BLACK );
        canvas.setIgnoreRepaint( true );
        getContentPane().add( canvas );
        setTitle( "Simple Mouse Example" );
        setIgnoreRepaint( true );
        pack();
        // Add key listeners
        keyboard = new KeyboardInput();
        canvas.addKeyListener( keyboard );
        // Add mouse listeners
        mouse = new SimpleMouseInput();
        canvas.addMouseListener( mouse );
        canvas.addMouseMotionListener( mouse );
        canvas.addMouseWheelListener( mouse );
        setVisible( true );
        canvas.createBufferStrategy( 2 );
        bs = canvas.getBufferStrategy();
        canvas.requestFocus();
        gameThread = new Thread( this );
        gameThread.start();
    }
    public void run() {
        running = true;
        frameRate.initialize();
        while( running ) {
            gameLoop();
        }
    }
    private void gameLoop() {
        processInput();
        renderFrame();
        sleep( 10L );
    }
    private void renderFrame() {
        do {
```



```
do {
    Graphics g = null;
    try {
        g = bs.getDrawGraphics();
        g.clearRect( 0, 0, getWidth(), getHeight() );
        render( g );
    } finally {
        if( g != null ) {
            g.dispose();
        }
    }
} while( bs.contentsRestored() );
bs.show();
} while( bs.contentsLost() );
}
private void sleep( long sleep ) {
try {
    Thread.sleep( sleep );
} catch( InterruptedException ex ) { }
}
private void processInput() {
    keyboard.poll();
    mouse.poll();
    if( keyboard_KeyDownOnce( KeyEvent.VK_SPACE ) ) {
        System.out.println("VK_SPACE");
    }
    // if button is pressed for first time,
    // start drawing lines
    if( mouse.buttonDownOnce( MouseEvent.BUTTON1 ) ) {
        drawingLine = true;
    }
    // if the button is down, add line point
    if( mouse.buttonDown( MouseEvent.BUTTON1 ) ) {
        lines.add( mouse.getPosition() );
        // if the button is not down but we were drawing,
        // add a null to break up the lines
    } else if( drawingLine ) {
        lines.add( null );
        drawingLine = false;
    }
    // if 'C' is down, clear the lines
    if( keyboard_KeyDownOnce( KeyEvent.VK_C ) ) {
        lines.clear();
    }
}
```



```
private void render( Graphics g ) {
    colorIndex += mouse.getNotches();
    Color color = COLORS[ Math.abs( colorIndex % COLORS.length ) ];
    g.setColor( color );
    frameRate.calculate();
    g.drawString( frameRate.getFrameRate(), 30, 30 );
    g.drawString( "Use mouse to draw lines", 30, 45 );
    g.drawString( "Press C to clear lines", 30, 60 );
    g.drawString( "Mouse Wheel cycles colors", 30, 75 );
    g.drawString( mouse.getPosition().toString(), 30, 90 );
    for( int i = 0; i < lines.size() - 1; ++i ) {
        Point p1 = lines.get( i );
        Point p2 = lines.get( i + 1 );
        // Adding a null into the list is used
        // for breaking up the lines when
        // there are two or more lines
        // that are not connected
        if( !( p1 == null || p2 == null ) )
            g.drawLine( p1.x, p1.y, p2.x, p2.y );
    }
}
protected void onWindowClosing() {
    try {
        running = false;
        gameThread.join();
    } catch( InterruptedException e ) {
        e.printStackTrace();
    }
    System.exit( 0 );
}
public static void main( String[] args ) {
    final SimpleMouseExample app = new SimpleMouseExample();
    app.addWindowListener( new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
            app.onWindowClosing();
        }
    });
    SwingUtilities.invokeLater( new Runnable() {
        public void run() {
            app.createAndShowGUI();
        }
    });
}
```